
SimplIQ

Programming and Language Guide



May 2009



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 *SimplIQ* servo drives in implementing CANopen networking.
- 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. Corporate and individual names and data used in examples herein are fictitious unless otherwise noted.

Doc. No. MAN-SIMPG
Copyright © 2009
Elmo Motion Control Ltd.
All rights reserved.

Revision History:

Ver. 1.1 May 2009 Minor updates.

Ver. 1.0 Jan. 2008 Initial release.

Elmo Motion Control Ltd. 64 Gisin St., P.O. Box 463 Petach Tikva 49103 Israel Tel: +972 (3) 929-2300 Fax: +972 (3) 929-2322 info-il@elmomc.com	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	Elmo Motion Control GmbH Steinkirchring 1 D-78056, Villingen-Schwenningen Germany Tel: +49 (0) 7720-85 77 60 Fax: +49 (0) 7720-85 77 70 info-de@elmomc.com	 www.elmomc.com
---	---	---	---

Contents

Chapter 1: Introduction.....	1-1
1.1 Scope	1-1
1.2 Glossary	1-1
Chapter 2: The Interpreter Language.....	2-1
2.1 The Command Line.....	2-1
2.2 Expressions and Operators.....	2-2
2.2.1 Numbers.....	2-2
2.2.2 Mathematical and Logical Operators	2-3
2.2.3 General Rules for Operators.....	2-4
2.2.4 Operator Details	2-5
2.2.5 Mathematical Functions.....	2-6
2.2.6 Expressions	2-7
2.2.7 Comments	2-11
Chapter 3: The <i>SimplIQ</i> User Programming Language.....	3-1
3.1 User Program Organization	3-1
3.2 Single and Multiple Command Execution.....	3-3
3.3 Standard Conventions.....	3-3
3.3.1 Line and Expression Termination.....	3-3
3.3.2 Line Continuation	3-4
3.3.3 Limitations	3-4
3.4 Expressions and Operators.....	3-4
3.4.1 Numbers.....	3-4
3.4.2 Mathematical and Logical Operators	3-5
3.4.3 General Operator Rules	3-5
3.4.4 Operator Details	3-5
3.4.5 Mathematical Functions.....	3-5
3.4.6 Exclusive OR Operation	3-5
3.4.7 CAN Object Emission	3-6
3.4.8 Expressions	3-8
3.5 Comments.....	3-11
3.5.1 Double Asterisk.....	3-11
3.5.2 Double Slash	3-11
3.5.3 C-style Start and End Comment	3-11
3.6 Fault Handling.....	3-12
3.6.1 Unexpected Fault.....	3-12
3.6.2 Expected Fault	3-12
3.7 Program Flow Commands.....	3-13
3.7.1 Labels (Entry Points) and Subroutines.....	3-13
3.7.2 For Iteration	3-15
3.7.3 While Iteration.....	3-16
3.7.4 Until Iteration	3-16
3.7.5 Wait Iteration.....	3-17
3.7.6 If Condition.....	3-18
3.7.7 Switch Selection.....	3-18
3.7.8 Continue.....	3-19

3.7.9	Break	3-20
3.7.10	Return	3-21
3.7.11	Try-Catch.....	3-21
3.8	Functions	3-22
3.8.1	Function Declaration.....	3-22
3.8.2	Dummy Variables.....	3-25
3.8.3	Count of Output Variables	3-25
3.8.4	Automatic Variables.....	3-26
3.8.5	Global Variables	3-26
3.8.6	Jumps.....	3-27
3.8.7	Functions and the Call Stack.....	3-28
3.8.8	Killing the Call Stack.....	3-29
3.8.9	Automatic Subroutines	3-30
Chapter 4: Program Development and Execution.....		4-1
4.1	Editing a Program.....	4-1
4.2	Compilation.....	4-1
4.3	Compiler directives	4-14
4.3.1	#define.....	4-15
4.3.2	#if	4-17
4.3.3	#else	4-17
4.3.4	#elseif.....	4-18
4.3.5	#endif.....	4-18
4.3.6	#ifdef.....	4-18
4.3.7	#ifndef	4-19
4.3.8	#undef	4-19
4.3.9	Evaluating Expressions Used in Compiler Directives.....	4-20
4.4	Program Execution	4-20
4.4.1	Initiating a Program	4-20
4.4.2	Halting and Resuming a Program.....	4-21
4.4.3	Automatic Program Execution with Power Up	4-22
4.4.4	Save to Flash	4-22
4.4.5	Running, Breaking and Resuming.....	4-22
4.4.6	The Elmo Studio	4-23
Chapter 5: User Program Maintenance.....		5-1
5.1	Downloading and Uploading a Program	5-1
5.1.1	Binary Data	5-1
5.1.2	Auxiliary Upload/Download Commands	5-2
5.1.3	Downloading a Program	5-3
5.1.4	Uploading a Program.....	5-4

Chapter 1: Introduction

1.1 Scope

The manual describes the user programming language for Elmo's *SimpliIQ* drives.

1.2 Glossary

The following terms are used in this document:

Download	Transfer of data from the host to the drive.
DSP	Digital signal processor.
EDS	Electronic data sheet. The list of CAN objects supported by a device, in a form suitable for standard configuration software.
IDE	Integrated development environment.
PDO	Process data object. A CAN message type, which eliminates the need for allocating the data payload for object addressing by pre-agreement concerning the message contents (PDO mapping).
Upload	Transfer of data from the drive to the host.

Chapter 2: The Interpreter Language

SimplIQ 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 status.

Two methods can be used to communicate with the drive:

- A communication interface – either RS-232 or CANopen – 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 physics and standards of RS-232 and CANopen communication require different command syntax for each method. This chapter describes the drive language according to basic RS-232 or CAN “OS” syntax.
- Writing a program in the drive language and storing it in the drive memory. The drive can then run the program with minimal or no host assistance.

The CANopen communication method can access simple numeric interpreter “get” and “set” commands very efficiently. The CAN binary interpreter uses PDO objects to issue interpreter commands and to collect the responses. This is the most efficient way to minimize both the communication load and the drive CPU load.

The CAN OS (command prompt) method 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 method is beyond the scope of this manual (it is covered in the Elmo *CAN Implementation Manual*).

Software programs use the interpreter syntax, with extensions for support of program flow instructions and in-line documentation.

The full set of drive commands is documented in the *SimplIQ for Steppers Command Reference Manual*.

2.1 The Command Line

The Interpreter evaluates input strings, called “expressions,” which are sequences of characters, terminated by a semicolon (;), a line feed or a carriage return.

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

A command line may include a comment marker, which is 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, sent directly to the drive later.

Example:

Command Line	Results	Remarks
3+4;	7	
PX=7; PX-3;	4	PX is set to 7 and 3 is then subtracted.
(3.2+4)/2;	3.6	

2.2 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 almost any way to create an expression. The following sections describe the operators and expression syntax rules.

2.2.1 Numbers

SimplIQ drives use two number types: 32-bit integers and 32-bit floating-point numbers (“floats”). At text inputs, numbers containing a decimal point or an exponent notation are interpreted as floats. Other numbers are interpreted as integers.

The range for integers is [-2,147,483,648...2,147,483,647]. If an integer number exceeds the integer range, it is interpreted as an error. For example, if 2,147,483,648 is entered, the *SimplIQ* drive will respond with a Bad Command Format error.



The lowest integer - 2,147,483,648 - cannot be entered explicitly through the interpreter due to the means by which immediate numbers are internally evaluated. Nevertheless, this integer value is valid and can be entered in hexadecimal form as 0x80,000,000.

Positive integers may be written as decimal or as hexadecimal.

The hexadecimal notation 0x10 is equivalent to the decimal number 16.

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

The range for floating-point numbers is [-1e20...1e20].

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

2.5e4 is equivalent to 25,000.0. It is *not* equivalent to 25,000, because the latter number is interpreted as an integer. If a floating-point number exceeds the floating-point range, it is also interpreted as an error.



SimplIQ drives evaluate floating-point numbers with the standard IEEE floating point precision of approximately six significant decimal digits. For example, the number 12,345.0 has an exact IEEE floating point representation. The number 1,234,568.0 is understood by the *SimplIQ* drive to be 12,345,680.0 due to truncation into the IEEE float format.

SimplIQ drives cannot evaluate numbers with an absolute value greater than 10^{20} . For example, if you enter =12.3e+20 or -13.56e-20, the *SimplIQ* drive will respond with a “Badly Formatted Number” error.

Logical operators yield 0 or 1 as a result.
The results of logical operators are integers.

2.2.2 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 *SimplIQ* drive language. The table also specifies operator precedence, ordered from highest to lowest precedence level.

Operator	Description	Precedence
~	Bitwise NOT of an operand	17
!	Logical negation	17
-	Unary 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 between two operands	11
	Bitwise OR between 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 2-1: **Mathematical and Logical Operators**

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

```
A = 3;  
B = 2;  
C = A/B/2;  
C = 0.75  
C = A/(B/2)  
C = 3
```

2.2.3 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:

```
1+2           (The result is 3, integer.)  
1+0x10       (The result is 17, integer. Note that 0x10 is treated as a standard  
              integer.)  
1+2.0       (The result is 3.0, float.)  
2.1+3.4     (The result is 5.5, float.)
```

If the result of add and subtract operations between two integers exceeds the integer range $[-2,147,483,648 \dots 2,147,483,647]$, the result is truncated and the type remains an integer. For example:

The result of $2,147,483,647 + 10$ is $-2,147,483,639$

A division operation between two integers may yield a floating-point result if the result includes a remainder. For example:

```
8/2           (The result is 4, integer.)  
9/2.0       (The result is 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. For example:

```
100,000 * 100,000 (The result is 1.0e+10, float.)
```

Bit operators require an integer input. Floating-point inputs to bit 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 applying the operator $\&$ (bitwise AND).



The result of a unary minus operation for the minimum integer value exceeds the integer range; therefore, the result is truncated to the maximum integer value: $-0x80,000,000$ results in $2,147,483,647$ or $0x7FFFFFFF$.

2.2.4 Operator Details

The following table describes the operators in detail.

Operator / Description	Notation	No. of Arguments	Output Type	Examples
Arithmetic addition	+	2	See operator rules	4+5=9 3.45+2.78=6.23
Arithmetic subtraction	-	2	See operator rules	4-5=-1 3.45-2.78=0.67
Arithmetic multiplication	*	2	See operator rules	PA=PA*2 doubles PA 5*4=20 1.5*2=3.0
Arithmetic division	/	2	See operator rules	20/4=5 3/1.5=2.0
Remainder after division of two integers	%	2	32-bit long integer	20%4=0 5%2=1
Bitwise NOT	~	1	32-bit long integer	~3 is 0xfffffc, 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 as x==y yields 1 If x=3 and y=5 as x==y yields 0
Logical inequality	!=	2	0 (false) or 1 (true)	If x=3 and y=3 as x!=y yields 0 If x=3 and y=5 as x!=y yields 1
Logical greater than	>	2	0 (false) or 1 (true)	If x=3 and y=3 as x>y yields 0 If x=3 and y=2 as x>y yields 1 If x=1 and y=2 as x>y yields 0
Logical greater than or equal to	>=	2	0 (false) or 1 (true)	If x=3 and y=3 as x>=y yields 1 If x=3 and y=2 as x>=y yields 1 If x=1 and y=2 as x>=y yields 0
Logical less than	<	2	0 (false) or 1 (true)	If x=3 and y=3 as x<y yields 0 If x=3 and y=2 as x<y yields 0 If x=1 and y=2 as x<y yields 1
Logical less than or equal to	<=	2	0 (false) or 1 (true)	If x=3 and y=3 as x<=y yields 1 If x=3 and y=2 as x<=y yields 0 If x=1 and y=2 as x<=y yields 1
Logical AND: Result is 1 if both arguments are non-zero, 0 if any is zero *	&&	2	0 or 1	1 && 5 yields 1 0.21 && 2 yields 1 0 && 2 yields 0

Operator / Description	Notation	No. of Arguments	Output Type	Examples
Logical OR: Result is 1 if any argument is nonzero, 0 if both are zero *		2	0 or 1	1 0 yields 1 0 0 yields 0
Logical NOT: Result is 1 if argument is zero; otherwise it is 0*	!	1	0 or 1	!4 yields 0 !0 yields 1 !0.0004 yields 1
Unary minus: Result is negative if 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 1st operand left by number of positions the 2nd operand specifies*	<<	2	32-bit long integer	8<<2 yields 32
Bitwise right shift: Shifts 1st operand right by number of positions the 2nd operand specifies*	>>	2	32-bit long integer	8>>2 yields 2

* The arguments are truncated to integers before evaluation.

Table 2-2: **Operator Details**

2.2.5 Mathematical Functions

The following table lists the built-in mathematical functions of the *SimplIQ* Interpreter language. Function names are case sensitive.

Operator	Description	Returns
sin	Sine	Floating point
cos	Cosine	Floating point
abs	Absolute value Note: The absolute value of an input argument of hexadecimal 0x80,000,000 exceeds the long value range and will therefore be limited to the maximum long value for positive numbers.	Same type as input argument
sqrt	Square root, or zero if argument is negative	Floating point

Operator	Description	Returns
<code>fix</code>	Truncate to integer: <code>fix(3.8)</code> is 3 <code>fix (-3.8)</code> is -3 Note: If an 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
<code>rnd</code>	Truncate to nearest integer: <code>rnd(3.8)</code> is 4 <code>rnd(-3.8)</code> is -4 <code>rnd(3.4)</code> is 3 Note: If an 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
<code>sign</code>	Returns the sign of the input argument: -1 for negative numbers, 1 for positive numbers and zero for a zero. <code>sign (-3.8)</code> is -1 <code>sign (3.8)</code> is 1	Integer
<code>real</code>	Convert integer to float. If the argument is a floating point number, the function does nothing: <code>5/2</code> is 2 <code>real (5)/2</code> is 2.5 <code>5/real (2)</code> is 2.5	Floating point

Table 2-3: **Mathematical Functions**

2.2.6 Expressions

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

2.2.6.1 Simple Expressions

A simple expression is evaluated in 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 [Table 2-1](#). In the case of equal priorities, the expression is evaluated from left to right. The use of parentheses is allowed up to 16 nesting levels.

Examples:

Command Line	Results	Remarks
SP*2/5+AC	101,000	The order is ((SP*2)/5) + AC
IP 5		OR operation on IP
2+3	5	
1,400,000	1,400,000	

2.2.6.2 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>

Examples:

SP=SP*2/5+AC

OP=IP|5

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

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

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

Example:

CA[1]=1

Note 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, the floating 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.

Example:

Expression Sent	Response Received	Remarks
AC=12,345.6789	-	Assign floating value to integer AC command.
AC	12,346	Floating value rounded to nearest integer.
KV[10]=215.789e8	-	Assign floating value to integer KV[10] command.
KV[10]	2,147,483,647	Floating value truncated to 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	-	Assign integer value to floating point command TC.
TC	1.0	Assigned integer value is converted to float.
temp=12,345,678	-	Assign integer value to floating point variable.
temp	1.234568e+7	Assigned value is truncated to 12,345,680.0.

2.2.6.3 User Variables

User variables are defined within a user program. The description and syntax rules of the variable definition are outlined in [Chapter 3](#).

User program variables may be used within the command line only if a 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.

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 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 ZEBRA,GIRAFFE[3];
float GNU;
```

The expression `GNU=ZEBRA*GIRAFFE[1]+2*sin(GIRAFFE[2]);` is valid. User program variables are case sensitive.

2.2.6.4 Built-in Function Calls

The built-in function call may be used in a single expression. For a list of mathematical built-in functions, refer to [Table 2-3](#). Non-mathematical built-in functions are as follows:

Operator	Description	Returns
tdif	Time difference x=TM tdif(x) returns the time in msec since x=TM has been sampled.	Integer
emit(n)	Emits the n TPDO (CAN transmit process data object), where n equals 1, 3 or 4. For details see section 3.4.7 .	Integer, 1 if function is completed successfully; otherwise 0

Operator	Description	Returns
emcy(n)	Issues an emergency message from a user program, where n is the error code (32-bit long integer) defined by a user.	Nothing
PrgErr(N)	Returns the last program error of machine N, where N is a value between -1 and 2. This operator can be used in an AUTO_PERR routine to query about the recent failure. If N = -1, the <i>SimplIQ</i> drive returns the last error code of the machine from which this function was called. Note that it makes no sense to call PrgErr(-1) from the interpreter.	Integer
tick	<p>Reads the system time in internal units. This function uses an internal timer unrelated to the system microsecond counter (refer to the TM command section in the <i>SimplIQ for Steppers Command Reference Manual</i>). 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.</p> <p>This function is used to implement the wait statement (3.7.5). The tick(x) argument is the time difference used in the wait statement. The valid range of the argument is [0...32,000] milliseconds, a limitation based on the implementation of the tock function (following). An argument greater than 32,000 will abort the tick function with an OUT_OF_RANGE error code. To read the system time in internal units, set the argument to 0.</p>	Integer
tock	<p>Returns the time difference.</p> <p>If <code>internal = tick(x)</code>, <code>tock (internal)</code> will return the time in milliseconds once <code>internal = tick(x)</code> has been sampled. The tock function operates in a manner similar to <code>tdif</code>, but it uses an internal timer, unrelated to the system microsecond counter.</p> <p>Note: For a time difference greater than 32 seconds, the function <code>tock(internal)</code> may return an erroneous result.</p>	Integer

The 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)
```

2.2.6.5 Time Functions

The TM command is used to read the system 32-bit microsecond counter. The time difference from the present time to an older sampling of TM can be determined using two methods, as in the following examples:

- `QP[1] = TM ; ** QP[1] is used just as storage`
`... ** Do something`
`QP[2] = TM -QP[1]; ** QP[2] is time difference in microseconds`
- `QP[1] = TM ; ** QP[1] is used just as storage`
`... ** Do something`
`QP[2] = tdif(QP[1]) ; ** QP[2] is time difference in milliseconds`

Time differences can be no longer than 31 minutes. To pause for a given time in a user program, use the Wait function ([section 3.7.5](#)).



In a CAN network, the time counter can be changed by the CAN network master, in which case, the `tdif(x)` function may return a value different than the time elapsed since `X=TM` was sampled.

To prevent an external event from affecting the timer, use the tick function. The time difference between the present time and an older 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] = QP[1]-tick(0); **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`



The `tick()` and `tock()` functions cannot measure time differences greater than 32 milliseconds.

2.2.6.6 User Function Calls

The XQ command enables a user function call (see [Chapter 4](#)). A user function cannot be called from the command line without the XQ command.

2.2.7 Comments

Comments are texts written into the code to enhance its readability. A comment starts with a 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.

Chapter 3: The SimplIQ User Programming Language

SimplIQ servo drives read a user program in Elmo High-level language (EHL) after it has been translated by the compiler into a sequence of virtual assembly commands (described in [Chapter 4](#)). The Compiler, part of the Elmo Studio IDE, is integrated into the Composer. The compilation process can run off line inside the PC. It is not part of the *SimplIQ* firmware. Before the *SimplIQ* drive executes a user program, the program must first be compiled, and the compiled code must then be downloaded to the serial flash memory of the *SimplIQ* drive. By compiling code prior to downloading, text analysis can be performed offline, saving online time and boosting user program performance. Another advantage is that user syntax improvements can be made without upgrading the drive software.

A drive program is a list of commands in a certain order. A user program can be anything from a simple list of commands to a very complicated machine management algorithm. The compiled code stored in the *SimplIQ* drive is a list of commands in a certain order.

This chapter describes how to write, maintain and run user programs for the *SimplIQ* drive.

3.1 User Program Organization

A user program is organized as follows:

- Integer and floating point variable declarations.
- Program text, including expressions, commands, labels and comments.
- An exit directive, which may be used to terminate the program.

Most Interpreter commands can be used in the program text. This feature is given for each command in the "Source" attribute of the command in the *SimplIQ for Steppers Command Reference Manual*. Interpreter commands that cannot be used in a program are those that:

- Upload or download data between the drive and its host.
For example, VR cannot be used for version identification (upload process).
- Store data in the flash memory or retrieve data from the flash memory .
For example, CD cannot be used to reload parameters from the flash memory.
For example, XC## resumes a halted user program.
- Are involved in executing the user program.

In addition to the Interpreter commands, a program may include program flow statements that manage how the program runs:

- Iterations, such as: `for I=1:10:100`
- Subroutine execution commands, such as: `If (I>=100)`
- Conditions, such as: `while(I<1000)`

In the program text, semicolons, commands, line feeds or carriage returns separate the commands, as in the following examples:

<code>int x,k;</code>	Variable declarations
<code>##Func</code>	Label definition
<code>X=0;</code>	Initialize
<code>for K=0:10</code>	Iterate
<code>X = x + 1 ;</code>	Do something
<code>End</code>	End of iteration
<code>Exit</code>	End of program directive
<code>##Lab</code>	
<code>...</code>	More code

The program defines two variables named `x` and `k`. `##Func` is an entry point. After compilation, this piece of code can be run by sending the command `XQ##Func` to the Interpreter.

When the program starts at `##Func`, it clears the user variable `x`. This is not performed automatically and an initial value must be set manually for every relevant variable. The program then iterates 11 times, incrementing `x` with every iteration. Finally, `x = 11`.

The `Exit` command terminates program execution. Another code section can then be executed by sending the command `XQ##Lab`.

Example:

<code>switch (IP & 3)</code>	Select according to the two low input bits.
<code>case 1</code>	The numerical interpretation of <code>(IP&3)</code> is 1.
<code>PR=1000;</code>	
<code>case 2</code>	The numerical interpretation of <code>(IP&3)</code> is 2.
<code>PR=500;</code>	If value is two . . .
<code>otherwise</code>	
<code>PR=100;</code>	Otherwise . . . (last two bits are 0 or 3).
<code>end</code>	
<code>BG</code>	Begin motion.

This example moves an axis with a step that depends on the state of the digital inputs.



Program variables that are declared out of the function scope are considered global. Global variables can be used — for monitoring and modifications — from an external terminal such as the Composer Smart Terminal. Moreover, the evolution of global variables can be recorded by the recorder in the Composer IDE.

3.2 Single and Multiple Command Execution

A single line in a *SimplIQ* program is executed as a single unit, preventing intervention by the Interpreter or by a CAN command. For example, in the sequence:

```
UM=5;
```

```
MO=1;
```

an Interpreter command could be executed between the execution of the two program statements. If, for example, the Interpreter statement between these two lines is `UM=2`, `MO=1` would be specified for a wrong unit mode. The sequence:

```
MO=0; UM=5; MO=1; BG
```

guarantees that `MO=1` is executed with the correct unit mode, because no other command can intrude.

The policy of executing a full line ensures that commands are executed in a guaranteed sequence and enables the user to regulate the speed of program execution. The more commands in a single program line, the faster program execution will be, at the expense of a slower response to host communications.

Note, however, that a drawback to this policy is that if the execution of a single program line takes a long time, or if it loops internally forever, the *SimplIQ* drive may become totally unresponsive to its CAN and RS-232 communication. In order to reduce this hazard, the execution of a single program line is time-out protected by a limitation of 3 seconds as the maximum time that program line execution can last. If a program line takes more than 3 seconds to be executed, the *SimplIQ* drive stops it with error code 96: User Program Time Out.

3.3 Standard Conventions

A user program contains lines of text code, which must use defined syntax in order for the Compiler to recognize it. This section describes common elements of program text.

3.3.1 Line and Expression Termination

A line can have the following terminators: carriage return, line feed or a combination. A line may contain a single expression or a sequence of expressions. Expressions in a sequence on the same line can be separated with a semicolon or comma (not inside parentheses or brackets).

Examples:

<code>a = 3 , b = 2 , c = a + b ,</code>	One line of three expressions separated by commas
<code>a = 3 ; b = 2 ; c = a + b ;</code>	One line of three expressions separated by semicolons
<code>a = 3 , b = 2 ; c = a + b</code>	One line of three expressions separated by a comma, a semicolon and terminated with a line feed
<code>[a,b] = func (23, c, 3.14)</code>	An expression in which the comma is not an expression separator because it is inside parentheses

3.3.2 Line Continuation

A user program may contain a line that is too long, and whose representation on the screen is not easily readable because not all its symbols are shown on the screen. In order to improve program readability, the expression can be continued on the next line by using an ellipsis (three periods) to indicate that the line continues.

Example:

```
c = 12 * a + sqrt(2) - sin(3.14 / 2) + 7 ^ 3 * ...  
(6 + b) * 34
```

The ellipsis (...) at the end of the first line indicates that this expression is not complete and is continued on the next line.

3.3.3 Limitations

Every line of user program text may contain a maximum of 128 characters (for proper on-screen readability). If a text line exceeds this value, the Compiler issues an error.

Expressions also have limitations: the maximum admissible length of an expression is 512 symbols, not including comments and ellipses. If a program contains a complex expression that takes multiple lines, and the summary length of the expression (without comments and ellipses) exceeds 512 characters, the Compiler issues an error.

User program text is also limited according to the specific *SimplIQ* drive. The list of setup parameters that limit a user program are:

- Maximum length of user program text
- Maximum number of routines, including functions, labels and auto-routines
- Maximum number of variables, both global and local
- Maximum length of data segment – space for storing global variables
- Maximum length of code segment – space for compiled code
- Maximum depth of stack – working space for the program.

The Elmo Studio IDE enables a user to view these parameters.

3.4 Expressions and Operators

3.4.1 Numbers

The number syntax in the user program language is similar to that of the Interpreter language ([section 2.2.1](#)), although ranges and range-exceeding behavior differ. A user program can be compiled on the PC in off-line mode (without communication), so that it has more resources than the Interpreter does.

The range for floating-point numbers is [-1e38...+1e38], which is greater than that of the Interpreter language. If an integer number exceeds the integer range, it is interpreted as a floating-point number, while the Interpreter interprets it as an error. If a floating-point number exceeds the floating-point range, it is interpreted as an error.

3.4.2 Mathematical and Logical Operators

The description and syntax is the same as for the Interpreter language (refer to [section 2.2.2](#)).

3.4.3 General Operator Rules

The description and syntax is the same as for the Interpreter language (refer to [section 2.2.3](#)).

3.4.4 Operator Details

The description and syntax is the same as for the Interpreter language (refer to [section 2.2.4](#)).

3.4.5 Mathematical Functions

The description and syntax is the same as for the Interpreter language (refer to [section 2.2.5](#)).

3.4.6 Exclusive OR Operation

The exclusive OR is a function that can be operated only from a user program; it returns an XR value out of two arguments, which are interpreted as integers.

Example:

`XOR(170, 75)` is interpreted as follows:

The binary representation of 170 (a int_value) is 0000 0000 1010 1010. The binary representation of 75 (b int_value) is 0000 0000 0100 1011. Performing the bitwise exclusive OR operation on these two values gives the binary result 000 000 1110 0001, which is decimal 225:

```

XOR(a,b)
    0000 0000 1010
1010
    0000 0000 0100
1011
-----
0000 0000 1110 0001

```

The exclusive OR between the binary bit representation of two integers returns 0 if both integers are identical; otherwise, it returns 1.

Syntax:

`z = XOR(x,y)`

where *z* is the result of an exclusive OR operation between *x* and *y*.



Notes:

- This operation is valid only in the user program; it is not valid through the Interpreter.
- If any operand is floating point, it is first converted to integer.

3.4.7 CAN Object Emission

3.4.7.1 The Emit Function

The emit(n) function emits the n TPDO (Can Transmit Process Data Object) subject to the following restrictions:

- The drive supports CAN communication.
- The CAN communication state is operational.
- n equals 1, 3 or 4.
- The corresponding TPDO has its transmission type set to 254.
- The corresponding TPDO is mapped to valid signals.

The emit(n) function returns 1 if successful and 0 if it does not succeed in emitting a TPDO. It returns immediately after the TPDO is placed in the CAN transmitter queue. The actual emission of the TPDO may be delayed.

3.4.7.2 The EMCY Function

The EMCY function issues an emergency message from a user program. It is valid only for drives that support CAN controllers.

Syntax:

EMCY(x)

where x is an error code (integer 4 bytes in length) defined by the user.

The EMCY() function always returns 0.



When the EMCY() function is called, the emergency message is transmitted only if bit 6 of object 0x2F20 (request by user program EMCY function) is set to 1. Refer to the object 0x2F20 section in the Elmo *CANopen Implementation Manual*.

When the program reaches a location at which an emergency is specified, the CAN emergency object (EMCY) is sent. The emergency object data is according to CiA DS301, with its 8 bytes of data as follows:

Byte	Contents	Remarks
0	Emergency error code	0xFF01 - device specific
1		
2	Error register (object 0x1001)	0x80 - manufacturer specific
3	Used only for PT/PVT motion, manufacturer error codes	0
4	Manufacturer-specific error field	Error code defined by user as argument of EMCY function
5		
6		
7		

Example:

The following code sends an EMCY from a routine in two cases: timeout (error code 1) or out-of-range (error code 2):

```
...
if (isTimeOut) // Check if timeout occurred
    EMCY(1); // Issue EMCY with error code 1
end
...
if (isOutOfRange) // Check if out of range
    EMCY(2); // Issue EMCY with error code 2
end
```

If a timeout occurs, the *SimpliIQ* drive sends an EMCY with the following data:

Byte	Contents	Remarks
0	Emergency error code	01
1		FF
2	Error register - 0x80 (manufacturer specific)	80
3	Not used	0
4	Manufacturer-specific error field	01
5	(1 for timeout)	0
6		0
7		0

If an out-of-range occurs, the *SimpliIQ* drive sends an EMCY with the following data:

Byte	Contents	Remarks
0	Emergency error code	01
1		FF
2	Error register - 0x80 (manufacturer specific)	80
3	Not used	0
4	Manufacturer-specific error field	02
5	(2 for out-of-range)	0
6		0
7		0

Note that the user error code and the emergency error code are formatted in the messages starting with the least significant bit.

3.4.8 Expressions

An expression is a combination of operands (parameters) and operators that is evaluated as a single value. Expressions are used with immediate numbers, drive commands, and drive and global user-program variables. This section describes the different types of expressions.

3.4.8.1 Simple Expressions

The description and syntax of simple expressions is the same as for the Interpreter language.

3.4.8.2 Constant Expressions

A constant expression is a simple expression that contains only operations with immediate numbers as operands. The Compiler recognizes the constant expression, evaluates it and uses the final result to generate the executable code of the *SimplIQ* drive. Note that the result of evaluating a constant expression by the Compiler may differ from the result calculated inside the *SimplIQ* drive, because the Compiler uses more resources and may evaluate and operate with double-precision floating-point numbers.

3.4.8.3 Assignment Expressions

The description and syntax is the same as for the Interpreter language. User programming language allows multiple assignments to multiple outputs of the function.

3.4.8.4 User Variables

User variables are defined within the program, using the “int” or “float” declaration, with the following syntax:

```
int int_var1, int_var2[12], ..., int_varN;  
float flt_var1[13], flt_var2, flt_varN;  
int a;  
float b;
```

A variable must first be declared before it is used (in the expression or assignment). The variable definition line consists of type name (int or float) and variable names. Variables on the definition line must be separated by commas; alternatively, each variable may be declared on a separate line. Variables may be scalar (for example, `int var1`, `float temp`) or one-dimensional arrays (for example, `int arr[10]`, `float ftemp[4]`). If a variable is a vector, it must be declared with its dimension in brackets after its name. 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. A dimension of less than 1 is illegal.

Examples:

```
int arr[12.5]; The floating-point number defining dimension will be truncated to 12.  
int arr[-2]; This variable definition is illegal because the dimension is negative.
```

Only global variables may be one-dimensional arrays. Neither a local variable nor an input/output argument of a function may be a vector.

Local variables must be defined at the beginning of function bodies. Any local or global variable definition after executable code of the function is illegal.

Example:

```
function func (int a)  Function definition
int b ;               Local variable definition
b = a ;               Executable code
float c :              Local variable definition
return
```

The definition of the variable `c` is illegal because it comes after executable code.



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 12 characters. A variable name cannot be a keyword.

The following is the list of keywords:

int	otherwise	wait
float	break	until
if	end	goto
else	return	nargin
elseif	function	nargout
for	global	XOR
while	exit	quit
switch	virtual	DummyLabel
case	reset	try
clear	catch	this
leave	cd	pw
save	continue	

All keywords are case sensitive.

Variable names must be distinct from function or label names.

After a program is compiled, all program variables may be used within the command line. For example, assume that a compiled program includes the following lines:

```
int ZEBRA,GIRAFFE[3];
float GNU;
```

The expression `GNU=ZEBRA*GIRAFFE[1]+2*sin(GIRAFFE[2]);` is valid.

More information about global and local variables is given in the section on [Global variables](#).

3.4.8.5 System Commands

The *SimplIQ* system commands (described fully in the *SimplIQ for Steppers Command Reference Manual*) consist of a two-letter mnemonic notation (only English letters, not case sensitive). For example, the expressions `ac = 100,000` and `AC = 100,000` have the same meaning, although the notation is different.

Each command has a 16-bit flag, each bit defining any feature. For example, the fourth bit (PostProcess flag) defines whether the command can be used to set a value or not.

Examples:

- `a = AC` This expression assigns a value of the system command AC to the variable a. It is valid if the AC command is allowed to “get a value”; that is, it has a PreProcess flag.
- `AC = a` This expression assigns a value of the variable a to the system command AC. It is valid if the AC command is allowed to “set a value”; that is, it has a PostProcess flag.
- `BG` This is an executable command. It cannot be assigned; that is, it has no PreProcess, PostProcess or Assign flag. For example, the expressions `BG = 1` and `a = BG` are illegal.
- `LS` This expression is illegal because it uses a command that has a NotProgram flag. Such commands are not allowed in a user program, although they are available for use in the Interpreter language.

3.4.8.6 Built-in Function Calls

The description and syntax is the same as for the Interpreter language.

3.4.8.7 User Function Calls

Functions perform a pre-written and compiled code. They get a list of arguments from their caller and return a list of return values. Functions get their list of input arguments by value; they do not modify them directly. The general syntax of a function call is:
`[OUT1,OUT2,...OUTN]=FUNC(IN1,IN2,...IN_N)`

If only one value is returned, brackets are not required and the syntax is as follows:
`OUT=FUNC(IN_1,...,IN_N)`

Not all output variables must be assigned. For example, if the function FUNC returns two values, then

```
[ IA[1] , IA[2] ]=FUNC()
```

assigns the two returned values to `IA[1]` and `IA[2]`.

`IA[1]=FUNC()` returns the first return value to `IA[1]` and the other returned values remain unused. If the returned value type differs from the returned variable type, the result is a type cast to the type of the variable.

The number of input arguments during a function call is strictly according to its definition.

If a function call is part of an expression, then multiple output of the function is illegal. For example:

```
[ a, b ] = 5 + func(c)
```

and

```
[ a, b ] = func(c) + 5
```

are both illegal because a function call is part of an expression and must be evaluated in a single value.

3.5 Comments

Comments are texts that are written into the code to enhance its readability. They can be written in three ways, as indicated in the following sections.

3.5.1 Double Asterisk

A comment starts with a double asterisk (**) marker and terminates at the next end of line. The drive ignores the comments when running a program or evaluating an expression.

Example:

```
**my first program
PX=1
**UM=5
MO=1; **motor on
```

In this program, the first line is a comment used to enhance program readability. The comment terminates at the next end of line, so that the `PX=1` instruction will be compiled and executed. In the third line, the comment marker tells the drive to ignore the `UM=5` command. This technique is useful for temporarily masking program lines in the process of debugging. The last line demonstrates that a comment may start anywhere in the program line. The `MO=1` instruction preceding the comment market will be compiled and executed.

3.5.2 Double Slash

A double slash marks comments in the same way as a double asterisk. The comment starts with a double slash marker and terminates at the next end of line.

Example:

```
//my first program
PX=1
//UM=5
MO=1; //motor on
```

In this example, the double slash acts exactly like the double asterisk from the previous example.

3.5.3 C-style Start and End Comment

In the C-style method of marking comments, the comment begins with a start comment mark (`/*`) and terminates with the end comment mark (`*/`). This enables the user to close text in the middle of an expression, or to close several text lines.

Example:

```
/*
This is a multiple line comment.
All this text is ignored.
*/
if ( 1 /* x == 1 */)
    y=1;
end
```

The expression `y = 1` will always be executed. The `x==1` condition enclosed by the comment markers is ignored.

3.6 Fault Handling

3.6.1 Unexpected Fault

In order to receive more information about the reasons for run-time errors, the Elmo Studio IDE must be used. This enables the *SimplIQ* drive to inform the virtual assembly that a failure has occurred, and to then have the Elmo Studio interpret this to a user program command and error reason.

The [AUTO_PERR](#) routine enables the user to program failure reactions, which may include, for example, manipulating certain outputs, stopping the motor, or emitting a CAN emergency. The `AUTO_PERR` can determine locate the last error by using the `PrgErr(N)` function.

If, after taking emergency steps, the `AUTO_PERR` is to resume the user program, the following cautions should be taken:

- A **return** command will resume execution at the next user program instruction. In most cases, after the unexpected failure of a previous instruction, this should be avoided.
- Because the depth of the program stack is unknown at the time of failure, jumping to an absolute address should be carried out only after the **reset** command has reset the call stack.
- The `AUTO_PERR` routine automatically blocks all other auto-routines; therefore, the `MI` command should be used to restore reaction to them.
- The **exit** keyword can be used to exit the user program.

3.6.2 Expected Fault

Certain places in the user program are prone to faults; for example, a user program may fail to start a motor due to an externally-activated switch. To react to an unexpected fault, a [try-catch block](#) should be used.

3.7 Program Flow Commands

The *SimplIQ* drive uses a set of commands to manage the flow of the user program. With these commands, the user program can make decisions iterate or respond automatically to certain events. The program flow commands enable user programs to perform much more complicated functions than just running a set of commands sequentially.

The program flow commands are:

while - end	Iterate as long as condition is satisfied.
until	Iterate (suspend program execution) until condition is satisfied.
wait	Iterate (suspend program execution) until a specified time elapses.
for - end	Iterate for a number of counted times.
break	Break an iteration or a switch expression (for, while, switch)
if - elseif - else - end	Conditional expression.
switch-case-otherwise-end	Case selection.
goto	Go to a certain point in the program.
reset	Kill the state of the executing program and jump to a certain point in the program.
function-return	Declare a function and its return point.
##	Declare a label or an auto-routine.
#@	Declare a label or an auto-routine.
#@ - return	Declare an auto-routine and its return point.
exit	Terminate program execution.
continue	Transfers control to next iteration of smallest enclosing for or while loop in which it appears.
try-catch	Used to react to an expected fault.

3.7.1 Labels (Entry Points) and Subroutines

Labels denote that program execution can start from that location, or that program execution can be branched to that location.

Label definition has the following syntax:

```
##<LABEL_NAME>
```

or

```
#@<LABEL_NAME>
```

A maximum of 12 characters – letters and/or digits (not leading) and underscore (not leading) only – may be used for a label. The label name must be unique.

Labels can reside inside or outside function bodies, but not within a program flow structure, such as a **for** iteration. Labels inside function bodies are regarded as local to the function and serve as targets for **goto** instructions with the same function. Labels in the global text scope serve as possible execution starting points, and also as targets for **reset** and global scope **goto**.

A subroutine is a global label in which the body ends with a **return** keyword. A subroutine is similar to a function without input and output arguments. It can be called from any place in the program and may serve as a target for a **reset** instruction.

An auto-routine is a subroutine whose **return** keyword is an instruction to return to the next line in the code, from where the subroutine was called.



DummyLabel is a keyword used internally by the Compiler. It is not a legal label or subroutine name.

The XQ and the XC program launch commands use labels to specify where to start program execution and where to terminate. For example, the command XQ ##LOOP2 is used to begin execution at ##LOOP2.

Example 1:

```
##START;           Start program.
##LOOP1;          Label
...              Body code A
Goto##LOOP1;
##LOOP2;
...              Body code B
##LOOP3;
...
```

According to this example, if the program runs from label ##START, body code A will be performed forever. ##LOOP2 will never be reached.

Example 2:

```
...
#@WAIT_POWER:    Subroutine definition
wait 3000        subroutine body - wait 3 sec
return          End subroutine
...
WAIT_POWER      Call subroutine
...
reset WAIT_POWER; Kill stack and call subroutine
```

In this example, the subroutine WAIT_POWER is called explicitly the first time and is used as the target for a **reset** instruction the second time.

3.7.2 For Iteration

This performs an indexed iteration in a program.

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.



Notes:

- If the iteration step is zero, the program is aborted with the error code INFINITE_LOOP.
- 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.

Example:

```
... Start user program or function.
float ra[20]; Float array declaration.
int ia[20]; Integer array declaration.
int k; Variable declaration.
...
for k=1:10 Start iteration loop.
ia[k]=100; Update first 10 elements of integer array.
ra[k]=55.55; Update first 10 elements of real array.
...
end End of iteration loop.
...
```

3.7.3 While Iteration

Syntax:

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

The **while** keyword executes a statement repeatedly until *expression* becomes 0. The expression can be logical and/or numerical. It may be within parentheses or without parentheses.

Example:

```
OB[1]=0           Digital output 1 is OFF.
while(IB[1])
OB[1]=1           While digital input 1 is ON, digital output 1 is ON.
end
OB[1]=0;          Digital output 1 is OFF.
...
...
While (IB[2])
end
MO=1;             This command will be performed only after digital output 2 is OFF.
```

3.7.4 Until Iteration

Syntax:

```
until (expression) ;
```

The **until** keyword suspends execution of the program until *expression* becomes true (non-zero). The expression can be logical and/or numerical.

Example:

```
...
until ((PX==20,000)&IB[1]); Suspend the program until the variable PX exceeds
                           20,000 and digital input 1 is ON.
...

```

The **until** expression can be useful for synchronizing threads for drives that support multi-thread programs. For example, if there are two threads and the second thread must start after the first thread finishes a certain procedure, a global variable should be defined to indicate whether the first thread has finished or not. The second thread is suspended by the **until** expression, as follows:

```
int IsFirstFinished ;           Global variable definition. Variable is initially set to zero.
```

The code of the first thread could be:

```
...
... Do some work...
IsFirstFinished = 1 ;           Signal that some work is complete.
```

The code of the second thread will include:

```
...                               Prior to suspension code
Until (IsFirstFinished) ; Second thread suspended until ** signal
...                               Continue program.
```

To suspend the thread without terminating it, **until** can be used with a false expression.

3.7.5 Wait Iteration

Syntax:

```
wait (expression) ;
```

The **wait** keyword suspends execution of the program until the specified time elapses. The expression can be within parentheses or without them. The expression specifies the waiting time in milliseconds. It can be a numerical expression only, and is evaluated in a single value.

Example:

```
PA=10,000
BG
until (MS == 0)
wait (20) ;           Wait 20 milliseconds
```

This sequence initiates a motion using the `until (MS == 0)` to wait until the motor is stabilized at a new position. The `wait (20)` allows an additional 20 milliseconds for final stabilization.



Notes:

- The **wait** argument range is [0...32,000] milliseconds. This limitation stems from the implementation of the tick and tock system functions, used in the **wait** statement algorithm. If an expression exceeds the valid range, the *SimplIQ* drive returns an OUT_OF_RANGE error code.
- The **wait** expression is evaluated only once, at the beginning of iterations, and is not recalculated.

3.7.6 If Condition

Syntax:

```
if ( expression1)
...
statement1
...
elseif ( expression2 )
...
statement2
...
else
...
statement3
...
end
```

The **if** keyword executes `statement1` if `expression1` is true (non-zero); if **ifelse** is present and `expression2` is true (non-zero), it executes `statement2`. The **ifelse** keyword may repeat scores of times during the **if** condition. `statement3` will be executed only if `expression1`, `expression2`, ... `expressionN` are all false (or zero).

Example:

```
if (IB[4])
PR=1000;           PR=1000 only if digital input 4 is ON.
elseif(IB[3])
PR=5000;          PR=5000 only if digital input 3 is ON.
elseif(IB[2])
PR=3000;          PR=3000 only if digital input 2 is ON.
else
PR=500;           PR=500 only if digital inputs 2, 3 and 4 are OFF.
end
```

3.7.7 Switch Selection

Syntax:

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

The **switch** statement causes an unconditional jump to one of the statements that is the “switch body,” or to the last statement, depending on the value of the controlling expression, the values of the **case** labels 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. Labeled statements are not syntactic requirements either, 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 in the **case** label is compared for equality with the **switch** expression. If the **switch** expression and the **case** expression are equal, then this **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** is performed automatically, and the next **case** statement is not executed.

If a number of **case** expressions match the switch expression, then the first matching **case** is selected.

Example:

The following example selects the size of a point-to-point motion according to the value of variable k.

ink k	Variable declaration
...	
switch (k)	For example, k=2
case 1	
PA=1000;	
case2	
PA=2000;	This statement will be performed.
otherwise	
PA=500;	If k does not equal 1 or 2, PA=500.
end	

3.7.8 Continue

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

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

Example 1:

```

...
for k=1:5
    if arr[k] == 0
        continue
    end
end
...
end

```

In this example, the **continue** keyword is within an **if** block. If the condition `arr[k] == 0` is true, it jumps to the beginning of the **for** loop.

Example 2:

```

while 1
...
    try
...
        for k=0:5
            if IB[16+k] == 1
                break
            end
            if MS != 0
                continue
            end
            MO=1;PA=0;BG
        end
    catch
...
    end
end

```

If the condition `MS != 0` is true, the program jumps to the beginning of the for loop and the statements `MO=1;PA=0;BG` are not executed.

3.7.9 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 outside **for**, **switch** or **while** statements is illegal.

Example:

```

...
while (IB[2])           Loop until digital input 2 is ON.
if (!IB[1])
break;                 Break the loop when digital input 1 is OFF.
end
MO=1;                 This command will be performed only after digital input 1 or
                    2 is OFF.

```

3.7.10 Return

The **return** command terminates the current function and returns control to the invoking function. A called function normally transfers control to the function that invoked it when it reaches the end of the function. A **return** may be inserted within the called function to force an early termination and to transfer control to the invoking function. If the function was called by an execute (XQ) command, a **return** command will close that program thread. Closure of the main thread will close all other active threads.



Global variables can still be used and monitored by the user with external terminals, such as the Composer Smart Terminal.

3.7.11 Try-Catch

A try-catch block is used to react to an expected fault.

Syntax:

```
try
statement, ..., statement,
catch
statement, ..., statement
end
```

The *SimplIQ* drive stores the status (stack and base pointers) and executes the statements between the **try** and the **catch**. If successful, nothing else happens. If an error occurs, the status is restored and the *SimplIQ* drive executes the catch block. A failure in the **catch** block is treated as an unexpected failure.

The *SimplIQ* drive cannot undo statements that were already executed in the **try** block before a failure.

Limitations for using a **try-catch** block include:

- It is illegal to use **goto** and **return** statements within a **try-catch** block.
- A control block within a **try-catch** block must be closed. For example:

```
int a;
...
try
    if (a==1)
catch
...
end
...
```

This example is illegal, because the **if** control block is not closed inside the **try** block.

- A **try-catch** block within a control block must be closed. For example:

```
int a;
...
if (a==1)
    try
    ...
else
    catch
    ...
end
end
...
```

This example is illegal because the **try-catch** block is not closed within the **if** block.

3.8 Functions

Functions are program sections that can be defined by parameters and called from anywhere in the program.

3.8.1 Function Declaration

A function declaration consists of the following parts:

- Reserved **function** keyword.
- List of output arguments with their types in brackets (optional).
- Assignment sign, only if there is an output argument.
- Function name.
- List of input arguments with their types in parentheses (list of input arguments can be empty).
- Reserved **return** keyword that closed the function scope.

Example 1:

```
function [int y1, int y2] = func1 (float x1, int x2)
```

A function named `func1` has two input arguments – `x1` and `x2` – and returns two output arguments.

The list of output arguments may be partial or empty. Thus

```
[y1, y2] = func1 (x1, x2)
```

```
y1 = func1 (x1, x2)
```

and

```
func1 (x1, x2)
```

are all perfectly valid.

The maximum admissible number of input and output arguments is 16.

Example 2:

```
function func2 (float x1)
```

A function named `func2` has a single input argument of float type and returns no output.

Example 3:

```
[int y1] = func3 (int x1)
```

This function declaration is illegal because the **function** keyword is missing.

Example 4:

```
function y1 = func5
```

This function prototype is illegal because the type of output variable is missing.

The valid function name follows the same rules as the variable name. It must be distinct from any variable, function or label name. The definition of dimensions of the output and input arguments at the function declaration is illegal.

Example 5:

```
function [int x[100]] = func3 ()
```

This function declaration is illegal because the dimension of the output argument is defined.

A function may have a prototype before its declaration. The prototype has the same syntax as the function declaration, but it must end with a semicolon.

Example 6:

```
function [float y1] = func4 () ;
```

A prototype of function `func4` that has no input argument and returns only the output argument.

Example 7:

```
function float y1 = func4 ;
```

Same as example 6.

Example 8:

```
function float = func4 ;
```

Same as examples 6 and 7. The name of input or output argument of the function prototype may be omitted, but during function definition, it will be an error.

Example 9:

```
function [int, int] = func1 (float, int);
```

Prototype of function from example 1. Same as function `[int y1, int y2] = func1 (float x1, int x2) ;`

The prototype of the same function can be written several times (multiple prototype), but all prototypes must be identical. The names of the input/output arguments in the function prototype may be omitted, or may be different from the names of the corresponding argument names in the function declaration.

Example 10:

```
function [int y1, int y2] = func (float x1, int x2) ;
function [float y1, int y2] = func (float x1, int x2)
```

The first expression is the function prototype and the second is the function definition. This is illegal because the type of first output argument in the prototype does not match that of the declaration.

Example 11:

```
function [int y1, int y2] = func (float x1, int x2) ;
function [int a, int b] = func (float x1, int x2) ;
function [int y1, int y2 ] = func (float x1, int x2)
```

The first two expressions are the function prototype and the third is the function definition, which is legal.

The body of a function resides below the declaration and must end with the **return** keyword. If the **return** is inside a control block within the function body, it is not the end of the function body.

Example 12:

function [int y1, int y2] = func (float x1, int x2)	Function definition.
y1 = x1;	Function body
y2 = x2;	
if x2 > 0	If block
Return	return inside block is not end of the function
	End of if block
end	Executable code
y2 = y1 + y2 ;	
Return	Function end

Before a function call, the function must be declared, either as a function prototype or a function definition.

Example 13:

function [int y1, int y2] = func (float x1, int x2);	Function prototype
...	
function main ()	Function main definition
int a, b;	Local variable definition
[a,b] = func (2.3, -9.0);	Function call
Return	Function main end
...	
function [int y1, int y2] = func (float x1, int x2)	Function definition
...	Function body
Return	Function end

If a function is declared without a body, its call is illegal.

Example 14:

function [int y1, int y2] = func (float x1, int x2) ;	Function prototype
...	
function main ()	Function main definition
int a, b;	Local variable definition

```
[a,b] = func (2.3, -9.0);           Function call
Return                               Function main end
...
```

In this example, function `func` has a prototype, but no body, so its call is illegal.

3.8.2 Dummy Variables

The input and output arguments of a function are called dummy variables. A true variable is substituted for the dummy variable when a function is called.

Example:

```
function [float mean, float std]=statistic();
```

In this function, the variables “mean” and “std” are dummy variables (full example in the next section).

3.8.3 Count of Output Variables

A function may return multiple values, although in certain cases, not all outputs need be computed. The **nargout** keyword can therefore be used to indicate which of the results actually need to be evaluated.

The number of outputs is the number of items in the left side of the expression during the function call. If this number exceeds the maximum number of defined output arguments for the function, or the maximum admissible number of output arguments, an error will be declared.

If a function does not return any output by definition, a zero output value will be inserted to the stack. For example, if the function `func` is declared with no output arguments, the expression `func() + 3` is legal because `func` returns zero by default.

Example:

```
float vec[11], RA[100];           Declare the global variables.
float value;                       Declare the global variable.
function [float mean, float std]=statistic();  Function prototype.
...
[RS[1],RA[2]]=statistic()         Call the statistic function. After
                                  execution, RA[1] will be equal to
                                  the variable mean and RA[2] will
                                  be equal to variable std.

[value]=statistic()              In this case, after executing the
                                  statistic function, value will be
                                  equal to variable mean.
...
function [float mean, float std]=statistic();  Declare a function that
                                  calculates mean and standard
                                  deviation of the global vector
                                  vec.
```

```

int k;
global vec[11];

mean=0;
for k = 1:10
mean = mean + vec[k];
End
mean = mean/10;
if (nargout>1)

std = 0;
for k = 1:10
std = std + vec[k] * vec [k];
End
st = (1/10) * ssrt. (std - 10 * mean)
End
Return
...

```

Declare k as automatic variable.

Redeclare for vec variable (vec is global variable previously declared).

Calculate mean of vec[]

Only if standard deviation is queried . . .

End of function body

For count of input arguments, the number of input arguments during a function call must be the same as declared during the function definition.

3.8.4 Automatic Variables

A variable declared within a function is automatic; it is generated when the function is called and ceases to exist when the function exits. At the time of the function call, all of its automatic variables are set to zero. When the function exits, the value of the automatic variables is not saved.

Automatic variables cannot be vectors.

Example:

In the example of the statistic function in the previous section, the variable `k` is the automatic variable.

3.8.5 Global Variables

A function can reference a persistent variable, which, in this case, must be declared as **global** inside the function and defined above the function definition. The **global** keyword is obligatory; otherwise, the variable will be referenced as an automatic variable of the function.

The dimension of a persistent variable is defined once in a program, during its definition, and is the same for the global variable in all functions in which it is used. The legal way to declare the dimension of a persistent variable within a function is to use empty brackets after its name, or no brackets at all.

Examples:

In the example of the statistic function in the previous section, the variable [vec\[11\]](#) is the global variable. It is defined above the function definition and is redeclared as global within the function body.

<code>float temp;</code>	Declare the global vector variable.
<code>int vec[10];</code>	Declare the global vector variable.
<code>...</code>	
<code>function [float a]=func(int b)</code>	Declare a function func.
<code>float temp;</code>	Declare temp as an automatic variable, because the global keyword is absent.
<code>...</code>	Function body.
<code>return</code>	End of function.
<code>...</code>	
<code>function [float a]=func1(float temp)</code>	Declare function func1. In this case, temp is not a global variable, but rather an input argument.
<code>...</code>	
<code>return</code>	End of function.
<code>...</code>	
<code>function [float a]=func2(float b)</code>	Declare function func2.
<code>global float temp;</code>	Redeclare global variable temp. In this function, temp is the global variable.
<code>global int vec[]</code>	Redeclare global variable vec. Notice that its dimension is omitted during redeclaration and that its actual dimension is 10, as defined above.

3.8.6 Jumps

Syntax:

```
goto ##LABEL1
```

The **goto** command instructs the program to continue its execution at the label specified by the jump command. It may be specified only for destinations within the present function scope and is illegal if used in a **for** loop or **switch-case** block. Jumps to labels within other functions are not possible. Jumps to a global label from within a function are illegal.

Example:

<code>...</code>	Working code.
<code>if (PX>1000)</code>	Condition for jump.

<code>goto##LABEL1;</code>	Go to LABEL1 if condition is true.
<code>else</code>	Return axis to origin.
<code>goto##LABEL2;</code>	Go to LABEL2 if condition is false.
<code>end</code>	
<code>...</code>	Working code.
<code>##LABEL1;</code>	Declare LABEL1.
<code>...</code>	Working code.
<code>##LABEL2;</code>	Declare LABEL2.
<code>...</code>	Working code.

3.8.7 Functions and the Call Stack

A function is a piece of code that may be called from anywhere in the program. After the function is executed, the program resumes from the line just after the function call.

Example:

<code>function JustSo;</code>	Function prototype.
<code>...</code>	
<code>JV=1000</code>	
<code>JustSo()</code>	Function call.
<code>BG</code>	
<code>...</code>	
<code>function JustSo</code>	Function definition
<code>IA[1]=1;</code>	Function body.
<code>return</code>	Function end.

This code executes the sequence:

```
JV=1000;IA[1]=1;BG;
```

After executing `JV=1000`, the program jumps to the subroutine `JustSo`. Before doing so, it stores its return address, which is the place in the code at which execution should resume after the routine is complete. In this example, the return address is the line number of instruction `BG`, which is just after the subroutine call.

The **return** command instructs the program to resume from the stored return address. After execution is resumed at that address, it is no longer required and is no longer stored.

Functions may call each other; in fact they may even call themselves. Return addresses for nested function calls are stored in a call stack, as shown in the following example:

<code>function factorial();</code>	Function prototype.
<code>...</code>	
<code>IA[1]=3</code>	
<code>IA[2]=1</code>	
<code>factorial()</code>	Function call.
<code>BG</code>	

```

...
function factorial()           Function for factorial.
global int IA[];              Define array as global inside function.
IA[2]=IA[2]*IA[1]             Recursive algorithm.
IA[1]=IA[1]-1
if (IA[1]>1) factorial() ; end  Recursive call.
return                         Function end

```

The factorial function in this example calculates the factorial of 3 in IA[2]. The variable IA[1] counts how many times the function factorial is executed.

The program executes as follows:

Code	IA[1]	IA[2]	Call Stack
IA[1]=3	3	Undefined	Empty
IA[2]=1	Unchanged	1	Empty
factorial	Unchanged	Unchanged	→BG
IA[2]=IA[2]*IA[1]	Unchanged	3	Unchanged
IA[1]=IA[1]-1	2	Unchanged	Unchanged
if (IA[1]>1)	Unchanged	Unchanged	→RT
factorial(); end			→BG
IA[2]=IA[2]*IA[1]	Unchanged	6	Unchanged
IA[1]=IA[1]-1	1	Unchanged	Unchanged
if (IA[1]>1)	Unchanged	Unchanged	Unchanged (condition is false)
factorial(); end			
return	Unchanged	Unchanged	→BG (program jumps to return on top of call stack)
return	Unchanged	Unchanged	Empty (program jumps to BG on top of call stack)
BG	Unchanged	Unchanged	Unchanged

3.8.8 Killing the Call Stack

In rare situations, it may be desirable to exit a function without returning to its return address. The **reset** instruction solves this problem by emptying the call stack before making a jump.

Syntax:

```
reset<JUMP_NAME>
```

The valid jump after the reset keyword is one of the following:

- Label
- Auto-routine
- User function with no defined input arguments

All other expressions or absence of expressions after the **reset** keyword are illegal.



A label in a reset expression must be global. A local label is illegal, because the stack will be emptied and all local variables and the return address of the function to which the local label belongs will be erased.

Example:

Assume that a drive (an axis) runs a programmed routine. An inspection station may assert a “Product defective” digital signal that is coupled with digital input #1. An automatic routine is therefore coupled to digital input #1 in order to stop the part assembly and prepare for the assembly of the next part.

##START_NEW	Label for starting a new part.
...	Working code.
...	Last line of working code.
#@AUTO_I1	Subroutine label.
PA=0;BG	Return axis to origin.
reset START_NEW	Clear stack and go to beginning.

The **reset** in the #@AUTO_I1 routine is required because it is not known if any function calls are executing when digital input #1 is asserted. If a function *is* executing, the **reset** prevents junk from accumulating in the call stack. Otherwise, the call stack is empty and **reset** does no harm. Note that after the **reset**, control does not return to the function that was executing prior to the #@AUTO_I1 routine. The stack is cleared and the return address to the interrupted function is removed from the stack.



The #@ATUO_I1 routine is executed after the work code is completed for every assembled part. The program proceeds from the last line of the working code to PA=0;BG, which resets the machine for another part assembly. The next instruction is a **reset** to the START_NEW label.

3.8.9 Automatic Subroutines

3.8.9.1 List of Automatic Routines

An automatic routine (auto-routine) is a special type of routine that is executed automatically according to system events. These routines are executed only when their invocation condition is satisfied. Auto-routines have no output and input arguments.

Syntax:

An auto-routine can be defined either as a function or as a subroutine:

- If defined as a *function*, all syntax rules for function definition are relevant (see [section 3.7.1](#)).
- If defined as a *subroutine*, the auto-routine name follows the sequence of characters ## or #@ in the definition line. The body of the auto-routine follows the definition line and ends with the **return** keyword unless this **return** is inside a flow control block (see [section 3.7.1](#)).



After calling the auto-routine and performing its body, the **return** keyword instructs the program to return to the line in the code after which execution was halted by a interrupt event.

There are no default handlers for auto-routines. If a user does not define the auto-routine, no handling will be activated when an automatic event is asserted.

The following table summarizes all automatic routines, ordered from highest to lowest priority.

Routine Name	Priority	Activated by	Mask (MI)
AUTOEXEC	0	An autoexec code is executed automatically upon power on. An autoexec function can be called subsequently at any time.	1 (0x1)
AUTO_PERR	1	Called when a run-time error occurs (see section 3.6.1).	32,768 (0x8000)
AUTO_ER	2	A motor fault event, in which MO=0 is set automatically.	2 (0x2)
AUTO_STOP	3	Called when a digital input configured to the "Hard Stop" function is activated.*	4 (0x4)
AUTO_BG	4	Called when a digital input configured to the "Begin" function is activated.*	8 (0x8)
AUTO_RLS	5	Called when a digital input configured to the "RSL" function is activated.*	16 (0x10)
AUTO_FLS	6	Called when a digital input configured to the "FLS" function is activated.*	32 (0x20)
AUTO_ENA	7	Called when a digital input configured to the "Enable" function is activated.*	64 (0x40)
AUTO_I1	8	Called when a digital input #1 configured to the "GPI" (General Purpose Input) function is activated.*	128 (0x80)
AUTO_I2	9	Called when a digital input #2 configured to the "GPI" (General Purpose Input) function is activated.*	256 (0x100)
AUTO_I3	10	Called when a digital input #3 configured to the "GPI" (General Purpose Input) function is activated.*	512 (0x200)
AUTO_I4	11	Called when a digital input #4 configured to the "GPI" (General Purpose Input) function is activated.*	1024 (0x400)
AUTO_I5	12	Called when a digital input #5 configured to the "GPI" (General Purpose Input) function is activated.*	2048 (0x800)

Routine Name	Priority	Activated by	Mask (MI)
AUTO_I6	13	Called when a digital input #6 configured to the "GPI" (General Purpose Input) function is activated.*	4096 (0x1000)
AUTO_HM	14	Called when a main homing sequence is completed.**	8192 (0x2000)
AUTO_HY	15	Called when an auxiliary homing sequence is completed.***	16,384 (0x4000)

* Refer to the IL command section in the *SimpliIQ for Steppers Command Reference Manual*.

** Refer to the HM command section in the *SimpliIQ for Steppers Command Reference Manual*.

*** Refer to the HY command section in the *SimpliIQ for Steppers Command Reference Manual*.

Table 3-1: **Automatic Subroutines and Their Priorities**

All automatic routines except AUTOEXEC are activated only if a program is running.

Example:

```
##LOOP                                An endless loop.
goto##LOOP
#@AUTO_I3                              Subroutine definition.
...                                     Subroutine body.
return                                  End of subroutine.
```

In this program, the endless loop in the first two lines of the routine is intended to make the program run forever, so that the automatic routine will be able to handle the digital input #3 event.

The #@AUTO_I3 routine will be called if digital input #3 is sensed and not masked. Digital input #4 will not invoke any automatic action in the user program, because no #@AUTO_I4 routine is defined to handle a digital input #4 event. The user has the option to make an explicit call of an automatic routine, if desired. The syntax for an auto-routine call is the same as that of a user function.

3.8.9.2 Automatic Routine Arbitration

Each automatic routine has an assigned priority, according to [Table 3-1](#). When the conditions for activating two automatic subroutines occur simultaneously, the automatic subroutine with the higher priority is called. The other automatic subroutine will be marked as pending and will execute at the first opportunity in which it receives permission to execute, even if the reason for its call does not exist any more.

3.8.9.3 The Automatic Subroutine Mask

Automatic subroutines can be masked, to indicate that they are inactive. For example, it may be desirable to limit the automatic response to a certain digital input to certain situations. This is carried out using the MI command.

Example:

A drive receives a PLC command. It has an autoexec routine that activates its program upon boot. The PLC sends instructions to the drive using RS-232 communication for task parameters and a digital input to start an action immediately. The drive sets the output according to the state of the program. For safety's sake, the drive is not allowed to perform any task before digital input 2 is set.

function autoexec ()	Declare the autoexec function.
MI=128;	Inhibit the routine #@AUTO_I1.
OP=1;	Set output to indicate start.
...	
while (!IB[2])	Wait for digital input 2 to be set.
End	
goto #@TEST_PARS;	Jump to subroutine.
##LOOP	Label LOOP.
...	Label LOOP body.
goto **LOOP;	Endless loop.
@TEST_PARS	Subroutine.
OP=2;	Set output 2.
wait 2000;	Wait 2 seconds for testing the part.
MI=0;	Enable automatic handling of digital input #1.
goto ##LOOP;	
return	End of automatic function.
@AUTO_I1	Automatic handler for digital input #1.
OP=3;	Set output 3 to indicate that digital input #1 is sensed.
...	Subroutine body.
return	Subroutine end.
...	
exit	Exit program.

The mask may also be used to prevent switch bouncing from generating spurious routine calls.

Example:

A machine performs a periodic task. Digital input #1 is connected to a sensor to which the drive should react. The following code limits the automatic response to digital input #1 during execution of the #@AUTO_I1 routine code, even though digital input #1 may bounce.

##LOOP;	Repetitive task.
...	
MI=0;	Re-enable automatic routine.
...	
goto##LOOP;	

```
#@AUTO_I1  
MI=MI | 128 ;  
...  
return
```

Automatic handler for digital input #1.
Prevent nested calls to #@AUTO_I1.
Subroutine body.
Subroutine end.

Chapter 4: Program Development and Execution

The process of *SimplIQ* drive program development includes the following steps:

- Program editing: Write and/or edit the program.
- Compilation: Use the Compiler to process the program and find errors.
- Program loading: Load the program to the +flash memory of the drive.
- Debugging: Observe the behavior of the program and correct it where necessary.
- Running the program
- Saving to flash: Move the program permanently to the drive.

The Elmo Studio IDE includes all the tools needed to perform this procedure. Using the Elmo Studio is fully explained in Chapter 4 of the *Composer for SimplIQ Software Manual*.

4.1 Editing a Program

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

4.2 Compilation

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 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 and is described in Section 4.3.

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

While the Compiler can locate syntax errors, it cannot find:

- 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.

The following table enumerates the list of compilation errors.

Error

Error Code	Error String	Meaning	Example
0	No errors	Successful compilation without errors.	
1	Bad format	General error for bad syntax in right- or left-hand expression.	<pre>for k = 1::10 Double colon between 1 and 10. b = a + () Empty expression in parentheses. k = 1:2:20 Colon expression used out of for statement.</pre>
2	Empty expression	Expected right-hand expression is missing.	<pre>a = ; Right-hand expression is missing after assignment symbol.</pre>
3	Stack is full	Stack has overflowed its capacity.	
4	Bad index expression	An index expression of a variable is not evaluated in a single value.	<pre>a(2,3) The result of evaluation of the expression in parentheses is two values, not a single value. a() The index expression is empty.</pre>
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 at function definition.	<pre>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. Here, the type long is unknown.</pre>
6	Parentheses mismatch	The number of opening parentheses does not match the number of closing parentheses. This applies to both parentheses <i>and</i> square brackets.	<pre>b = a(1)) There is an unneeded closing parenthesis.</pre>
7	Value is expected	A right- or left-hand expression is not evaluated in a single value or it has failed during evaluation due to a bad syntax expression.	<pre>b = ^ a ; A value is expected before the ^ operator.</pre>

Error Code	Error String	Meaning	Example
8	Operator is expected	During right-hand expression evaluation, there is no operator or terminator of a simple expression after successful value evaluation.	<pre>b = a c ;</pre> 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 not enough space in the data 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 less than two values – in the colon expression may also cause this error.	<pre>for k = 10:-1:5:9</pre> Colon expression contains more than three values. <pre>for k = a</pre> Expected colon expression is missing in the for statement, after the =.
11	Name is too long	Variable or function name exceeds 12 characters.	<pre>int iuyuafdsf_876234 ;</pre>
12	No such variable	Left-hand value is not recognized as either a variable or as a system command.	<pre>de = sin(0.5)</pre> de is neither a variable nor a function.
13	Too many dimensions	Dimension of array exceeds the maximum admissible number of dimensions (syntax allows only a one dimensional array).	<pre>int arr[12][2];</pre> An attempt is made to define a two dimensional array.
14	Bad number of input arguments	The number of input arguments during a function call does not match the number of input arguments at the function definition.	<pre>function func(float a);</pre> ... <pre>func(a, 2);</pre> The number of input arguments during the function call is two, while the function func is defined with only the input argument.
15	Bad number of output arguments	Bad syntax in left-hand expression: multiple output without brackets, or multiple output that exceeds maximum admissible number of outputs (maximum of 16 outputs is allowed).	<pre>a, b = func(1,2);</pre> Multiple outputs must be within brackets.

Error Code	Error String	Meaning	Example
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).	<pre>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,</pre> <p>The number of input arguments exceeds 16.</p>
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.	
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. In this case, email technical support at asusid@elmo.co.il . Attach the Composer date and version (in the Help menu) and the program that you attempted to compile.	
22	Function definition is inside another function or flow control block	An illegal function definition.	<pre>if a<0 a = 0; function func (int a); end</pre> <p>Attempt to define a function inside if block.</p>
23	Too many functions	The user program contains too many functions and there is not enough space for them in the database.	

Error Code	Error String	Meaning	Example
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.	<pre>int switch;</pre> <p>switch is a keyword, so its use as a variable name is illegal.</p>
25	Name is not distinct	A variable or function name is not unique.	<pre>int func ; function func (int a)</pre> <p>The function and the variable have the same name.</p> <pre>function func (int a) int a ;</pre> <p>The definition of local variable a is illegal because this function already contains a local variable a as an input argument.</p>
26	Variable name is invalid	<p>This error occurs when:</p> <ul style="list-style-type: none"> ▪ 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 a comma is missing. 	<pre>int_abc ;</pre> <p>The variable name has a leading underscore.</p> <pre>function (int a)</pre> <p>The function name is missing after the function keyword.</p> <pre>int a, b,</pre> <p>The variable name is missing after the comma.</p>
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.	<pre>int a b;</pre> <p>A command as a variable separator is missing between a and b.</p>

Error Code	Error String	Meaning	Example
28	Illegal global variable definition	<p>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:</p> <ul style="list-style-type: none"> ▪ 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 at definition outside the function differs from the type of the declaration inside the function. 	<pre>int a1 ; function func (int a) global float a1; The variable type a1 at its definition is int, while inside the function, it is declared as float.</pre>
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.	<pre>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.</pre>
30	Variable is undefined	Iteration variable in a for statement is not defined before it.	<pre>function func (int a) for k = 1:10 a = k; end return The iteration variable k is not defined before its use.</pre>
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.	

Error Code	Error String	Meaning	Example
32	Bad variable dimension	The legal variable dimension must be a positive number inside brackets. An expression inside brackets that is not evaluated into a number, or a number that is less than 1 (zero or negative) is illegal.	<pre>int arr[12];</pre> Variable dimension is negative
33	Bad function format	Appears at function definition when: <ul style="list-style-type: none"> Function name is not unique or is empty function definition does not match its prototype 	<pre>function func (int a); function func (float a)</pre> Type of input argument in function definition does not match type in prototype.
34	Illegal minus	A minus is illegal before a function call with multiple output arguments, and before parentheses that include multiple expressions.	<pre>[ab] = -func(c);</pre> Illegal minus before function call with multiple outputs. <pre>-(2+3), c/5);</pre> Illegal minus before multiple expressions within parentheses.
35	Empty program	User program is empty.	
36	Program is too long	User program exceeds maximum admissible length.	
37	Bad function call	An attempt has been made to jump at the goto statement to a function with a non-zero number of input or output arguments.	<pre>[a,b,c] - func(x,y) +5;</pre> Illegal sentence. The Compiler checks if 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.	<pre>if a = 0 end</pre> An expression expected after 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

Error Code	Error String	Meaning	Example
40	Line compilation is failed	A general error has occurred during an attempt to compile an expression.	
41	Case must follow switch	After a switch statement, the only legal statement is case ; otherwise, this error occurs.	<pre>switch a b = 0; case 1, b = 1; end</pre> <p>The expression b=0 is illegal because switch must be followed by case.</p>
42	Illegal case after otherwise	The otherwise statement must be the last statement of a switch block. Any case after otherwise is illegal.	<pre>switch a case 1, b = 1; otherwise b = 0; case 2, b = 1; end</pre> <p>The statement case 2 is illegal because otherwise must be the last statement of switch.</p>
43	Bad nesting	Flow control block contradiction: else without if , mismatched end , flow control 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 "=" sign after the iteration variable name in a for statement.	<pre>for k a = 0; end</pre> <p>A = is missing after the k.</p>
46	Too many errors	The compilation error buffer is full.	
47	Expression is out of function	A function or label must contain executable code; otherwise this error occurs.	<pre>int a1; a1 = 0; function func (int a) Executable code is illegal because it is out of the function.</pre>

Error Code	Error String	Meaning	Example
48	Otherwise without any case	An otherwise appears immediately after a switch statement.	<pre>switch a otherwise b = 0 ; end</pre> <p>After a switch statement, case is expected, not otherwise.</p>
49	Misplaced break	Break is legal only inside a switch , for or while block; otherwise, this error occurs.	<pre>If a < 0 break ; end</pre> <p>break in an if statement is illegal.</p>
50	Too many outputs	The number of actual output arguments during a function call exceeds the number of output arguments during function definition.	<pre>function [int b] = func (int a) ... return ; ... [c,d] = func(a) ;</pre> <p>The function call is illegal because the number of outputs is two, while this function is defined with a single output argument.</p>
51	Line is too long	User program contains a line of more than 128 characters.	
52	Clause is too long	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.	
54	Open file failure	There has been an attempt to open a non-existing 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 a goto and reset statement; otherwise, this error occurs.	

Error Code	Error String	Meaning	Example
57	Variable is array	An attempt has been made to assign the entire variable array, not its single member.	<pre>int arr[10]; ##START arr[1] = 0; arr = 0</pre> <p>The last expression is illegal because it tries to assign the entire array.</p>
58	Variable is not array	An attempt has been made to assign a scalar variable according to an index, as an array.	<pre>int a1; (scalar) ##START a1[1] = 0;</pre> <p>The last expression is illegal because it tries to assign a scalar variable according to an index.</p>
59	Mismatch between left- and right-hand side expressions	The number of left values does not match the number of values in the right side of the expression.	<pre>[a,b] = 12 + c;</pre> <p>The number of values on the left is two, while the number of values after evaluation of the right-hand expression is 1.</p>
60	Illegal local array	Syntax does not allow definition of a local array. The array must be global.	<pre>function func (int a) int arr[10]; ... return;</pre> <p>Local array is illegal.</p>
61	Function already has body	Function has more than one body.	<pre>function func (int a) wait 2000 return ; ... function func (int a) until a end</pre> <p>Text is illegal because function func has multiple bodies.</p>
62	Opcode is not supported by <i>SimplIQ</i> drives	The specified version of the <i>SimplIQ</i> drive does not support a certain virtual command.	
63	Internal compiler error	If this occurs, email technical support at asusid@elmo.co.il . Attach the Composer date and version (in the Help menu) and the program you have attempted to compile.	

Error Code	Error String	Meaning	Example
64	Expression is not finished	The user program contains an unfinished sentence: an ellipsis (...) may be missing to indicate that the expression is continued on the next line.	The last line of user text may be: <pre>a = b + 8 / 12 / (8^2*sqrt(2) - sin (3.14/2))...</pre> After the ellipsis, the next 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 <i>SimplIQ</i> drive's serial flash memory.	
66	Corrupted the <i>SimplIQ</i> drive setup files	The file containing the <i>SimplIQ</i> 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 <i>SimplIQ</i> drive setup	The allowed variable name length does not equal the length defined in the Compiler.	
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 with 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: <pre>/* Just a comment</pre> The comment block is not closed.
73	Bad function after reset	A reset statement must contain either an auto-routine, global label or user function without input arguments; otherwise, this error occurs.	reset func(12) A user function with an input argument comes after a reset keyword.

Error Code	Error String	Meaning	Example
74	Bad jump to label	Occurs when an attempt is made to jump to: <ul style="list-style-type: none"> ▪ 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 	<pre>function func () ... return ... goto##func</pre> <p>The last expression is illegal because func is not a label.</p>
75	Illegal nargout	The nargout keyword is used outside a function.	<pre>##START if nargout > 2</pre> <p>The keyword nargout is used outside a function.</p>
76	Function without body	An attempt has been made to call a function that has been defined but does not have a body.	
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.	<pre>goto START</pre> <p>Between goto and the label name, ## or #@ is missing.</p> <pre>for k = 1:10 ... goto##START ... end</pre> <p>The goto statement in the for loop is illegal.</p>
78	Auto routine is local	An auto-routine is defined as a local label.	<pre>function start (int a) ... #@AUTOEXEC ... return</pre> <p>The AUTOEXEC auto-routine is defined inside a function as a local label.</p>
79	Command has 'not program' flag	The program refers to a command that has a "Not program" flag; that is, it cannot be used inside a user program.	<pre>LS</pre> <p>The program attempts to use the LS flag, which has a "Not program" flag defined for it.</p>

Error Code	Error String	Meaning	Example
80	Image file too long	The Image file length exceeds the user code partition size.	
81	System function <code>tdif</code> is not supported by the <i>SimplIQ</i> drive	During evaluation of the wait flow control, the <code>tdif</code> system function must be defined inside the <i>SimplIQ</i> 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.	<code>BG=10;</code> The BG command has a "not assign" flag and is illegal.
83	Keyword is not implemented - for future use	An attempt has been made to use a non-implemented keyword or feature.	
84	Misplaced return	Illegal return, such as from a try-catch block	<code>try</code> ... <code>if a > 6</code> <code>return;</code> <code>end</code> <code>catch</code> ... <code>end</code> The return keyword is used inside the try block.
85	Try and catch keywords must be on a separate line	Any code on the same line with try or catch is illegal.	<code>try, AC=10,000,000</code> ... 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.	<code>#define</code> The identifier is missing.
88	The name of identifier is not valid	The name of the identifier of the #define directive is not valid.	<code>#define 2PI 6.24</code> <code>#define VER+</code> The first name has a leading number. The second name contains the + character. A valid name may contain only letters, numbers or underscores.

Error Code	Error String	Meaning	Example
89	Identifier was defined before	An attempt has been made to use a #define directive with the same identifier.	<pre>#define NUM 3 . . . #define NUM 4</pre> <p>The second statement is illegal because NUM has already been defined.</p>
90	Condition is missing	The condition is missing in an #if or #elseif directive.	<pre>#if</pre> <p>A condition must follow the #if directive.</p>
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	#else or #elseif has been used out of 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 .	<pre>#ifNUM > 10 #elseif NUM > 5 #else #elseif NUM > 3 #endif</pre> <p>The #else directive is not the last directive before the #endif.</p>
93	Identifier wasn't declared	The identifier of an #undef directive was not defined previously.	<pre>#undef NUM</pre> <p>This statement is illegal if it is not preceded by an #define NUM directive.</p>
94	Mismatched type	Type of evaluated constant expression is neither integer nor float. If error is received, contact asusid@elmo.co.il . Attach the Composer date and version (from the Help menu) and the program you have attempted to compile.	
95	Too many identifiers have been defined	User has defined more than 100 #define directives in the program.	
96	Misplaced #endif	#endif directive is not preceded by #if .	
97	Unknown error	Unknown error.	

4.3 Compiler directives

Prior to code generation, the Compiler preprocesses the user code, handling the Compiler directives. The preprocessor performs the following:

- Searches for literal constant definitions, 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.

4.3.1 #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 token_string
```

The **#define** directive substitutes *token_string* for all subsequent occurrences of an *identifier* in the source file. The *identifier* is replaced only when it forms a token. For instance, *identifier* is not replaced if it appears in a comment, within a string or as part of a longer identifier.

The **#define** directive without *identifier* is illegal.

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

The *token_string* argument consists of a series of tokens, such as keywords, constants or complete statements. One or more space characters must separate *token_string* from *identifier*.

A legal *identifier* name may consist of a maximum of 32 characters. A valid name consists only of letters, underscores and numbers (not leading zeros). The name is case sensitive, although uppercase *identifiers* are accepted.

The maximum admissible number of **#define** directives in the *SimplIQ* drive is 100.

When evaluating the *token_string* (refer to [section 4.3.9](#)), if the evaluation is successful, the *identifier* is replaced with the obtained value; otherwise, every appearance of the *identifier* is replaced with the *token_string* as a string.

The syntax that defines Compiler directives in the *SimplIQ* language differs from the C language in a number of ways. The redefinition of **#define** with the same *identifier* is illegal unless the second definition of **#define** appears after the first definition is removed with the **#undef** directive.



The *token_string* may contain identifiers of other **#define** directives as part of the expression. When such an expression is successfully evaluated, every occurrence of this identifier is replaced with the evaluation result even if the identifiers that are part of its *token_string* were removed with an **#undef** directive (described in Example 6 following).

Example 1:

```
#define DEBUG_FLAG
Defines the identifier DEBUG_FLAG.
```

Example 2:

```
#define ARR_LEN 10
Defines the identifier ARR_LEN as the integer constant 10. Each occurrence of ARR_LEN will be replaced by 10.
```

Example 3:

```
#define ARR_LEN num+10
Defines the identifier ARR_LEN as the string num+10. The precompiler cannot evaluate token_string because num is not a constant expression. Each occurrence of ARR_LEN will be replaced by this string.
```

Example 4:

```
#define DEBUG_FLAG
...
#define DEBUG_FLAG
The first statement defines the identifier DEBUG_FLAG and the second statement is illegal because it redefines the #define with an already-defined identifier.
```

Example 5:

```
#define DEBUG_FLAG
...
#undef DEBUG_FLAG
...
#define DEBUG_FLAG 1
The first statement defines the identifier DEBUG_FLAG and the second statement removes this identifier. The third statement defines DEBUG_FLAG again, and is legal because the previous definition of DEBUG_FLAG was removed by the #undef directive. This identifier has a token_string and each occurrence of DEBUG_FLAG will be replaced by 1.
```

Example 6:

```
#define MAX_LEN 10
#define ARR_LEN MAX_LEN +10
...
#undef MAX_LEN
...
int arr[ARR_LEN];
This example contains two identifiers with token_strings that have been successfully evaluated. The identifier ARR_LEN contains MAX_LEN as part of its constant expression. In the last statement, the identifier ARR_LEN will be replaced with 20 despite the fact that the identifier MAX_LEN was removed with the #undef directive.
```

4.3.2 #if

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

Syntax:

#if *constant-expression*

The *constant-expression* is either an integer or a float constant expression.

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
```

In this example, `REDUCE_MAX_LEN` will be defined as the integer constant 10 only if `MAX_LEN` is greater than 10; otherwise, it will not be defined at all.

4.3.3 #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 REDUCE_MAX_LEN 10
#else
    #define REDUCE_MAX_LEN 5
#endif
```

In this example, `REDUCE_MAX_LEN` will be defined as the integer constant 10 only if `MAX_LEN` is greater than 10; otherwise, it will be defined as the integer constant 5.

4.3.4 #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
#elif MAX_LEN > 20
    #define REDUCE_MAX_LEN 20
#elif MAX_LEN > 10
    #define REDUCE_MAX_LEN 10
#else
    #define REDUCE_MAX_LEN 5
#endif
```

The **#if**, **#elseif** and **#else** directives in this example are used to make one of four choices, based on the value of `MAX_LEN`. The constant `REDUCE_MAX_LEN` is set to 30, 20, 10 or 5, depending on the definition of `MAX_LEN`.

4.3.5 #endif

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

Syntax:

endif

The **#endif** directive without a previous **#if** directive generates an error.

4.3.6 #ifdef

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

Syntax:

#ifdef *identifier*

The **#ifdef** and **#ifndef** directives can be used anywhere that **#if** can be used. The **#ifdef** *identifier* statement is equivalent to **#if 1** when *identifier* has been defined, and 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
    printf("Parameter is equal to %d", debug_var);
#endif
```

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

4.3.7 #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 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 prevents the possible redefinition of `DEBUG_FLAG`.

4.3.8 #undef

The **#undef** directive, as in C, removes the current definition of the specified name. All subsequent occurrences of the 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 the *SimplIQ* 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.

4.3.9 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 the **#define** directive, the *identifier* must have a successfully-evaluated *token_string*.



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 valid operators are:

- Calculating operators: * / = - %
- Logical operators: && ||
- Comparison operators: == != < > <= >=
- Bitwise operators: & | << >>
- Unary logical operators: !
- Unary bitwise operators: ~

A detailed description of the operators is given in [section 2.2.2](#). 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 calculating operations, if both operands are integers, the result is integer; otherwise, the type is float.

4.4 Program Execution

The following sections describe how to run a program.

4.4.1 Initiating a Program

A program is initiated by the XQ command, which indicates at which label execution should begin.



The XQ command does not reset program variables; initial values for all variables must be set by the user.

The XQ command clears the call stack, kills any pending automatic routines and clears the interrupt mask. For a description of the XQ command, refer to [section 4.4.5](#).

4.4.2 Halting and Resuming a Program

The HP Interpreter command can be used to stop execution of the user program and the automatic routines. HP freezes the status of the program and does not reset it. A later XC command can resume the program from the instruction at which the program was halted. Pending interrupts will remain pending.

Example 1:

MO=1;	Start motor.
JV=2000;	Set jog speed.
##LOOP;	Repetitive task
BG;	
wait(1000)	Wait and switch direction.
JV=-JV;	
goto##LOOP;	Repeat.

This program indicates that the motor should travel at 2000 counts/second for one second, then reverse direction for one second, and then continue to travel back and forth forever.

If the HP command is applied when the program is waiting (executing the wait(1000) instruction), the motor will continue to travel at the same direction for an unlimited time.

An XC command reverses the direction immediately, because the waiting time has already elapsed.

Example 2:

A servo axis in a machine has two different tasks to perform, in two different machine modes. The following routine implements the two tasks.

##TASK1;	
##LOOP;	Repetitive task 1.
...	Task 1 body.
goto##LOOP;	Repeat task 1.
##TASK2;	
##REP;	Repetitive task 2
...	Task 2 body.
goto##REP;	Repeat task 2.

The first task is invoked by XQ##TASK1. In order to switch to the second task, the first task must be killed before:

```
HP;
XQ##TASK2;
```

4.4.3 Automatic Program Execution with Power Up

If the autoexec function is included in the user program, the program line following function declaration will be performed at power up.

4.4.4 Save to Flash

Because a program is downloaded to a non-volatile memory, it is always saved. Therefore, information is not lost in case of power down. As noted in [section 5.1.2.2](#), the CP command clears the entire user area in the flash and the running program must be killed (not halted) before the command is issued.

4.4.5 Running, Breaking and Resuming

The XQ command starts program execution from a label or executes a function, as follows:

- XQ##MYFUNCTION(a,b,c) runs the function MYFUNCTION(a,b,c).
- XQ cannot return values from a function.
- XQ##LABEL runs from ##LABEL.
- XQ runs from the start of the user program code.
- XQ without a parameter is illegal.
- XQ does not return a value.



Notes:

- XQ without a label or function name is designed to run a program written in the Elmo Saxophone or Clarinet; that is, without function definitions, local variables and so on. If the program starts from a function definition, the XQ command without a label or function name causes an error to occur. If the function starts from a label, the dummy start label is inserted into the Function Symbol table and execution begins at the start of the program.
- XQ does not change the current values of global variables; the initial values of global parameters must be set by the programmer.

KL=0 kills all virtual machines, if any are running.

KL stops the motor.

HP halts all virtual machines; they can be continued later by the XC command.

If HP halts inside a wait statement, the wait time stops running while the program is halted.

XC continues all virtual machines.

4.4.6 The Elmo Studio

The Elmo Studio IDE provides the environment for debugging a user program. It includes these functions:

- Error reporting
- Pause and continue a program
- Breakpoints
- Run to cursor
- Step in
- Step out
- Step over
- Watch global variables
- Watch auto/local variables
- View call stack functions

Chapter 5: User Program Maintenance

This chapter describes how user programs are downloaded to *SimplIQ* drives and how existing user programs may be retrieved from drives.



Under most circumstances, the procedures of this chapter are only performed by the Elmo Studio environment.

5.1 Downloading and Uploading a Program

After a successful compilation, the compiled code can be downloaded to the *SimplIQ* drive. This step is supported by the Elmo Studio, a part of the Composer IDE. Before each download, the Elmo Studio automatically clears the flash memory sector, which is used for saving the user program.

The serial flash performs downloads and uploads using two commands: DL and LS. Both commands use the auxiliary LP command, which is a vector integer command. The CP command clears the user flash area, and the CC command verifies checksum and enables the newly loaded program for use.

5.1.1 Binary Data

The *SimplIQ* drive flash memory is interfaced with binary data. Sending the binary data on the RS-232 lines is problematic, because they do not differentiate between data and delimiters.

Characters that are problematic for sending on RS-232 lines are:

- All numbers from 128 to 255
- All possible terminators: 0, <CR>, <LF>, semicolon (;) and comma (,)
- Equals sign (=)
- Backspace
- <ESC>

In order to prevent this problem, a hexadecimal binary format is used during data upload and download, although this increases the amount of data to be transmitted.

Every byte in hexadecimal format consists of two numbers (such as 0x12), considered to be a single character. For example, the 8-bit number 0x12 in hexadecimal binary format is the sequence of two characters: 1 and 2.

The representation of numbers in the DSP flash memory differs from its representation inside a personal computer.

- The 8-bit number is represented the same way.

- The 16-bit number, represented in hexadecimal format as equal to 0x1234, is represented in the DSP memory in the following two bytes:
Byte 1 equals the value 0x12 in hexadecimal form
Byte 2 equals the value 0x34 in hexadecimal form
- The 32-bit number, represented in hexadecimal format as equal to 0x12345678, is represented in the DSP memory in the following four bytes:
Byte 1 equals the value 0x56 in hexadecimal form
Byte 2 equals the value 0x78 in hexadecimal form
Byte 3 equals the value 0x12 in hexadecimal form
Byte 4 equals the value 0x34 in hexadecimal form

Binary data to be loaded to the serial flash is represented in this format.

Examples:

Number in Hexadecimal Form in PC	Sequence of Characters in Hex Binary Form for Transmission
0x12	12
0x1234	1234
0x12345678	56781234

5.1.2 Auxiliary Upload/Download Commands

5.1.2.1 The LP[N] Command

This command sets the properties of the serial flash data upload and download; it is used together with the DL and LS commands ([sections 5.1.3.1](#) and [5.1.4.1](#)).

- LP[1] defines the byte (out of 128K bytes in the flash) at which the next action should start.
- LP[2] defines how many byte to send (LS command).
- LP[3] specifies the start address (bytes) of the user/factory program partition in the flash.
- LP[4] specifies the size (bytes) of the user/factory program partition in the flash.

The TW[11] parameter indicates the type of program partition: user or factory.

5.1.2.2 The CP Command

This command clears the entire user area in the serial flash. The clearing operation may take a significant amount of time. CP sets the Program Valid flag to -1.

Failures in executing the CP command may occur if:

- The motor is on.
- A program is running.

5.1.2.3 The CC Command

The CC=xxxx command performs the following:

- Reads the actual length of the user partition from the flash memory table of contents (TOC).
- Calculates a 32-bit checksum for the entire user partition. The checksum is calculated by totaling all the consecutive 2-byte sequences (short integer numbers) that occupy user program space, along with the checksum number itself. The total result must be zero to pass.
- If the resulting checksum matches xxxx, sets the Program Ready flag (CC returns 1) and copies functions and variable symbol tables from the user partition to the internal DSP flash memory. If the checksum does not match xxxx, an error code is issued (CC returns 0).

Failures in executing the CC command may occur if:

- The actual length of the TOC is less than two flash pages, or if it exceeds the user program address limit.
- The calculated checksum does not match xxxx.

5.1.3 Downloading a Program

5.1.3.1 The DL Command

The DL command downloads data to the serial flash memory of the drive. The command is used primarily to download compiled user programs to the drive. When downloading to a non-protected area in the flash, the process is as follows:

```
LP[1]=start;
```

```
DL##xxxxxxxxxx<ESC>CS;
```

where:

xxxxxxxxxx denotes the escape-sequenced data payload.

start denotes the byte address in the user program flash.

CS denotes the 16-bit checksum for the message, including DL##.

The DL process takes time to proceed, because it needs to burn and verify.

Failures in executing the DL command may occur if:

- An attempt is made to write to a protected area of the flash. While the DL may begin to write legally to the flash, its last bytes may attempt an illegal (protected) write. In any such case, the DL command will be rejected and the contents of the serial flash will be unpredictable.
- DL is used when the motor is on.
- DL is used when a program is running.
- There is a faulty checksum. In this case, DL will be rejected but no harm will be caused.

- A verify error occurs. If DL attempts to write to an error in the flash that was written to previously, the write will probably fail due to a Verify error. In this case, the contents of the flash will be unpredictable and it will need to be cleared and completely rewritten.
- The DL string is too long. The maximum length of a DL string is 500 bytes, due to internal *SimplIQ* drive limits.
- The Program Valid flag is not -1. In this case, the DL command will not be executed (CP must be issued before DL).

5.1.3.2 The Program Download Process

In order to download a program image to the *SimplIQ* drive, perform the following procedure:

1. Read the location (*loc*) and the length of the user code partition, using the LP[3] and LP[4] commands.
2. Verify that your image block will fit inside the allocated space.
3. Clear the program flash using the CP command.
4. Download your image file using the following sequence:


```
LP[1]=loc;
DL##...100 bytes of payload
LP[1]=(loc+100)
DL##...next 100 bytes of payload
... and so on until the end of the image
```
5. Use the CC=checksum command to declare the end of loading and to verify the entire download process.

5.1.4 Uploading a Program

5.1.4.1 The LS Command

The LS command is used to upload a program that resides in the drive flash, for backup or for further editing. This option is disabled when a program is running. After program upload, the user can modify it and then return to the compilation step. When uploading from a non-protected area in the flash, the process is as follows:

```
LP[1]=start;
LP[2]=payload net length
xxxxxxxxxx<ESC>CS;
```

where:

xxxxxxxxxx denotes the escape-sequenced data payload.
 start denotes the byte address in the user program flash.
 CS denotes the 16-bit checksum for the message.

Failures in executing the LS command may occur if the LS sequence (including <ESC>s) exceeds 200 characters (internal *SimplIQ* buffer management limit). The LS output will be terminated with the proper checksum and terminator. The data transmitted in the payload will be good and meaningful, but not to the defined length.

5.1.4.2 The Program Upload Process

In order to upload a program image from the *SimpliQ* drive, perform the following procedure:

1. Read the location (*loc*) and the length (*len*) of the user code partition from the main TOC, using the LP[3] and LP[4] commands.
2. Upload your image file using the following sequence:
LP[1]=*loc*;
LP[2]=100
Use LS to get the next 100 bytes of the payload.
LP[1]=(*loc*+100);
Use LS to get the next 100 bytes of the payload.
... and so on until the end of the image.