CPL User Guide

P/N 16-124981
Rev 00
November 2019

This page for notes

TABLE OF CONTENTS

ABOUT THIS MANUAL

Overview and Scope

Copley Programming Language (CPL) is a high level programming language used to run on Copley's Virtual Machine (CVM). This manual describes the installation and use of CPL.

Related Documentation

CANopen-related documents:

- *Copley ASCII Interface Programmer's Guide* (describes how to send ASCII format commands over an RS232 serial bus to control one or more amplifiers)
- *Copley Amplifier Parameter Dictionary* (describes the parameters used to program and operate Copley Controls amplifiers)

Links to these publications, along with hardware manuals and data sheets, can be found under the *Documents* heading at: http://www.copleycontrols.com/Motion/Downloads/index.html

Copley Controls software and related information can be found at:
http://www.copleycontrols.com/Motion/Products/Software/index.html

Comments

Copley Controls welcomes your comments on this manual. See http://www.copleycontrols.com for contact information.

Copyrights

Document Validity

We reserve the right to modify our products. The information in this document is subject to change without notice and does not represent a commitment by Copley Controls. Copley Controls assumes no responsibility for any errors that may appear in this document.

Product Warnings

Observe all relevant state, regional and local safety regulations when installing and using Copley Controls amplifiers. For safety and to assure compliance with documented system data, only Copley Controls should perform repairs to amplifiers.

DANGER

Hazardous voltages.

Exercise caution when installing and adjusting Copley amplifiers.

Risk of electric shock.

On some Copley Controls amplifiers, high-voltage circuits are connected to mains power. Refer to hardware documentation.

Risk of unexpected motion with non-latched faults.

After the cause of a non-latched fault is corrected, the amplifier re-enables the PWM output stage without operator intervention. In this case, motion may re-start unexpectedly. Configure faults as latched unless a specific situation calls for non-latched behavior. When using non-latched faults, be sure to safeguard against unexpected motion.

Latching an output does not eliminate the risk of unexpected motion with non-latched faults.

Associating a fault with a latched, custom-configured output does not latch the fault itself. After the cause of a non-latched fault is corrected, the amplifier re-enables without operator intervention. In this case, motion may re-start unexpectedly.

For more information see:

When operating the amplifier as a CAN node, the use of CPL or ASCII serial commands may affect operations in progress. Using such commands to initiate motion may cause network operations to suspend.

Operation may restart unexpectedly when the commanded motion is stopped.

Use equipment as described.

Operate amplifiers within the specifications provided in the relevant hardware manual or data sheet.

FAILURE TO HEED THESE WARNINGS CAN CAUSE EQUIPMENT DAMAGE, INJURY, OR DEATH.

Revision History

Revision	Date	Comments
00	Nov 2019	Initial Release to Agile

CHAPTER 1: INTRODUCTION

This chapter provides an overview of the Copley Controls CPL programming language.

Topics include the following:

1.1: Overview

CPL is a high level programming language for writing custom CVM programs to single or dual axis Copley drives. CPL expands on Indexer 2's capabilities with interrupts and features that are faster and more flexible with looping and branching capabilities.

This manual provides detailed information on writing code and running, testing and debugging programs.

1.2: Specifications

1.2.1: CVM Memory

- RAM: 8K Words
- Flash Memory: 32K Words

1.2.2: Supported Drives

- Xenus Plus series
- Accelnet Plus series (Does not include the AEP)
- Stepnet Plus series

1.3: Host Computer Requirements

1.3.1: Computer and Operating System

Operating systems Supported:

- Windows 7, 8, 10

CHAPTER 2: INSTALLATION, STARTUP, AND COMMUNICATIONS

This chapter describes how to install, start, and set up communications for CPL. Perform the steps outlined below.

Topics include the following:

2.1: Install CPL Software

Optionally download software from the Web

1. Choose or create a folder where you will download the software installation file.
2. In an internet browser, navigate to:
   http://www.copleycontrols.com/Motion/Downloads/index.html
3. Under *Software Releases*, click on *CPL*.
4. Enter user name and password.
5. When prompted, save the *CPL.zip* file to the folder chosen or created in Step 1.
   The folder should now contain a file named *CPL.zip*.
6. Extract the contents of the zip file to the same location.
   The folder should now contain the files *CPL.zip* and *Setup.exe*.
7. If desired, delete *CPL.zip* to save disk space.

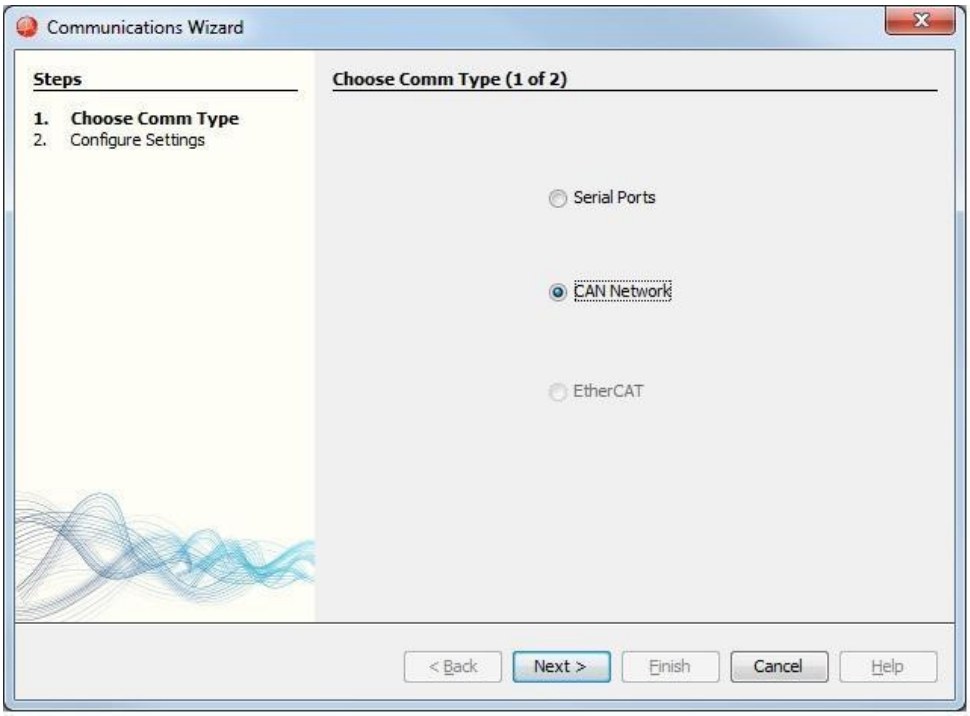
Install CPL Software from a CD or hard drive

1. If installing from a CD, insert the CD (Copley Controls part number *CPL*).
   Normally, inserting the CD causes the installation script to launch, and a CPL Installation screen appears. If so, skip to Step 3.
2. If the software installation file is on a hard drive, navigate to the folder and then double-click on *Setup.exe*
   OR
   if you inserted the CD and the CPL *Installation* screen did not appear, navigate to the root directory of the installation CD and then double-click on *Setup.exe*.
3. Respond to the prompts on the CPL *Installation* screens to complete the installation. We recommend accepting all default installation values.

2.2: Configure Communications

2.2.1: Choose a Communications Type

In the Menu Bar choose **Tools→Communications Wizard**. Choose a communication type and click **Next**.



There are three communication types to choose from:

Serial Communications

For each PC-to-amplifier connection via serial port:

- One standard RS-232 serial port or a USB port with a USB-to-RS-232 adapter.
- One serial communication cable. See amplifier data sheet for part numbers.

CANopen Communications Protocol

- One Copley Controls CAN PCI network card (part number CAN-PCI-02).
  CPL also supports CAN network cards made by these manufacturers: Copley, KVaser and IXXAT.
- One PC-to-amplifier CANopen network cable. See amplifier data sheet for part numbers.

See the amplifier data sheet for CAN network wiring instructions.

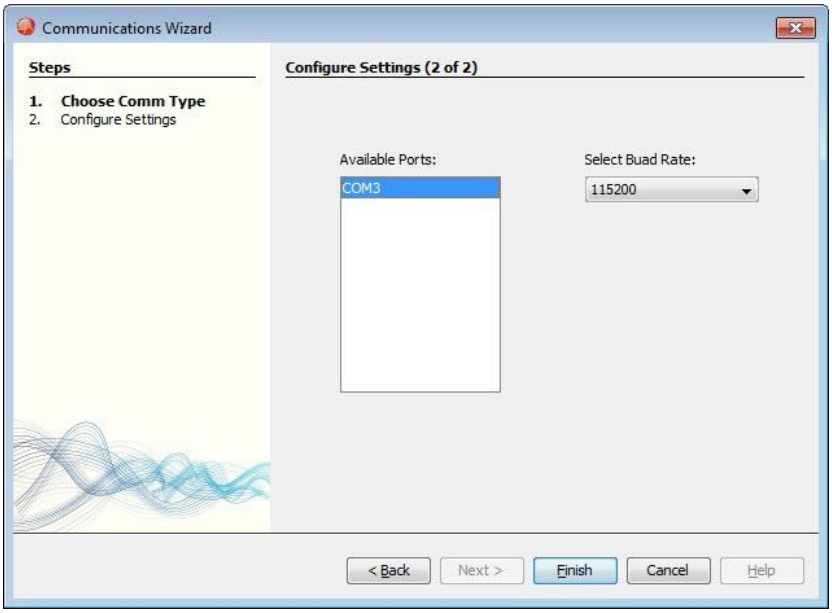
EtherCAT Communication Network

- One Ethernet adapter.
- One EtherCAT network cable, (see data sheet).

2.2.2: Configure Settings

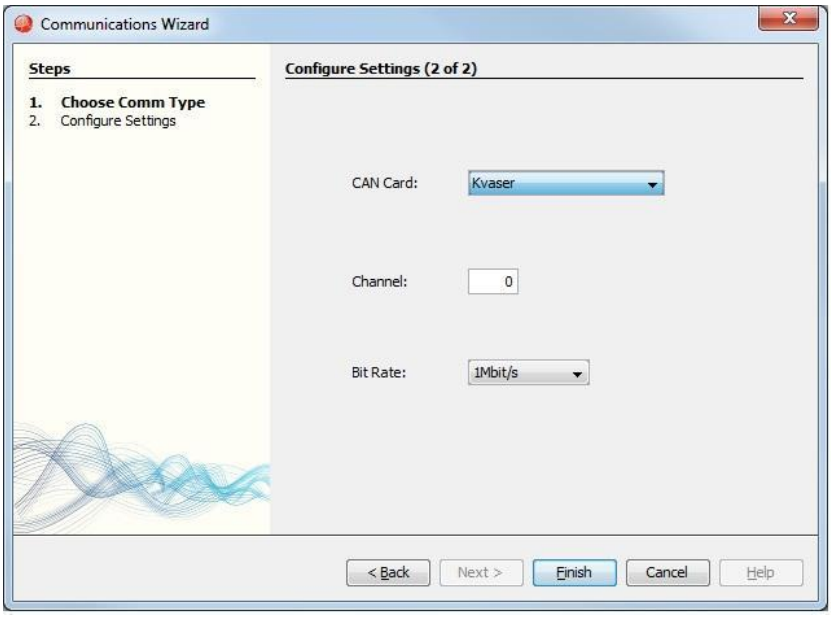
Serial Communications

Select a COM port and baud rate then click **Finish**.



CANopen Communications

Select the CAN card, channel and Bit Rate and click **Finish**.



CHAPTER 3: LANGUAGE BASICS

This chapter explains the language basics of CPL.

Topics include the following:

3.1: Structure of a CPL Program

There are three parts to a CPL program:

- `main()` entry point function
- Global variables
- Functions.

3.1.1: The `main()` Function

The `main()` function is the entry point for every CPL program, therefore every CPL program must contain exactly one `main()` function. Program execution starts with the first line of code in `main()`. The syntax is:

```
main()


end main
```

The `main()` function, and functions, can contain any number of local variables, statements, and function calls.

3.1.2: Global Variables and Functions

Global variables and functions are optional and are declared outside of `main()`. Below is an example of the `main()` function with a global variable and a function declaration.

```
int x

main()

end main

function void startMove()

end function
```

This is the global variable declared outside of the `main()` function

This is the `main()` function

This is the function outside of the `main()` function

Go to Global and Local Variables (p.19) for a more thorough description of global variables.

Go to Functions (p. 34) for a more thorough description of a function.

3.2: Data Types

3.2.1: Primitive Data Types

There are 5 primitive data types available in CPL: `short, ushort, int, uint,` and `float,`. Each has its own range. When a primitive data type is called for, it is recommended that `int or float` be used. Use `short` only when it is required in the system functions. For size and range details please see the table below.

Primitive Data Types		
Type	Size	Range
`short`	16-bit signed	-32768 to 32767
`ushort`	16-bit unsigned	0 to 65535
`int`	32-bit signed	–2,147,483,648 to 2,147,483,647
`uint`	32-bit unsigned	0 to 4294967295
`float`	32-bit IEEE 754 floating point number	Positive Range: 1.1754944E-38 to 3.4028235E+38 Zero: 0.0 Negative Range: -3.4028235E+38 to -1.1754944E-38

Literals

Literals are hard coded numbers for primitive data types.

Examples:

- int and short can be represented as decimal or hex. Hex numbers are preceded by 0x. The x can be upper or lower case.

  For example: `16 or 0x10`

- floats can be represented with or without exponent

  For example: `150.0 or 1.5E2`

  Note: Default data type for literal numbers is `int` and `float`

Promoting data

Promoting data converts a value from a smaller data type into a larger data type, such as `short` to `int`, `short` to `float`, or `int` to `float`. Because the value was originally in the range of a smaller data type, promoting it into a larger data type will not change its value.

Demoting data

Demoting data converts a value from a larger data type into a smaller data type, such as `int` to `short`, `float` to `int`, or `float` to `short`. When demoting data, it is possible to lose data because the value in the larger data type could be outside the value range allowed in the smaller data type. For instance, if an `int` is demoted to a `short` it will lose its upper 16 bits of data. In this case the compiler will generate a warning.

3.2.2: Derived Data Types

CPL uses two derived data types, `arrays` and `structs`. They are both a collection of primitive data types grouped as a single variable.

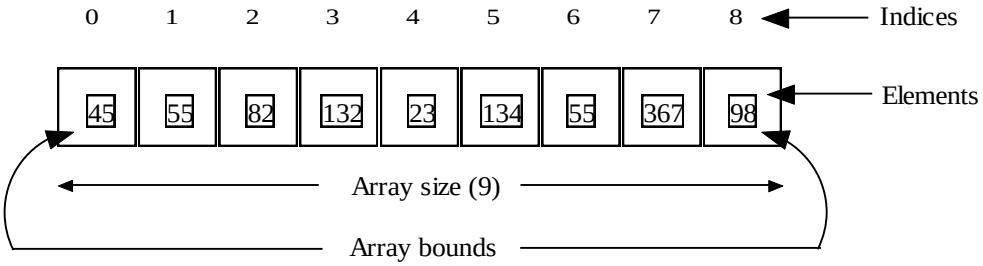
- Note: Assigning one array or struct to another array or struct is not allowed.

Arrays

An array is a fixed number of values of a single primitive data type. The size of the array is specified, as a positive integer, when it is declared, and cannot be changed. The values inside the array are called elements. Each element is referred to by its index. The first index is zero. The last index is the array size minus one. The first and last elements of the array are its bounds.

CPL does not perform run-time checking of array bounds. Reading outside the array bounds will result in indeterminate values being returned. Writing outside of the array bounds will result in overwriting memory, causing unpredictable program operation. It is highly recommended that bounds checking be performed before accessing an array.

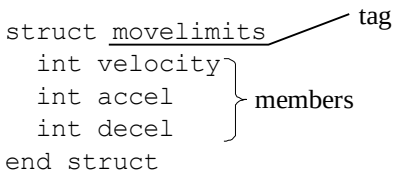
Below is an array that has a size of 9 with array bounds of 0 and 8.



Structs

A struct is a collection of values using one or more primitive data types. The keyword `struct` is required, followed by a struct name, called a tag. Structs are useful for grouping related values using a single variable.

For example:



For more examples of structs see Declaring Structs on page 22.

3.3: Variables

Variables are memory locations, referred to by a name, that store values for use in a program.

3.3.1: Naming Variables

When naming variables, keep the following in mind:

- Letters, numbers, or underscores can be used when naming a variable (may not start with a number).
- Variable names are case sensitive.
- There is no size limit to a variable name.

3.3.2: Global and Local Variables

Variables can be Global or Local.

Global variables

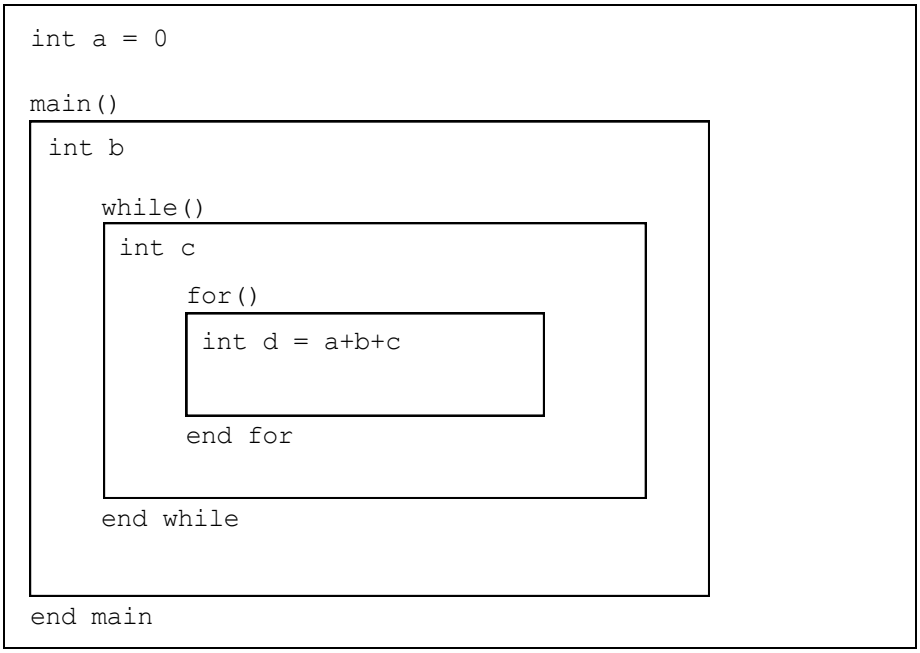
Any variable declared outside of a function, including `main()`, is a global variable. Global variables are visible to any function, and can be used by any function. Global variables declared in one file can also be used in another file. An example of a global variable is contained in the diagram below.

Local variables

Variables that are declared inside a block of code are called local variables. A block of code is either a function or one of the program control flow statements. Local variables can be defined anywhere in the block, but they must be declared before they can be used. They can be initialized with numeric values, global variables, expressions, function calls, and other local variables (if previously defined in the function). An example of a local variable is contained in the diagram below.

Visibility of variables

The degree of access to variables in a program depends on the block in which they are defined, and where those blocks are situated with respect to other blocks. This is called visibility or scope. The diagram below, and its accompanying text, describes the visibility of global and local variables within a program.



In the example above:

- `a` is global
- `b, c`, and `d` are locals
- `a` can be used in any function or block within a function
- `b` can be used anywhere in the `main()` function or any of `main()`'s enclosed blocks, after it's declared
- `c` can only be used within the `while` block, which includes the `for` block (after its declared)
- `d` can be used only in the `for` block (after its declared)

3.3.3: Constants

When the keyword `const` appears in a variable declaration, the variable's initial value cannot be changed.

3.3.4: Declaring Variables

All variables must be given a data type and a name in order for them to be used in a program. This declaration informs the compiler how much space to reserve in memory and what name will be used to refer to that memory location. If a variable is not given an initial value it will default to zero. The following shows how to declare and initialize variables.

```
// declare variables
int a
float b

// declare and initialize variables
int a = 1
float b = 1.4

// declare and initialize on the same line
int a = 1, b = 2, c

// declare a constant
const int MOVE_DISTANCE = 10000
```

Declaring primitive types

To declare a primitive type the following is needed:

- data type
- variable name
- initial value (optional)

If no initial value is provided, the primitive type will be set to the default value of 0.

Declaring arrays

To declare an array the following is needed:

- data type
- variable name
- array size
- Initial values (optional)

The size must be a positive integer. The array size must be a literal or const variable. The array size can be accessed by using the array name and the reserved word size, separated by the dot operator. For example:

```
myArrayVar.size
```

Certain rules also apply when initializing arrays. Arrays must be initialized with literals only. If no initial values are provided, the array will be set to the default value of 0. Keep in mind that if element values are provided, values must be provided for all elements in the array.

Examples of array declarations:

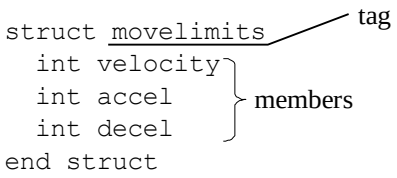
```
// declare an array with three elements
int velArray[3]

// declare multiple arrays of the same type
int velArray[3], accelArray[3], decelArray[3]

// declare and initialize an array with three
// elements
int velArray[3] = 4, 5, 6
```

3.3.5: Declaring Structs

There are two steps involved in using structs: defining the struct, and declaring the struct variable.

1. Define the struct

To define a struct it must have the keyword `struct` and a tag name (any valid variable name). Struct members must also be declared using primitive variable declarations. Members cannot be arrays or other structs.



The example below defines a struct.

```
// define a struct with MoveLimits as the tagname
struct MoveLimits
    int vel
    int accel
    int decel
end struct
```

2. Declare the struct variable

To declare a struct variable the following is needed:

- The keyword `struct`
- The tag name chosen from step 1 above
- A variable name

See the example below.

```
// declare a variable of the MoveLimits struct type
struct MoveLimits limits

// declare and initialize a variable of the
// MoveLimits struct type
struct MoveLimits limits = 100000, 200000,200000
```

To access struct members the variable name from step 2 above is needed, followed by a dot operator (.), and the `struct` member name. See the example below.

```
// access a member of the struct
limits.vel = 250000
```

3.4: Registers

CPL has 32 registers that can be used to pass data to and from external controllers to CPL programs. Each register is 32 bits long. The syntax for program registers is R_n$, where n is a register number (0-31).

$R0 = 34

Control applications (HMI, PLC, or PC-based programs) can use any of the supported protocols to read and write the CPL registers. Supported protocols include the Copley ASCII Interface, CANopen.

Current register values can also be viewed in the IDE as long as the drive is connected.

To view Program Registers see CVM Program Registers (p.72).

- Note: When a CPL program is started the register values are always set to zero.

3.5: Operators

Operators are a set of symbols that perform specific operations on values (also called operands) in a function, and then return a result. Details of specific kinds of operators used in CPL follow.

3.5.1: Arithmetic

Basic arithmetic operators

+	**Addition**
-	**Subtraction**
*	**Multiplication**
/	**Division**
%	**Remainder** Returns the remainder of an integer division. Example: 5 % 3 = 2

3.5.2: Unary

Operators that require one operand

-	**Unary minus** negates an expression show example
!	**Logical complement** converts a non-zero operand to a 0, and a zero to a 1
++	**Increment** Increments a value by 1. Must be used with a variable and not a constant or literal. May be used pre-incrementally (++a), or post-incrementally (a++). If used to increment a simple standalone statement, the pre and post increments result is the same (a++ would equal ++a). However, If used in a larger expression, the post incremented expression a++ evaluates to the original value, while the pre incremented expression ++a evaluates to the incremented value.
--	**Decrement** Decrements a value by 1. Follows the same rules as the increment operator.

3.5.3: Equality and Relational

Used to test if values are equal to, less than or greater than each other. A relational expression evaluates to 1 if the expression is true. It evaluates to 0 if the expression is false.

==	**Equal to**
!=	**Not equal to**
>	**Greater than**
>=	**Greater than or equal to**
<	**Less than**
<=	**Less than or equal to**

3.5.4: Conditional

Used to compare two or more relational expressions

&&	**Conditional AND** If both operands are non-zero, the result is 1, otherwise the result is zero. When the result is non-zero, the remaining expressions are not evaluated. Example: (x>2) &&(x<10) In the above conditional AND operation, the condition is met (evaluates to 1) if x=3 through 9. Consequently, the remaining expressions are not evaluated.
||	**Conditional OR** If either operand is non-zero the result is 1, otherwise the result is zero. When the result is non-zero, the remaining expressions are not evaluated. Example: (x==3) II (x==4) In the above conditional OR operation, the condition is met (evaluates to 1) if x=3 or 4. Consequently, the remaining expressions are not evaluated.

3.5.5: Bitwise

Operations are performed on integers in their binary form.

~	**Complement** (also a unary operator) inverts a bit pattern. Example: In the digital form: ~0 becomes 1 1 becomes 0 0110 becomes a 1001
&	**AND** When two corresponding bits both equal 1, a 1 is returned, otherwise a 0 is returned. Example: 6 & 4 = 4 The binary view of the same operation: 0110 0100 0100 An AND operation requires two operands.

|	**OR** When either of two corresponding bits equal 1, a 1 is returned. Otherwise a 0 is returned. Example: 6 | 4 = 6 The binary view of the same operation: 0110 0100 0110 An OR operation requires two operands.
^	**Exclusive OR** When either of two corresponding bits are the same, a 0 is returned, otherwise a 1 is returned. Example: 6 ^ 4 = 2 The binary view of the same operation: 0110 0100 0010 An exclusive OR operation requires two operands.
<<	**Left shift** Shifts the bits of an integer, in its binary form, a given number of spaces. Example: 1 << 3 = 8 The binary view of the same operation: 0001 << 3 = 1000
>>	**Unsigned right shift** Shifts the bits of an integer, in its binary form, to the right by a given number, consequently the most significant bit (MSB) becomes 0. Example: 8 >> 3 = 1 The binary form of the same operation: 1000>>3=0001
>>>	**Signed right shift** Shifts the bits of an integer, in its binary form, to the right by a given number, and the MSB remains unchanged. Example: 8 >>> 3 = 9 The binary form of the same operation: 1000>>3=1001

3.5.6: Assignment Operator (=) and Compound Assignment Operators:

A Compound assignment operator combines an operator and an = sign.

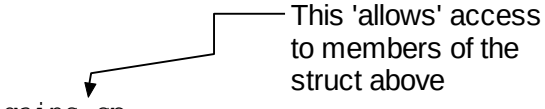
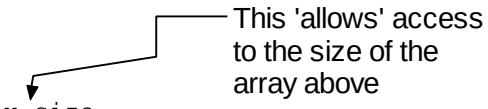
Example:

x +=3 is the same as x=x+3

=	**Assignment**
+=	**Addition assignment**
-=	**Subtraction assignment**
*=	**Multiplication assignment**
/=	**Division assignment**
%=	**Remainder assignment**
&=	**Bitwise AND assignment**
|=	**Bitwise OR assignment**
^=	**Bitwise exclusive or assignment**
<<=	**Left-shift assignment**
>>=	**Right-shift assignment**

3.5.7: Misc

()	**Parenthesis grouping, function calls**
[]	**Array indexing**

.	**Dot operator**
	Used with structs and arrays. Examples follow.
	Allows access to members of a struct.
	Example:
	`struct gains`
	`  int cp`
	`  int ci`
	`end struct`
	 
	
	`gains.cp`
	Used with arrays to gain access to element size.
	Example:
	`int x [3]`
	 
	
	`x.size`

3.5.8: Operator Precedence

Operators with higher precedence are executed first. Two or more operators with the same precedence get evaluated in the order shown below.

Order of evaluation	Operators in order of precedence
Left to right	( ) [ ] .
Right to left	++ -- - ~ ! (
Left to right	* / %
Left to right	+ -
Left to right	<< >> >>>
Left to right	< > <= >=
Left to right	== !=
Left to right	&
Left to right	^
Left to right	|
Left to right	&&
Left to right	||
Right	= += -= *= /= %= &= ^= |= <<= >>= >>>=

3.6: Expressions, Statements and Blocks

The following defines expressions, statements, and blocks.

3.6.1: Expressions

An expression is made up of one or more operators, variables, literals and/or one or more function calls which, when run, evaluates to a single value.

For example:

```
X + 1 / 4
```

3.6.2: Statements

Statements are one or more expressions, function calls, or declarations that complete a task.

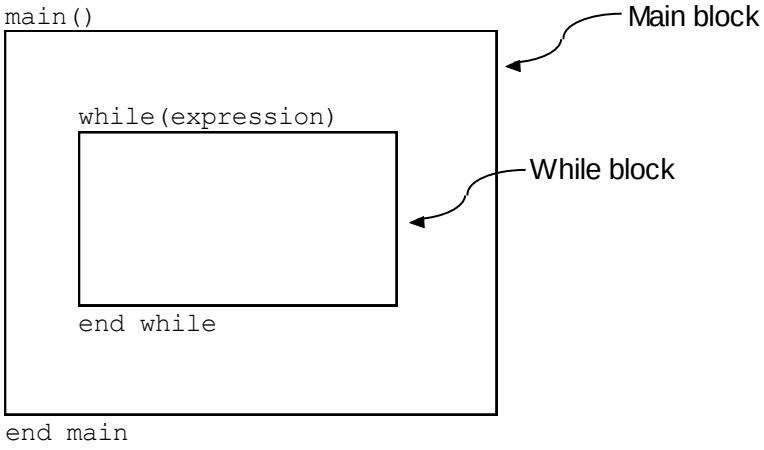
Statements are made up of one or more of the following:

- Assignment (see
- Unary operator
- Function call
- Variable Declaration
- Control flow

Control flow statements are used to regulate the order in which statements are executed.
A statement can span multiple lines using an underscore at the end of the line. However, the underscore may not split variables or numbers. For example, the number 100 cannot be split between two lines. There is no limit to expression or line length. However, there can be only one statement per line. To terminate a line use carriage return, line feed, or carriage return/line feed.

3.6.3: Blocks

Blocks are zero or more statements within a function or control flow statement; essentially a section of code grouped together. A block begins after the start of the function or control flow statement, and ends with the corresponding end statement. Blocks may also be nested within other blocks, as in the example below.



3.7: Program Control Flow Statements

Control flow statements allow a program to loop or branch. There are the six available control flow statements:

- While loop
- For loop
- if-else
- switch
- break
- continue

Details for each control flow statement follow.

3.7.1: While Loop

A while loop executes a block of code repeatedly as long as a condition is met (expression does not evaluate to zero).

The syntax for a while loop is:

```
while(expression)
        ←————————  while block
end while
```

The following numbered list describes what happens in the diagram above:

1. While loop is entered.
2. Expression evaluated.

- If it evaluates to a non-zero number, the block is executed. Expression evaluation and block execution are repeated as long as the expression evaluates to a non-zero number.
- If the expression evaluates to zero the block is skipped, the `end while` terminates the `while` loop, and the code immediately following the `end while` is read.

Here is an example of a while loop:

```
// increment x while its value is between y and z
while ( (x > y) &&  (x < z) )
    x += 1
end while
```

3.7.2: For Loop

Typically, a for loop is used to repeat a block of code a specific number of times. The flexibility of a for loop allows it to be used to execute a block of code repeatedly while a condition is being met, similar to a while loop.

The syntax for a for loop is:

```
for(declaration;expression;iterator)
        <-------- for block
end for
```

The following numbered list describes what happens in the diagram above:

1. For loop is entered.
2. Declaration is executed (this happens only once).
3. Expression evaluated.

   If it evaluates to a non-zero number, the block is executed. Move to step 4 below.

   If it evaluates to zero the for loop is exited and the code immediately following the `end for` is read.
4. Iterator is executed (if there is one). If there is no iterator step 3 and 4 are repeated.

   Expression evaluation, block execution and iterator execution (if there is one) are repeated as long as the expression evaluates to a non zero number.

Note that the declaration, expression, and iterator are all optional. However, the parentheses, along with the two semicolons are required. If there is no declaration, expression or iterator an infinite loop will result.

Here is an example of a for loop:

```
// calculate average of values in an array
sum = 0
for (int x = 0; x < myArray.size; x++)
    sum += myArray[x]
end for
sum /= myArray
```

3.7.3: If Statement

An if statement is used for branching. A single block of code will be executed if its expression evaluates to a non zero number. An if statement has three components: `if` (with an expression), `elsif` (with an expression) and `else`. `else` and `elesif` are optional. Multiple `elseif`s may be used. An if statement is useful if two or more acceptable conditions are possible.

The syntax for an if statement is:

```
if(expression)
        <-------------  if block
elseif(expression)
        <-------------  elseif block
else
        <-------------  else block
end if
```

The following numbered list describes what happens in the diagram above:

1. If statement is entered.
2. Expression in the if statement is evaluated. If it evaluates to a non-zero number, block a is executed the line of code immediately following the `end if` is read.
3. If the expression in the if statement evaluates to zero, the expression in the `elseif` statement is evaluated. If the expression in the `elseif` statement evaluates to a non-zero number, block b is executed and the line of code immediately following the `end if` is read.
   If the expression in the `elseif` statement evaluates to zero, block c is executed, and the line of code immediately following the `end if` is read.

Here is an example of an `if` - `elseif` - `else` statement:

```
// if x is negative set y to -1, else if x = 0
// set y to 0, otherwise set y to 1
if (x < 0)
    y = -1
elseif (x == 0)
    y = 0
else
    y = 1
end if
```

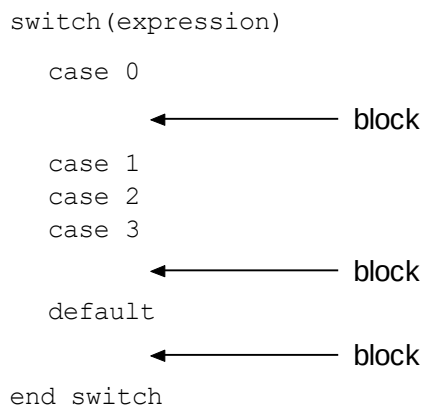
3.7.4:

Switch Statement

A switch statement is similar to an if statement in that it allows branching to different blocks of code that match the expression. If the case value equals what the expression evaluates to, that case's block is executed. Program execution will continue to the next case or default block, unless a `break`, or `return` is encountered.

When a `break` is encountered, the switch statement is exited, and the line of code immediately following the `end switch` is read. `return` causes a *return* to the calling function, in this case, the beginning of the switch statement. The default block is optional. If `default` is provided, and none of the case values match the value of the expression, the default block will be executed. Duplicate case values are not allowed. Case values must be constants or literals.

The syntax for a switch statement is:



Here is an example of a switch statement with `break` statements inserted:

```
// Call appropriate function to set up limits
// based on the mode variable
switch (mode)
    case 1:
        InitPositionLimits()
        break

    case 2:
    case 3:
        InitVelocityLimits()
        break

    default:
        InitCurrentLimits()
end switch
```

One or more case statements may be grouped to a single block as in the diagram above.

3.7.5: Break

A break statement terminates the closest enclosing loop or switch statement. `break` can be used in for, while and switch statements.

Here is an example of `break` used in a for statement:

```
// search an array for the first zero value and
// save the index of the array where it is found
foundIndex = -1
for (int x = 0; x < myArray.size; x++)
    if (myArray[x] == 0)
        found = x
        break
    end if
end for
```

3.7.6: Continue

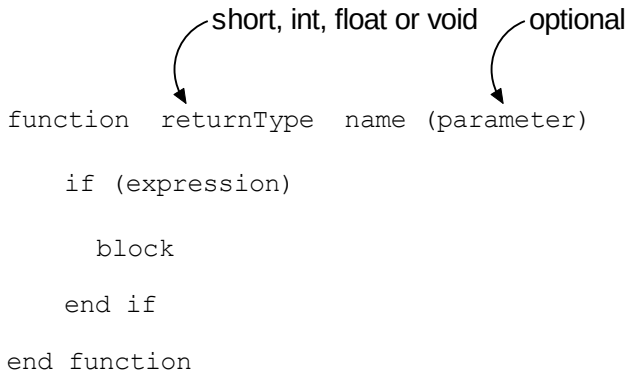
A continue statement passes control to the next iteration of the enclosing loop. `continue` can be used in for and while statements.
Here is an example of `continue` being used in a for statement:

```
// add up only the positive numbers in an array
sum = 0
for (int x = 0; x < myArray.size; x++)

    /*
      If the array element contains a negative
      number, then pass control to the next
      iteration of the loop without executing the
      code after the end if
    */
      (myArray[x] < 0)
        continue
    end if

    sum += myArray[x]
end for
```

3.8: Functions

Functions allow a program to be broken down into small well defined tasks. They are helpful in allowing the reuse of code that is used often in a program.

The syntax for a function is:



Functions have the option of returning values. They may also take parameters as input. Parameters may be primitive data types or derived data types. However, each is passed to a function differently: Primitive data types are passed by value; derived types are passed by reference.

3.8.1: Passing by Value

When a primitive data type is passed to a function, it is passed by value, meaning a copy of the original data is made and the copy is passed to the function. If the function changes the value, it is really changing the copy, not the original data. When the function returns, the copy is discarded and any changes are lost, but the original data is preserved.

3.8.2: Passing by Reference

When a derived data type is passed to a function, it is passed by reference, meaning an address of the data is passed to the function, not the data itself. If a function changes the data, the original data is changed, because it uses the address of the original data. When the function returns, the changes that the function made to the data are preserved.

3.8.3: Return Type

The `returntype` in a function can be `short`, `ushort`, `int`, `uint`, `float` or `void`. A return type of void means that a function does not return a value. Arrays and structs are not allowed as return types. However, an array or struct may be returned by passing it in as an argument and letting the *function* change its value. This is possible because arrays and structs are passed by reference as described above.

3.8.4: Function Name

A function name may be any valid variable name and is typically named for what it does.

3.8.5: Parameter List

The optional parameter list is defined using valid variable declarations separated by commas.

```
(int x, float y, struct MoveLimits limits)
```

If the function does not take any parameters, then an empty set of parenthesis is used.

```
function void CheckStatus()


end function
```

- Note: Initializing a variable in the parameter list is not allowed.

Arrays are declared without specifying the array size. The caller is responsible for declaring the array before the function is called.

```
(int[] velocities)
```

To prevent a function from modifying an array or struct, use the const modifier when declaring it in the parameter list.

```
(const int[] velocities)
(const struct MoveLimits limits)
```

3.8.6: Return Statement

The return statement is used to exit the function. If the function declaration specifies that a value is to be returned, then an expression must follow the return statement. If the return type is void, then the return statement is optional.

Examples:

```
function void StartMove(struct MoveLimits limits)
      //function body

      // no return statement needed
end function


function int GetMaxValue(int x, int y)
      if (x > y)
          return x

      else
          return y
end function
```

3.8.7: Calling Functions

A function can be called from anywhere in the program (main or other functions) as long as it is defined. A function is called by:

1. Optionally assigning a variable if the function returns a value.
2. Using the function name.
3. Providing the arguments to the parameter list, each separated by a comma. The data type of each argument must match the type in the function definition.

Example:

```
// function definition
function int GetMaxValue(int x, int y)

end function

// function call
int maxValue = GetMaxValue(a, b)
```

CHAPTER 4: SYSTEM FUNCTIONS

This chapter has a list of CPL system functions and detailed definitions of each.

4.1: System Functions

4.1.1: Motion

4.1.2: Wait

- Note: Wait function calls are ‘blocking. Meaning they won’t return from the function call until either the condition has been met or the time out has expired.

4.1.3: Status

4.1.4: Math

4.1.5: Miscellaneous

4.2: Definitions

Motion

int Move(int distance, int axis=0)

Description Start a move in position mode.

Note: This function only works for a single axis. To start a move on more than one axis at the same time (multi-axis drives only), first set up the commanded position (parameter 0xCA), then call the TrajUpdate system function.

Pre-condition The desired state (parameter 0x24) must be set to 21 (for servo mode) or 31 (in stepper mode). Drive must be hardware enabled. No faults present. The trajectory profile mode parameter, 0xC8, needs to be configured properly for the move type.

Parameters distance:

Relative: Number of counts to move from current position

Absolute: Abosolute position in counts

axis: Which axis the command is to be applied (default is axis A). Parameter is optional.

Return value OK = 0, error = 1

int VelMovePosMode(int velocity, int direction, int axis=0)

Description Start a velocity move using the trajectory generator in the position loop.

Pre-condition The desired state (parameter 0x24) must be set to 21 (for servo mode) or 31 (in stepper mode). Drive must be hardware enabled. No faults present. The trajectory profile mode parameter, 0xC8, needs to be configured properly for the move type.

Parameters velocity: Commanded velocity (0.1 counts/s). Positive values only.

direction: Direction of motion.

1 = Positive

-1 = Negative

axis: Which axis the command is to be applied (default is axis A). Parameter is optional.

Return value OK = 0, error = 1

int VelMoveVelMode (int velocity, int axis=0)

Description	Start a velocity move using programmed velocity in the velocity loop.
Pre-condition	The desired state (parameter 0x24) must be set to11. Drive must be hardware enabled. No faults present. Note: When setting desired state to 11 and commanded velocity is not 0, motion may occur.
Parameters	velocity: Commanded velocity (0.1 counts/s). axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, error = 1

int CurrentMove(int current, int currentRamp, int axis=0)

Description	Start a current move using programmed current mode.
Pre-condition	The desired state (parameter 0x24) must be set to 1. Drive must be hardware enabled. No faults present. Note: When setting desired state to 1 and commanded current is not 0, motion may occur.
Parameters	current: Commanded current (0.01 A). currentRamp: Rate at which the current will change to its commanded value (mA/s). If this parameter is 0, no motion will occur. axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, error = 1

int Home(int axis=0)

Description	Starts the homing sequence.
Pre-condition	The homing method configuration (parameter 0xC2) must be configured for the appropriate homing type. The desired state (0x24) must be set to a position mode, either servo or stepper.
Parameters	axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, error = 1

int Halt(int axis=0)

Description	Abort the move in progress. Note: This command only works in position mode.
Pre-condition	None
Parameters	axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, error = 1

int TrajUpdate(int axis)

Description	Update the trajectory generator. If a move is in progress, the trajectory parameters will be updated. If no move is in progress, a new move is started. Note: This command only works in position mode.
Pre-condition	All of the trajectory parameters (velocity, accel., etc) must be set up prior to calling this function.
Parameters	axis: A bit-mapped value that indicates which axis or axes the command is to be applied. Bit 0 for axis A, bit 1 for axis B. If no bits are set, then the command will be applied to axis A. This parameter is required.
Return value	OK = 0, error = 1

Wait

int Wait(int time)

Description	Wait for a fixed period of time.
Pre-condition	None
Parameters	time: Time to wait (ms). A negative value means wait forever.
Return value	OK = 0

Note: Wait function calls are ‘blocking. Meaning they won’t return from the function call until either the condition has been met or the time out has expired.

int WaitMoveDone(int timeout, int axis=0)

Description	Wait for the move to be done. Note: The move must be started before this command.
Pre-condition	None
Parameters	timeout: Maximum time to wait (ms). A negative value means wait forever. axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, timeout = 2

Note: Wait function calls are ‘blocking. Meaning they won’t return from the function call until either the condition has been met or the time out has expired.

int WaitForEvent(int mask, int condition, int timeout, int axis=0)

Description	Wait for an event to occur.
Pre-condition	None
Parameters	mask: Represents the bits in the event status parameter(0xA0. condition: The condition that triggers the wait to exit 0 = All the bits set. 1 = Any of the bits set. 2 = All of the bits clear. 3 = Any of the bits clear. timeout: Maximum time to wait (ms). A negative value means wait forever. axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, timeout = 2

Note: Wait function calls are ‘blocking. Meaning they won’t return from the function call until either the condition has been met or the time out has expired.

int WaitForInput(int inputNumber, int condition, int timeout)

Description	Wait for an input condition to be met.
Pre-condition	None
Parameters	inputNumber: The input to wait on (IN1, IN2, etc.). condition: The state of the input that will trigger the Wait to exit: 0 = Low level. 1 = Falling edge. 2 = High level. 3 = Rising edge. timeout: Maximum time to wait (ms). A negative value means wait forever.
Return value	OK = 0, timeout = 2

Note: Wait function calls are ‘blocking. Meaning they won’t return from the function call until either the condition has been met or the time out has expired.

int WaitForActualPosition(int position, int condition, int timeout, int axis=0)

Description	Wait for the actual position to meet the specified condition.
Pre-condition	None
Parameters	position: The actual position to wait for (counts). condition: The condition that triggers the wait to exit. 0 = Greater than or equal to the specified position. 1 = Less than or equal to the specified position. timeout: Maximum time to wait (ms). A negative value means wait forever. axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, timeout = 2

Note: Wait function calls are ‘blocking. Meaning they won’t return from the function call until either the condition has been met or the time out has expired.

int WaitForLimitedPosition(int position, int condition, int timeout, int axis=0)

Description	Wait for the limited position to meet the specified condition. This function is typically used for stepper motors operated in open-loop stepper mode.
Pre-condition	None
Parameters	position: The input limited position to wait for (counts). condition: The condition that triggers the wait to exit. 0 = Greater than or equal to the specified position. 1 = Less than or equal to the specified position. timeout: Maximum time to wait (ms). A negative value means wait forever. axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, timeout = 2

Note: Wait function calls are ‘blocking. Meaning they won’t return from the function call until either the condition has been met or the time out has expired.

int WaitForVelocity(int velocity, int condition, int timeout, int axis=0)

Description	Wait for the actual velocity.
Pre-condition	
Parameters	velocity: The actual velocity to wait for (.1 counts per second). condition: The condition that triggers the wait to exit. 0 = Greater than or equal to the specified velocity. 1 = Less than or equal to the specified velocity. timeout: Maximum time to wait (ms). A negative value means wait forever. axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, timeout = 2

int WaitForVelocityTraj(int velocity, int condition, int timeout, int axis=0)

Description	Wait for the trajectory velocity.
Pre-condition	
Parameters	velocity: The trajectory velocity to wait for (0.1 counts per second). condition: The condition that triggers the wait to exit. 0 = Greater than or equal to the specified trajectory velocity. 1 = Less than or equal to the specified trajectory velocity. timeout: Maximum time to wait (ms). A negative value means wait forever. axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, timeout = 2

int WaitForCurrent(int current, int condition, int timeout, int axis=0)

Description	Wait for actual current.
Pre-condition	None
Parameters	current: Actual current to wait for (0.01 A). condition: The condition that triggers the wait to exit. 0 = Greater than or equal to the specified trajectory current. 1 = Less than or equal to the specified trajectory current. timeout: Maximum time to wait (ms). A negative value means wait forever. axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	OK = 0, timeout = 2

Status

int GetFaults(int axis=0)

Description	Reads the Latching Fault Status register (0xA4) which contains any active latched faults.
Pre-condition	None
Parameters	axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	The value of parameter 0xA4

void ClearFaults(int axis=0)

Description	Clears any latched faults.
Pre-condition	None
Parameters	axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	N/A

int GetEvents(int axis=0)

Description	Reads the Event Status Register (0xA0).
Pre-condition	None
Parameters	axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	The value of parameter 0xA0

int GetStickyEvents(int axis=0)

Description	Reads the Sticky Event Status register (0xAC).
Pre-condition	None
Parameters	axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	The value of parameter 0xAC

int GetLatchedEvents(int axis=0)

Description	Reads the Latched Event Status register (0xA1).
Pre-condition	None
Parameters	axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	The value of parameter 0xA1

int GetTrajStatus(int axis=0)

Description	Reads the Trajectory Status register (0xC9).
Pre-condition	None
Parameters	axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	The value of parameter 0xC9

Math

float SQRT(float a)

Description	Calculates the square root of the input parameter.

float LOG(float a)

Description	Calculates the natural log of the input parameter.

float EXP(float a)

Description	Calculates *e* raised to the power of the input parameter.

float POW(float a, float b)

Description	Calculates a raised to the b power.

float SIN(float a)

Description	Calculates the sine (in radians) of the input parameter.

float COS(float a)

Description	Calculates the cosine (in radians) of the input parameter.

float TAN(float a)

Description	Calculates the tangent (in radians) of the input parameter.

float ASIN(float a)

Description	Calculates the arc sine (in radians) of the input parameter.

float ACOS(float a)

Description	Calculates the arc cosine (in radians) of the input parameter.

float ATAN(float a)

Description	Calculates the arc tangent (in radians) of the input parameter.

float ATAN2(float a, float b)

Description	Calculates the arc tangent (in radians) of the quotient of a and b.

float FLOOR(float a)

Description	Calculates the largest (closest to positive infinity) value that is less than or equal to the input parameter.

float CEILING(float a)

Description	Calculates the smallest (closest to negative infinity) value that is greater than or equal to the input parameter.

float ABS(float a)

Description	Calculates the absolute value of the input parameter.

Miscellaneous

int SetParameter16(int paramId, short newValue, int bank, int axis=0)

Description	Set the value of a parameter in the drive.
Pre-condition	None
Parameters	paramId: The ID of the parameter (see Copley Controls' Parameter Dictionary for details). newValue: The new value for the parameter axis: Which axis the command is to be (default is axis A). Parameter is optional. Bank: RAM=0 Flash=1
Return value	OK = 0, error = 1

Short GetParameter16 (int paramId, int bank, int axis=0)

Description	Get the value of a parameter from the drive.
Pre-condition	None
Parameters	paramId: The ID of the parameter (see Parameter Dictionary for details). axis: Which axis the command is to be applied (default is axis A). Parameter is optional. Bank: RAM=0 Flash=1
Return value	Returns the value of the specified parameter.

int SetParameter32(int paramId, int newValue, int bank, int axis=0)

Description	Set the value of a parameter in the drive.
Pre-condition	None
Parameters	paramId: The ID of the parameter (see Parameter Dictionary for details). newValue: The new value for the parameter axis: Which axis the command is to be applied (default is axis A). Parameter is optional. Bank: RAM=0 Flash=1
Return value	OK = 0, error = 1

int GetParameter32(int paramId, int bank, int axis=0)

Description	Get the value of a parameter from the drive.
Pre-condition	None
Parameters	paramId: The ID of the parameter (see Parameter Dictionary for details). Bank: RAM=0 Flash=1 axis: Which axis the command is to be applied (default is axis A). Parameter is optional.
Return value	Returns the value of the specified parameter.

int SetParameterExt(int paramId, short [ ] newValues, int bank, int axis=0)

Description	Set a multi-word parameter in the drive. This is used for parameters that take more than two words of data.
Pre-condition	None
Parameters	paramId: The ID of the parameter (see Parameter Dictionary for details). newValues: An array of type short . axis: Which axis the command is to be applied (default is axis A). Parameter is optional. Bank: RAM=0 Flash=1
Return value	OK = 0, error = 1

int GetParameterExt (int paramId, short [ ] paramValue, int bank, int axis=0)

Description	Get a multi-word parameter in the drive. This is used for parameters that take more than two words of data.
Pre-condition	None
Parameters	paramId: The ID of the parameter (see Parameter Dictionary for details). axis: Which axis the command is to be applied (default is axis A). Parameter is optional. Bank: RAM=0 Flash=1 paramValue: The value of the specified parameter will be returned here.
Return value	OK = 0, error = 1

int SetElecGearRatio(int inputPulses, int outputCounts, int axis=0)

Description	Sets electronic gear ratio.
Pre-condition	None
Parameters	inputPulses: Number of Input Pulses required to produce output counts. outputCounts: Number of Output Counts per given number of input pulses. axis: Which axis the command is to be applied (defaulted to axis A). Parameter is optional.
Return value	OK = 0, error = 1

int ReadInputs(int inputMask)

Description	Reads the 32-bit version of the Input Line State Parameter (0x15C)
Pre-condition	None
Parameters	A bit mapped integer that indicates which input or inputs are to be read. When a bit is set, the state of the corresponding input will be returned. IN1 corresponds to bit 0, IN2 corresponds to bit 1, etc.
Return value	The state of the inputs specified in the inputMask parameter. The value of IN1 is returned in bit0 (1 if input is hi, 0 if input is low), IN2 in bit 1, etc.

int SetOutput(int outputNumber, int state)

Description	Sets an output to the active/inactive state.
Pre-condition	Output must be set to manual mode using the SetOutputConfig system function.
Parameters	outputNumber: The output number to control. state: 1 = active, 0 = inactive
Return value	OK = 0, error = 1

int SaveGetRegs(int operation, int registerNumber)

Description	This function will save a program register to flash, or restore a program register from flash.
Pre-condition	None
Parameters	operation: 1 = save, 0 = restore registerNumber: the number register to save/restore. Valid inputs are 0-32. Note: Passing 32 as the registerNumber will save or restore all registers.
Return value	OK = 0, error = 1

int SetCANObj16(int objID, int newValue, int subIndex, int axis)

Description	This function sets a CAN object value. See the CANopen Programmers Manual for a list of CAN objects.
Pre-condition	None

Parameters	objID: ID number of the CAN object newValue: The new value the CAN object will be set to subIndex: The sub-index of the CAN object axis: Which axis the command is to be applied (defaulted to axis A). Parameter is optional.
Return value	OK = 0, error = 1

int GetCANObj16(int objID, int subIndex, int axis)

Description	This function gets a CAN object value. See the CANopen Programmers Manual for a list of CAN objects.
Pre-condition	None
Parameters	objID: ID number of the CAN object subIndex: The sub-index of the CAN object axis: Which axis the command is to be applied (defaulted to axis A). Parameter is optional.
Return value	Value of the CAN object

int SetCANObj32(int objID, int newValue, int subIndex, int axis)

Description	This function sets a CAN object value. See the CANopen Programmers Manual for a list of CAN objects.
Pre-condition	None
Parameters	objID: ID number of the CAN object newValue: The new value the CAN object will be set to subIndex: The sub-index of the CAN object axis: Which axis the command is to be applied (defaulted to axis A). Parameter is optional.
Return value	OK = 0, error = 1

int GetCANObj32(int objID, int subIndex, int axis)

Description	This function sets a CAN object value. See the CANopen Programmers Manual for a list of CAN objects.
Pre-condition	None
Parameters	objID: ID number of the CAN object subIndex: The sub-index of the CAN object axis: Which axis the command is to be applied (defaulted to axis A). Parameter is optional.
Return value	Value of the CAN object

CHAPTER 5: INTERRUPTS

This chapter describes Interrupts and their usage.

Topics include the following:

5.1: Introduction

Interrupts are used to handle asynchronous events. When an interrupt occurs, the virtual machine first finishes executing the current instruction, then calls the interrupt routine. When the interrupt routine is finished, the code resumes from where it left off. If an interrupt occurs while another is being handled, the most recent will be handled upon the exit of the current interrupt handle

All interrupts are disabled by default. To be active they must be enabled within the CPL program using the `GlobalEnableInterrupts()` call (see Global Enable/Disable Interrupts, p. 56). Individual interrupts are enabled by implementing the interrupt handler routine (see Interrupt Handler Routines, p. 55).

A mask determines what bit/s will trigger an interrupt. The mask is defined by the interrupt type (see Interrupt Types, p. 54). All interrupts are edge triggered.

5.2: Interrupt Types

Each interrupt has a pre-defined function.

5.2.1: interrupt_1: program exception

interrupt_1 is triggered by program exceptions. The interrupt mask defines which program exceptions will cause an interrupt. Exceptions are serious conditions that may cause unexpected program operation. It is highly recommended that interrupt_1 be implemented so that proper action can be taken if one of these exceptions occurs for your specific application. The following program exceptions are currently used:

Bit	Definition
0	Attempt to read/write an illegal address (stack overflow will generate this).
1	Attempt to write to a read only memory location.
2	Divide by zero.
3	Illegal op-code processed.
4-31	Reserved for future use.

5.2.2: interrupt_2: rising edge of digital inputs

interrupt_2 is triggered by the rising edge of a digital input. The interrupt mask defines which input/s will cause the interrupt to occur. Bit 0 for input 0, bit 1 for input 1, etc. For example: an interrupt mask of 0x70 corresponds to inputs 4, 5, and 6.

5.2.3: interrupt_3: falling edge of digital inputs

interrupt_3 is be triggered by the falling edge of a digital input. The interrupt mask defines which bits will cause the interrupt. Bit 0 for bit 1, bit 1 for bit 2, etc.

5.2.4: interrupt_4: events status for axis 1

interrupt_4 is triggered by the events status for axis 1. interrupt_4 is generated on the rising edge of enabled events status bits. The interrupt mask defines which input/s will cause the interrupt. For example: an interrupt mask of 0x180 corresponds to current output limited and voltage output limited events status bits.

Refer to CME2 User Guide for events status parameter.

5.2.5: interrupt_5: events status for axis 2

interrupt_5 is triggered by the events status for axis 2. The interrupt mask defines which bits will cause the interrupt. Interrrupt_5 is generated on the rising edge of enabled events status bits. It can be used only in multi-axis drives.

5.2.6: interrupt_6: events status for axis 3

interrupt_6 is triggered by the events status for axis 3. The interrupt mask defines which bits will cause the interrupt. Interrrupt_6 is generated on the rising edge of enabled events status bits. It can be used only in multi-axis drives.

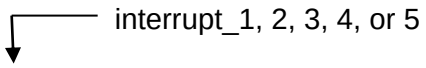
5.2.7: interrupt_7: events status for axis 4

interrupt_7 is triggered by the events status for axis 4. The interrupt mask defines which bits will cause the interrupt. Interrrupt_7 is generated on the rising edge of enabled events status bits. It can be used only in multi-axis drives.

5.3: Interrupt Handler Routines

5.3.1: Adding the Interrupt

To enable an individual interrupt, an interrupt routine must be implemented. The syntax for the interrupt routine is shown below.




```
interrupt_1 <interrupt mask>
   User code
end interrupt
```

5.3.2: i_return

An `i_return` statement is used to exit the interrupt handler routine. Interrupt routines do not return any values.

```
interrupt_2 <0x06>
   if (expression)
     i_return
   end if
end interrupt
```

5.4: Global Enable/Disable Interrupts

When a CPL program starts all interrupts are disabled by default. To enable interrupts use the `GlobalEnableInterrupts()` call. This will enable all interrupts. They can be disabled at any point in the program by calling the `GlobalDisableInterrupts()` described below.

GlobalEnableInterrupts()	
Description	Enables the global interrupt.
Pre-condition	Interrupt service routines must be defined.

GlobalDisableInterrupts()	
Description	Disables the global interrupt.
Pre-condition	None

5.5: Interrupt Status

An `ReadInterruptStatus()` call may be used to view the bit/s that trigger an interrupt routine. An integer value is returned.

Example:

```
interrupt_3 <0x0F>
   int triggerValue = ReadInterruptStatus()
   if( triggerValue == 1)
     SetOutput ( 1,0 )
   end if
end interrupt
```

5.6: Good Practices

Interrupts should remain as short and simple as possible. If an interrupt is being used to execute several lines of code it is recommended to use a flag.

Example:

```
main()
   int flag = 0
   while(1)
      if(flag)
         //if interrupt set flag, do some task here
      end if
   end while
end main
```

```
interrupt_2 <0x02>
   //on input 1 rising edge, set flag to do task
   flag = 1
end interrupt
```

If an interrupt function is configured to be triggered on more than one option bit, it is possible that ReadInterruptStatus() will return more with more than one bit set. This happens in the case of interrupts simultaneously occurring. The user should be aware of this and write their code as to not miss a desired interrupt.

GOOD: In the case of inputs 0 and 1 triggering at the same time, triggerValue would be equal to 0x3, and we would set both outputs as desired by the user.

```
interrupt_2 <0x0F>
   int triggerValue = ReadInterruptStatus()
   if( triggerValue & 0x1)
     SetOutput ( 1,0 )
   end if
   if( triggerValue & 0x2)
     SetOutput ( 1,1 )
   end if
end interrupt
```

BAD: If inputs 0 and 1 rise at the same time and triggerValue returns a value of 3, only the case 1 will be executed.

```
interrupt_2 <0x0F>
   int triggerValue = ReadInterruptStatus()
   //if in0 and in1 are triggered simultaneously
   //set the output
   switch( triggerValue )

      case 1:
          SetOutput(0,1)
          break
      case 2:
          SetOutput(1,1)
          break
   end if
end interrupt
```

CHAPTER 6: USING CPL INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Topics include the following:

6.1: Quick Start Guide

The following section is a step by step example of how to open, build and run a CPL program.

Note: It is assumed that the user is connected to a drive; the drive has been set up and tuned; and all safety precautions are in place.

1 Open project:

Click the **Open Projects** button from the toolbar, then choose a project from the *Open Project* dialog box, or select **File→Open Project** in the menu bar.

2 Clean and build

Click the **Clean and Build Main Project** button from the toolbar or select **File→Open Project** in the menu bar.

3 Save to flash

Click the **Save to Flash** button from the toolbar or select **Project→Save to Flash** in the menu bar.

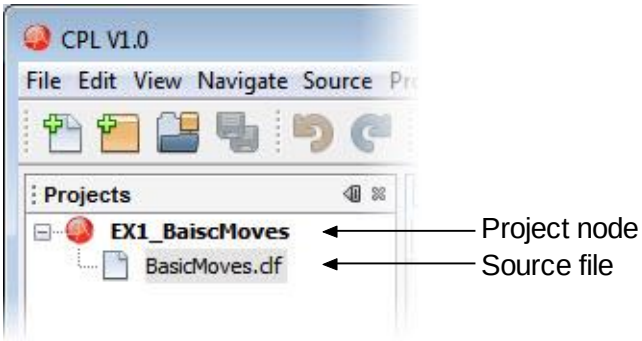
4 Run

Click the **Run CVM Program** button from the toolbar or select **Run→Run CVM Program** in the menu bar.

6.2: Working with Projects

A project is a collection of source files, that will be built into a single CPL program, that will run on the drive. The default location for CPL projects is My Documents\Copley Motions\CPL\Projects. In the projects window a CPL project title will begin with the Copley icon.

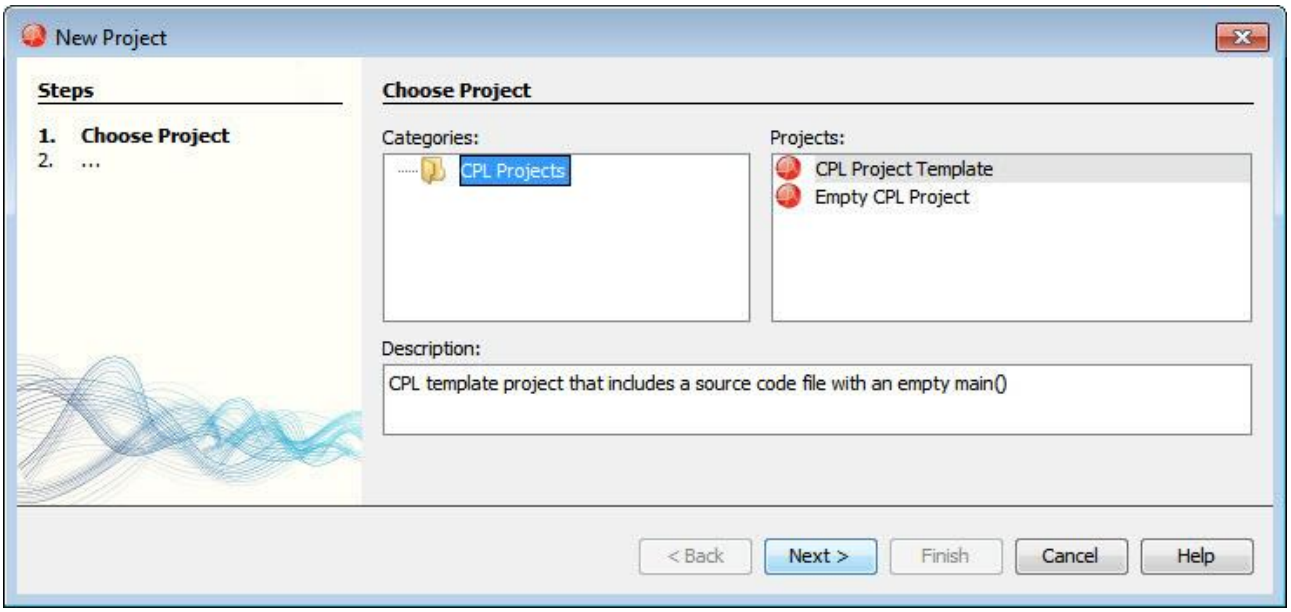
The project tree, and all its associated source files, will show the projects tab as in the example below.



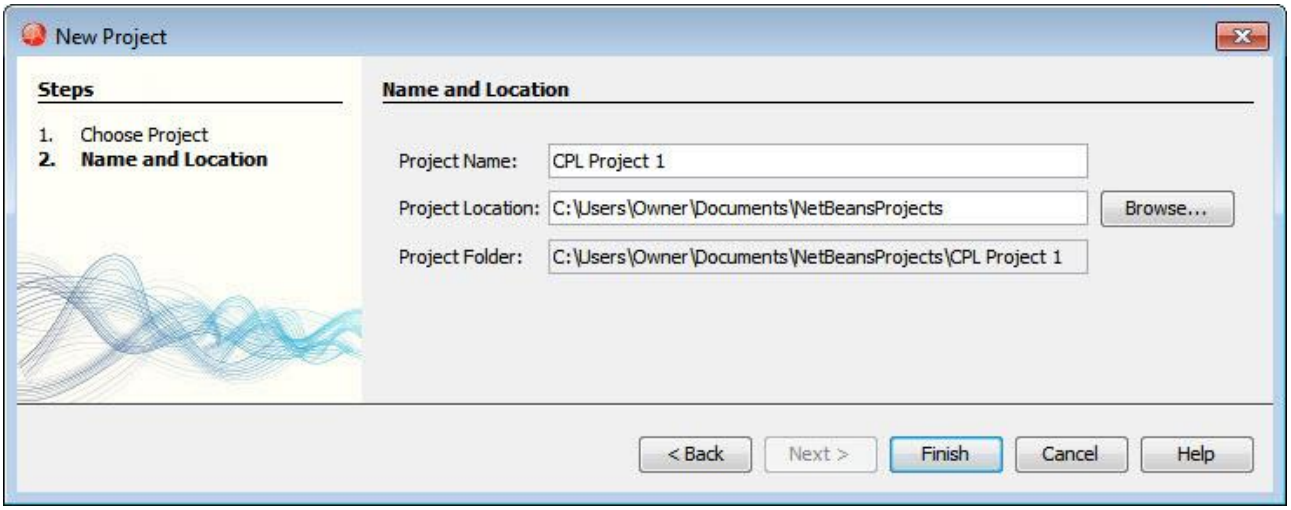
6.2.1: Create a New Project

When creating a new project there are two available types to choose from: *CPL Project Template* and *Empty CPL Project*. The *CPL Project Template* creates a project with a source file that contains a main function (`main()`). The *Empty CPL Project* creates an empty project with no source file.

To create a new project open the New Project wizard by clicking the **New Project** button from the Toolbar, then from the *Projects* screen choose a project type (Empty CPL Project or CPL Project Template) and click **Next**.



Under *Name and Location* enter a Project Name and click **Finish** (The Project Location and Project Folder may also be changed prior to clicking **Finish**).



6.2.2: Open Existing Projects

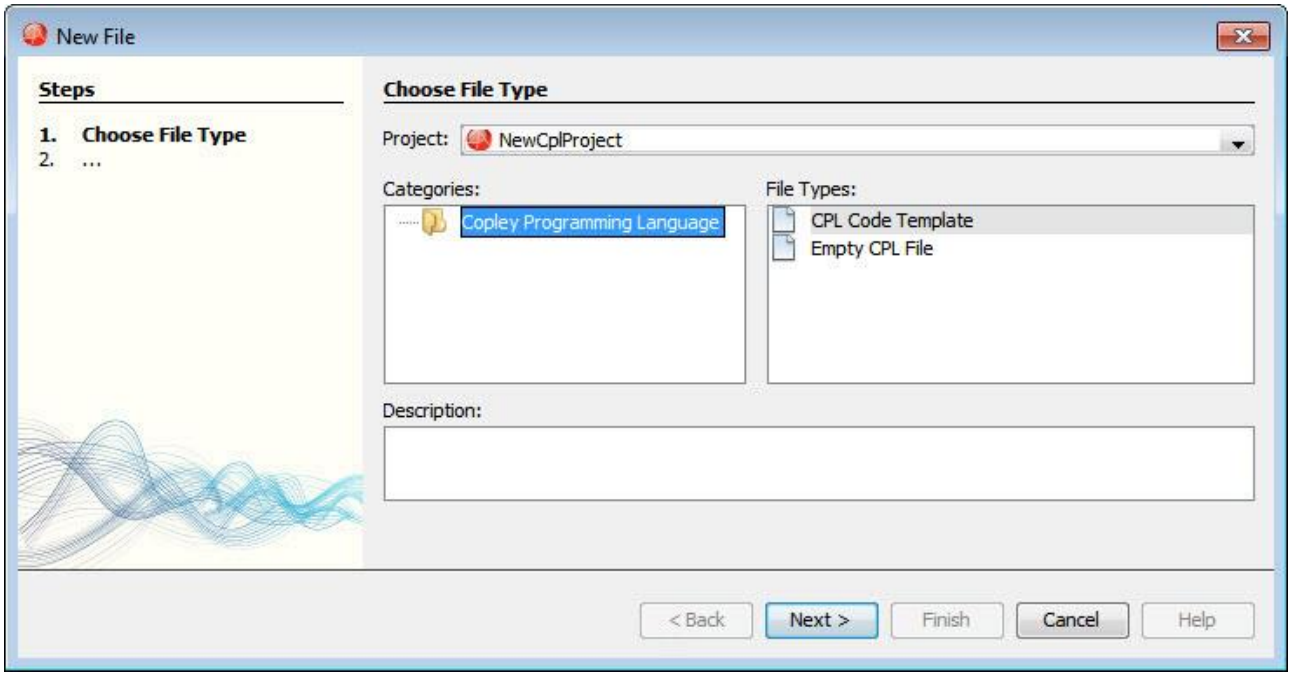
To open an existing project click the **Open Projects** button from the toolbar, then choose a project from the *Open Project* window, or select **File→Open Project** from the menu bar.

6.2.3: Adding Source Files to a Project

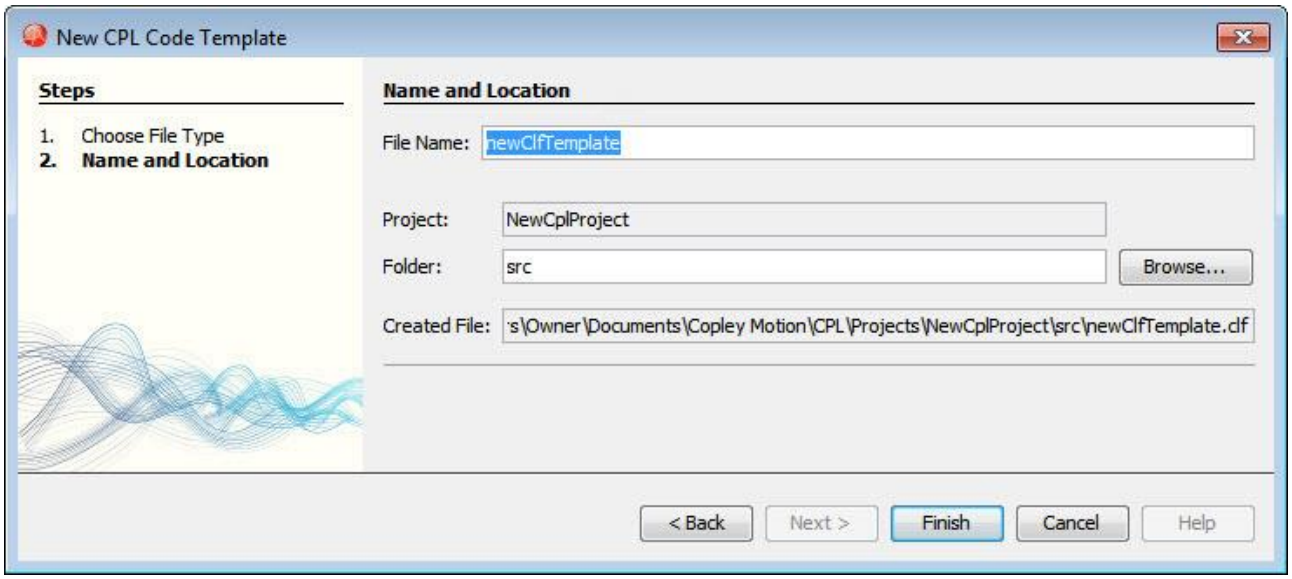
To add a source file to a project:

1 Highlight the project node by clicking on it. Then click the **New File** button from the toolbar, or select **File→New file** from the menu bar. The New File wizard will be displayed**.**

2 In the New File wizard choose a file type from the *File Types* screen. The template file type will create a file with a `main()` method. The empty CPL file creates and empty file. Click **NEXT**.



3 Name the file to be added to the project. Click **Finish**.



4 To open a source file in the editor window double click on the file under the project node.

6.2.4: Deleting Source Files from a Project

To delete source files from a project first select the source file to be deleted from the *Projects* tree, and then either select **Edit→Delete** from the menu bar, or right click the source file and choose **Delete**.

6.2.5: Set Main Project

If multiple projects are open, one must be set as the Main Project. A Main Project node will have **bold** text**.**

To set a main project right click on the project node from the *Projects* tree and choose *Set as Main Project*.

6.2.6: Close Project

To close a project right click on the project node from the *Projects* tree and choose *Close* or select **File→Close Project** from the menu bar.

- Note: Closing a project removes the project from the projects tab but not form the hard drive.

6.2.7: Building Projects 

To build a project, first add source files to a project (see Adding Source Files to a Project, p. 62). If multiple projects are open select one as the main project (See Set Main Project, p. 63). Then, click the **Clean and Build Main Project** button from the toolbar or select **Project→Clean and Build Main Project** in the menu bar.

6.2.8: Saving Program to Flash 

To save a program to flash, first build a project (see, Building Projects p. 63). If multiple projects are open select one as the main project (See Set Main Project, p. 63). Then, either click the **Save to Flash** button from the toolbar or select **Project→Save to Flash** in menu bar.

6.2.9: Running a CPL Program 

After the project has been built and saved to flash, either click the **Run CVM Program** button from the toolbar or select **Run→Run CVM Program** in the menu bar.

- Note: When the program is running breakpoints will be ignored.

6.2.10: Debugging 

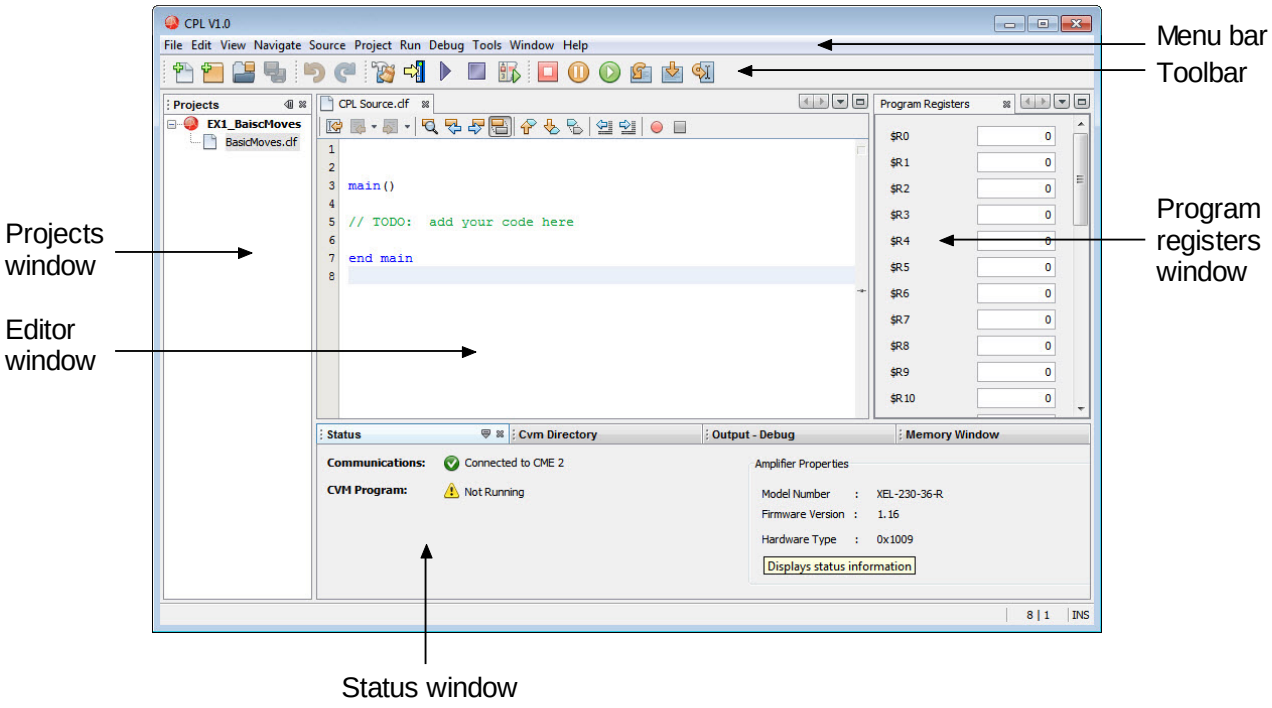
To start debugging a program click the **Debug CVM Program** button from the toolbar or select **Debug→Debug CVM Program** in the menu bar.

For detailed information on using the debugger please see Using the Debugger, p.73.

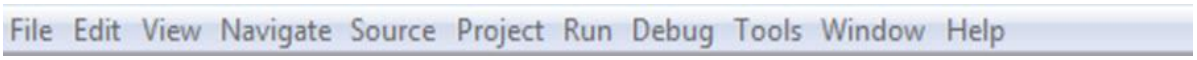
- Note: If there are no breakpoints set in the source code, starting the debugging program is equal to pressing the run button; the program will run without stopping.

6.3: CPL Interface Tour

Some CPL features are called out in the diagram below. Screen details vary depending on drive model and mode selection (In the view below the CVM directory and Output debug windows are immediately available but the Status window has been selected). Details follow in the chapter.



6.3.1: Menu Bar



Name	Selection	Description
File	New Project	Opens a new project wizard and asks the user to choose between a project template or an empty project that contains no source code. See Create a New Project, p. 61.
	New File	Opens a new file wizard and asks for a file type, file name and location. See Adding Source Files to a Project, p. 62.
	Open Project	Opens an existing project. See Open Existing Projects, p. 62.
	Open Recent Project	Opens a recently opened project.
	Close Project	Closes the project.
	Open File	Opens an existing file.
	Open Recent File	Opens a recently opened file.
	Save	Saves a project.
	Save As	Saves a project using a different name or destination.
	Save All	Saves all open files in the editor.
	Page setup	Adjusts page margins, layout, headers, etc.
	Print	Prints the open file in the Editor window.
	Print to HTML	Saves the open file in the Editor window as an HTML file. Optionally opens in browser.
	Exit	Exits CPL IDE.
Edit	Undo	
	Redo	
	Cut	
	Copy	
	Paste	
	Paste Formatted	
	Delete	
	Select All	
	Select Identifier	
	Find Selection	
	Find Next	
	Find Previous	
	Find	
	Replace	
	Find in Projects	
	Replace in Projects	
	Start Macro Recording	
	Stop Macro Recording	

View	Show Line Numbers	
	Show Diff Sidebar	
	Full Screen	
Navigate	Go To Previous Document	
	Last edition Location	
	Back	
	Forward	
	Go To Line	
	Toggle Bookmark	
	Next Bookmark	
	Previous Bookmark	
	Next Error	
	Previous Error	
	Select in Projects	
	Select in files	
Source	Format	
	Remove Trailing Spaces	
	Shift Left	
	Shift Right	
	Move Up	
	Move Down	
	Duplicate Up	
	Duplicate Down	
	Toggle Comment	
	Insert Next Matching Word	
	Insert Previous Matching Word	
	Scan for External Changes	
Project	Clean and Build Main Project	Clears existing build files, then compiles and builds the main project.
	Save to Flash	Saves the compiled code to flash memory.

Run	Run CVM Program	Starts CPL Program execution. CAUTION: Depending on setup configuration and input line state, motion could start immediately.
	Stop CVM Program	Stops CPL Program execution. CAUTION: Any programmed moves in progress will continue until finished.
	Enable CVM Program on Startup	Configures the CPL Program to auto start when the amplifier is powered up or reset. This choice is the default setting.
	Disable CVM Program on Startup	Disables auto start of the CPL Program.
Debug	Debug CVM Program	Starts the debugger.
Tools	Clear CVM Flash	Deletes all files in the CVM Flash memory, including CVM programs, Cam tables, and gain scheduling tables.
	Communications Wizard	Opens the communications setup wizard.
	Options	
Window	Status	Opens a status window.
	Cvm Directory	Opens a CVM Directory window.
	Program Registers	Opens a Program Registers window.
	Projects	Opens a Projects window.
	Files	Opens a Files window.
	Output	Opens the following submenu:  Output  Search Results
	Debugging	Opens the following submenu:  Variables  Breakpoints Internal Registers Memory Viewer
	Editor	Opens an Editor window.
	Close Window	Closes highlighted window.
	Maximize window	Maximizes highlighted window.
	Undock Window	Undocks highlighted window from main window.
	Clone Document	
	Close All Documents	
	Close Other Documents	
	Documents…	
	Reset Windows	

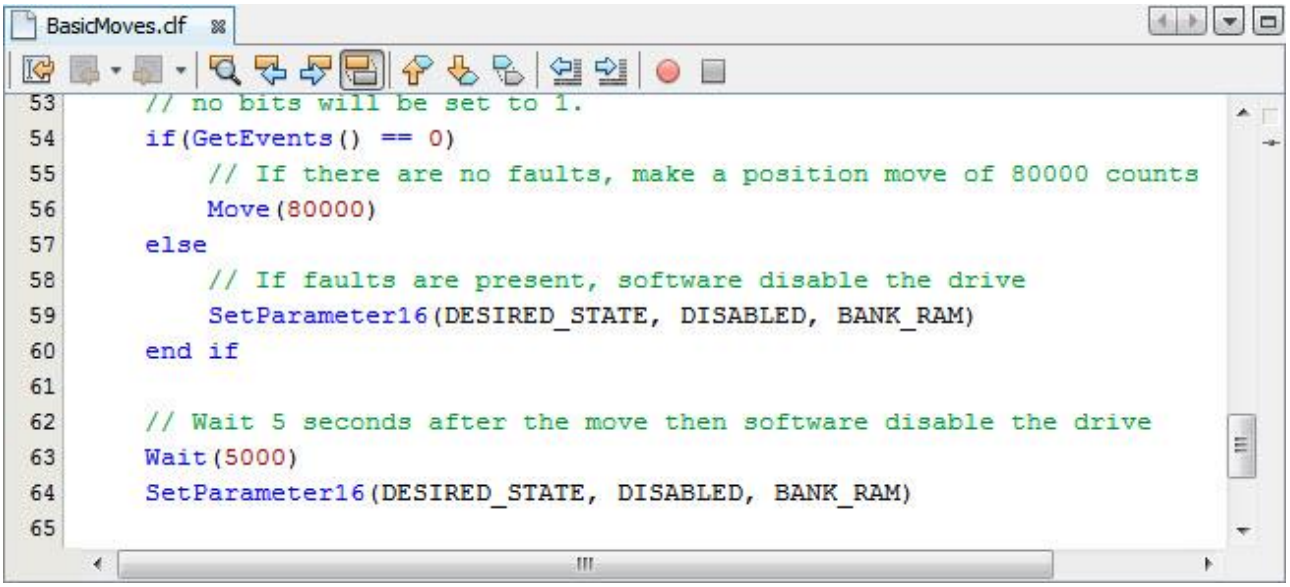
Help	CPL User Guide	Opens this manual.
	CPL Quick Guide	Open the CPL Quick Reference Guide.
	All Documents	Opens the Doc folder in the CPL installation folder. This folder contains all of the related documents that were installed with CPL.
	Downloads Web Page	Opens default web browser with pages from Copley Controls' website.
	Software Web Page	
	View Release Notes	Opens latest CPL release notes in a text viewer.
	About	Displays CPL version information.

6.3.2: Toolbar

Icon	Name	Description
	New File	Creates a new file.
	New Project	Creates a new project.
	Open Project	Opens an existing project.
	Save all files	Saves open files.
	Undo	Undo last edit.
	Redo	Redo last edit.
	Clean and Build	Clears existing build file, then compiles and builds the code from the selected project.
	Save to Flash	Saves the compiled code to flash memory.
	Run CVM Program	Starts CPL Program execution.
	Stop CVM Program	Stops CPL Program execution.
	Debug CVM Program	Starts a debugger session and displays the debugger toolbar buttons.
	Finish Debugger Session	Stops the debugger session.
	Continue	Re-starts or continues the debugger session.
	Step Over	Executes one line of source code at a time. The entire function will be executed in one step.
	Step Into	Executes one line of source code at a time. A function call will be stepped into.

6.3.3: Editor

The Editor window displays program source code.



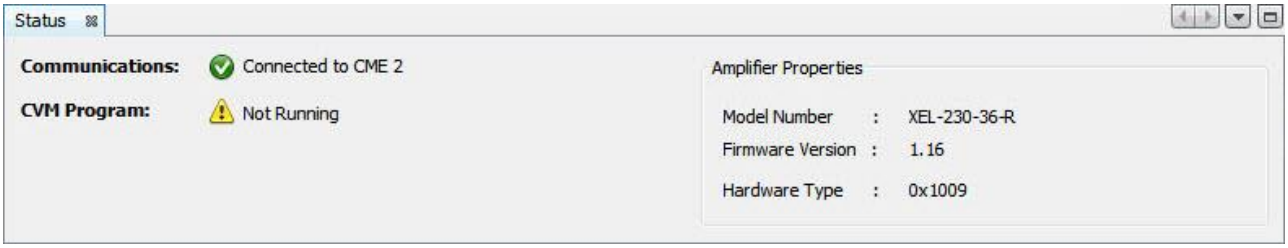
Icon	Name
	Last Edit
	Find Selection
	Find Previous Occurrence
	Find Next Occurrence
	Toggle Highlight Search
	Previous Bookmark
	Next Bookmark
	Toggle Bookmark
	Shift Line Left
	Shift Line Right
	Start Macro Recording
	Stop Macro Recording

6.3.4: Other Windows

Status

The status window describes the present amplifier, communications and CVM program status. In the following window there is no communications device connected, the CVM program is not running, and there is no amplifier connected.

In this window a communications device is connected, the CVM program is not running, and amplifier info is shown.

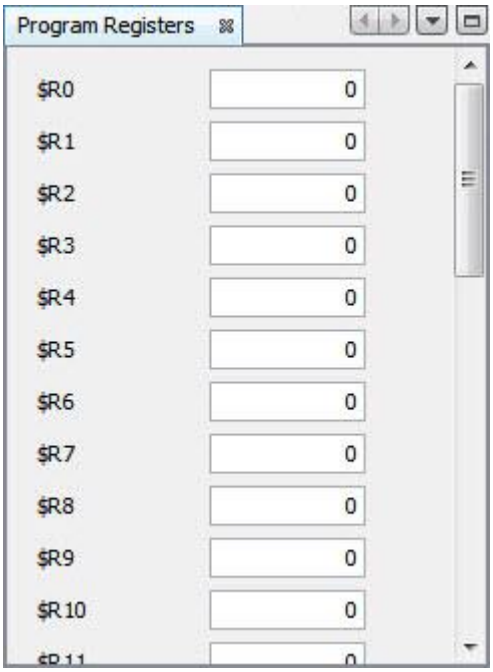


Output

The output window displays all messages related to actions taken in the IDE. The Debug tab displays the messages during a debugging session, such as a breakpoint being hit. The Console tab displays all other messages.

Program Registers

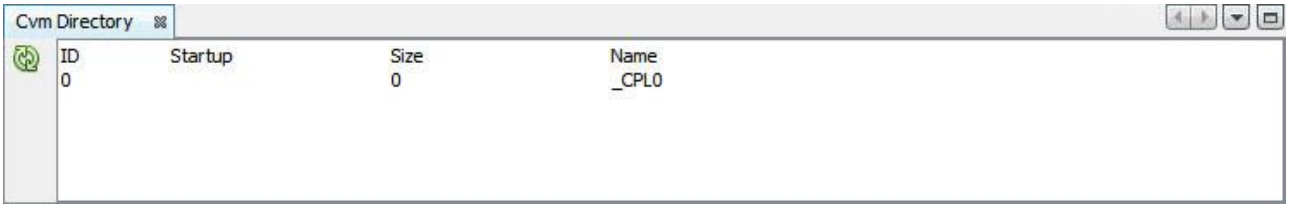
The program registers window displays the current values of CPL's 32 Program registers (R0-R31). Individual register values can be changed manually within the window.



CVM Directory

The CVM directory window displays all the files that have been stored in CVM flash memory. Displayed files include CAM tables and gain scheduling.

To enter new files from the CVM flash memory select the refresh icon in the CVM directory window.



The file number is listed under the ID column. Any file that is set to run on startup will have an asterisk under the Startup column. The size column lists the number of words in the file and the name of the file is listed under the Name column. CPL programs have the name **_CPL0.**

6.4: Using the Debugger

6.4.1: Overview

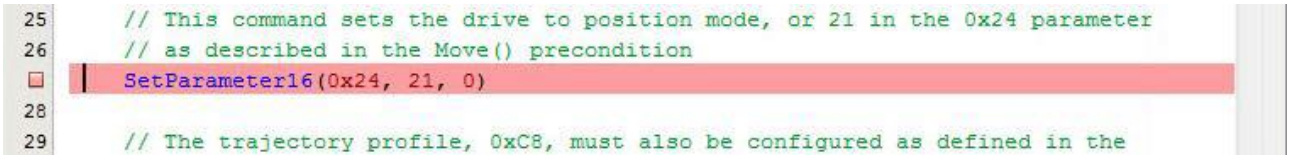
The Debugger allows the testing of a CPL program by inspecting variables, and by stepping through the program one line at a time

6.4.2: Breakpoints

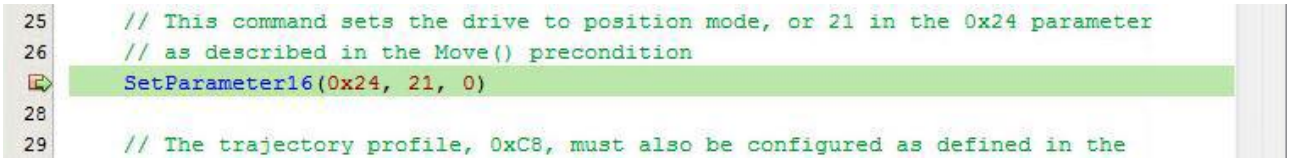
Breakpoints are used to stop execution of a program at a specific line of code. When a breakpoint is reached, actions such as viewing variables or stepping through the program line by line can be performed. A maximum of seven breakpoints can be set. The breakpoints should be set prior to starting a debugging session.

Setting/Clearing Breakpoints:

To set a breakpoint, first clean and build the project, then click on the line number on the left margin of the editor. The line will be highlighted in pink and the breakpoint icon will be placed over the line number. To clear the breakpoint, click on the breakpoint icon in the left margin of the editor.

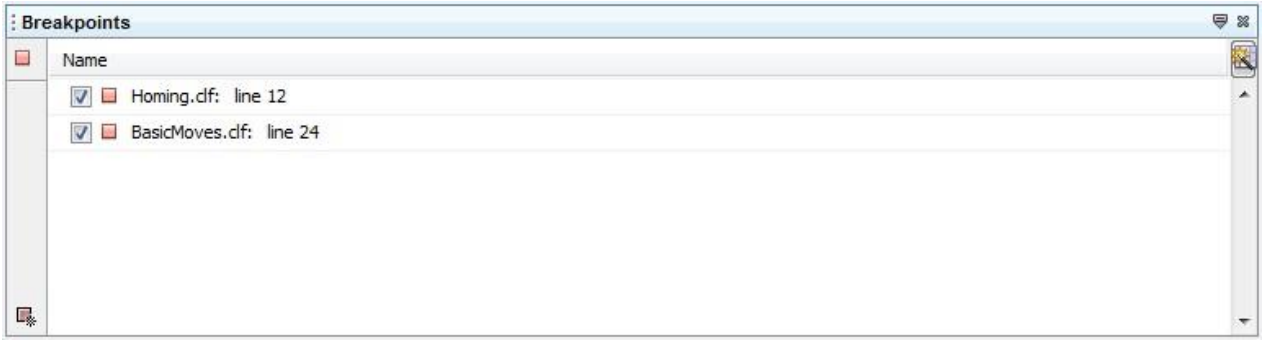


When the program stops at a breakpoint, the line will be highlighted in green and the icon will change to a breakpoint with the program counter . The program counter indicates the next line of code to be executed.



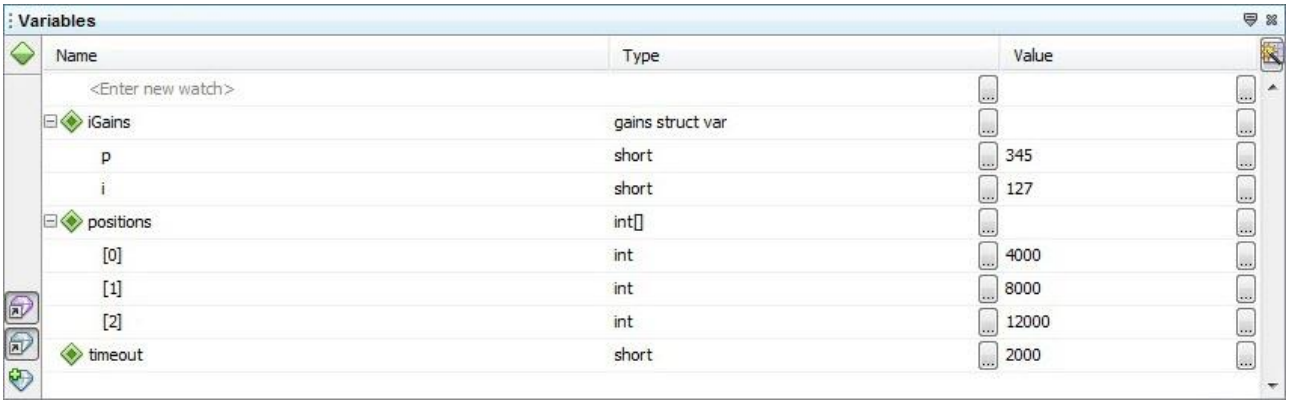
6.4.3: Breakpoints Window

The Breakpoints window displays a list of the breakpoints that have been set. The file name and line number is displayed for each breakpoint. This window is automatically opened when the debugger is started. It may also be opened by selecting **Window→Debugging→Breakpoints** from the menu bar.



6.4.4: Variables Window

The Variables Window is used to inspect the value of variables while debugging. The values are only valid when the CPL program is either stopped at a breakpoint, or while single-stepping. Global variables are always displayed. Local variables are only displayed if they are in scope (see Visibility of variables, p 19). Constants will not be displayed in the Variables Window. This window is automatically opened when the debugger is started. The Variables Window may also be opened by selecting **Window→Debugging→Variables** from the menu bar.



6.4.5: Starting Debugger 

Before starting a debugging a program:

- Clean and build project (see Building Projects, p. 64)
- Save to Flash (see Saving Program to Flash, p. 64)
- Set the appropriate breakpoints in the program (see Breakpoints, p. 73)

To start debugging a program click the **Debug CVM Program** button from the toolbar or select **Debug→Debug CVM Program** in the menu bar.

- Note: If there are no breakpoints set in the source code, starting the debugging program is equal to pressing the run button and the program will run through.

6.4.6: Program Execution

Once the Debugger program has been started, the Debugger buttons in the main toolbar will be displayed. The following describes these buttons.

Continue 

Continue resumes program execution from the current location in a program. Program execution will continue until either a breakpoint is hit or the end of the program is reached.

Step into 

Step into allows execution of one line of source code at a time. If the source code line contains a function call it will step into the function rather line stepping over it. While stepping through code, the next line of code to be executed will be highlighted in green and the program counter icon  will be placed over the line number in the left margin of the editor.

APPENDIX A: RESERVED WORDS

A.1 Reserved Words

break	end	return
case	float	short
const	for	struct
continue	function	switch
default	if	void
else	int	while
elseif	main	
Move	WaitForInput	GetLatchedEvents
VelMovePosMode	WaitForActualPosition	GetTrajStatus
VelMoveVelMode	WaitForLimitedPosition	SetParameter16
CurrentMove	WaitForVelocity	GetParameter16
Home	WaitForVelocityTraj	SetParameter32
Halt	WairForCurrent	GetParameter32
TrajUpdate	GetFaults	SetParameterExt
Wait	ClearFaults	GetParameterExt
WaitMoveDone	GetEvents	SetElecGearRatio
WaitForEvent	GetStickyEvents	ReadInputs
SetOutput		
interrupt_1	interrupt_4	GlobalEnableInterrupts
interrupt_2	interrupt_5	GlobalDisableInterrupts
interrupt_3	i_return	ReadInterruptStatus
SQRT	SIN	ACOS
LOG	COS	ATAN

EXP	TAN	ATAN2
POW	ASIN	FLOOR
CEILING	ABS	

CPL User Guide
16-124981
Rev 00
November 2019