

# **SmartStage Linear System C-Motion Guide**



## SmartStage Linear System C-Motion Guide

Document No. **41-1368**

Revision: **A**

Revision Date: **09/04/2020**


Sheet: **2 of 77**

This page is intentionally left blank.

## Contents


About This Guide .....	8
Additional Reference Guides .....	8
What is included?.....	9
Definitions Used Within This Guide .....	9
What steps are needed to communicate with the stages? .....	10
Point-to-Point Serial.....	10
Multidrop Serial .....	10
CAN .....	10
C-Motion Axis Handles.....	11
Communicating with the Controllers.....	12
Handshake Communications .....	12
Important Registers and Bits .....	12
Getting the stage to move .....	13
Commutation or Phase Initialization .....	13
Basic Motion Commands .....	15
Unit conversion.....	16
Setting and Reading I/O .....	18
Homing Routines.....	19
DOF-5 Homing Routine: .....	19
Items tracked by the homing routine: .....	19
Using the function:.....	20
Advanced Features .....	21
Stall detection and Position Errors .....	21
Breakpoints .....	24
Absolute positioning with Stepper Motors (Open Loop Control).....	25
Electronic Gearing (Step and Direction).....	26
TOP – Trigger on Position.....	28
UDP – User Defined Profile Mode.....	30
Setting the GPIO State .....	34

Error Codes .....	37
Overall Controller Behavior Notes .....	38
Known C-Motion Issues: .....	38
Serial Connection Desynchronization .....	38
Driver Dependency Issues.....	38
Starting Projects in Visual Studio .....	39
Visual Studio 2019 C++ Project Setup .....	39
Visual Studio 2019 C# Project Setup .....	48
Visual Studio 2017 C++ Project Setup .....	57
Visual Studio 2017 C# Project Setup .....	65
Original Source Code.....	74
The SDK .....	74
The DLLs .....	74
Rebuilding the DLLs.....	74
Review/Revision History .....	77

 A division of Invetech	<b>SmartStage Linear System C-Motion Guide</b>		
Document No. <b>41-1368</b>	Revision: <b>A</b>	Revision Date: <b>09/04/2020</b>	Sheet: <b>5 of 77</b>

## List of Tables

Table 1 - Event Status Register .....	12
Table 2 - Activity Status Register .....	13
Table 3 - Motor Type Vs. Control Loops .....	13
Table 4 - Homing Function Returns .....	19

 A division of Invetech	<b>SmartStage Linear System C-Motion Guide</b>		
Document No. <b>41-1368</b>	Revision: <b>A</b>	Revision Date: <b>09/04/2020</b>	Sheet: <b>6 of 77</b>

## List of Figures

Figure 1 - Axis Handle Declaration.....	11
Figure 2 - C-Motion Communications Functions .....	11
Figure 3 - Multiple Device Initialization Calls.....	11
Figure 4 - Additional Features Handle Creation .....	11
Figure 5 - Enabling Control Loops.....	13
Figure 6 – Calculating Phase Counts.....	14
Figure 7- Commutation Function.....	15
Figure 8 - Trajectory Parameters .....	15
Figure 9 - Unit Conversion Functions.....	17
Figure 10 - Setting I/O Commands.....	18
Figure 11 - Signal Sense Register .....	18
Figure 12 – SmartStage Linear Homing Function Declaration .....	19
Figure 13 - Event Status Register .....	21
Figure 14 - Activity Status Register .....	22
Figure 15 - Configuring the Tracking Window Settings .....	23
Figure 16- Resetting Breakpoint Event Status .....	24
Figure 17 - Breakpoint Value Set .....	24
Figure 18 - Setting the Breakpoint.....	24
Figure 19 - Breakpoint Code Example .....	25
Figure 20 - Electronic Gearing Set Gear Ratio .....	26
Figure 21 - Electronic Gearing Set Gear Master .....	26
Figure 22 - Electronic Gearing Function Example.....	27
Figure 23 - Incremental TOP Example .....	29
Figure 24 - Tabled TOP Example .....	29
Figure 25 – Configure UDP Example Functions .....	30
Figure 26 - configureUDPfile() Example.....	32
Figure 27 - Enable UDP Mode Example .....	33
Figure 28 - SetGPIO Example .....	36
Figure 29 - Set GPIO Custom Function Definitions .....	36
Figure 30 - C-Motion Error Codes.....	37
Figure 31 - VS2019 C++ New Project .....	39
Figure 32 - VS2019 Adding Header Files.....	40
Figure 33 - VS2019 Project Properties .....	41
Figure 34 - VS2019 VC++ Directories .....	42
Figure 35 - VS2019 Excluding Uninstalled Dependencies.....	43
Figure 36 - VS2019 Rebuild.....	44
Figure 37 - VS2019 Run Debugging.....	44
Figure 38 - VS2019 Output Directory.....	45
Figure 39 - VS2019 SAFEH Warning .....	46
Figure 40 - VS2019 Missing Driver Error.....	46
Figure 41 - VS2019 Linker Error .....	47
Figure 42 - VS2019 C# Connection Initialization.....	48
Figure 43 - VS2019 New Project C# .....	49
Figure 44 - VS2019 DLL Location.....	50
Figure 45 - VS2019 Add DLL to Project .....	51
Figure 46 - VS2019 Choose Solution Platform .....	52
Figure 47 - VS2019 C# Build Directory .....	53
Figure 48 - VS2019 Debugging C#.....	54

Figure 49 - VS2019 C# Output Directory Verification .....	55
Figure 50 - VS2019 HRESULT: 0x8007000B.....	56
Figure 51 - VS2017 C++ New Project .....	57
Figure 52 - VS2017 Project Properties .....	58
Figure 53 - VS2017 VC++ Directories .....	59
Figure 54 - VS2017 Removing Uninstalled Dependencies .....	60
Figure 55 - VS2017 Rebuild .....	61
Figure 56 - VS2017 Run Debugging.....	61
Figure 57 - VS2017 Output Directory.....	62
Figure 58 - VS2017 SAFESEH Warning .....	63
Figure 59 - VS2017 Missing Driver Error.....	63
Figure 60 - VS2017 Linker Error .....	64
Figure 61 - VS2017 C# Connection Initialization.....	65
Figure 62 - VS2017 New Project C# .....	66
Figure 63 - VS2017 DLL Location.....	67
Figure 64 - VS2017 Add DLL to Project .....	68
Figure 65 - VS2017 Choose Solution Platform .....	69
Figure 66 - VS2017 C# Build Directory .....	70
Figure 67 - VS2017 Debugging C#.....	71
Figure 68 - VS2017 C# Output Directory Verification .....	72
Figure 69 - VS2017 HRESULT: 0x8007000B.....	73
Figure 70 - DLL File Structure .....	75
Figure 71 - Visual Studio Retargeting Solution .....	75
Figure 72 - Build Conditions.....	75


## ***About This Guide***

This document should serve as a guide to the included documentation, and where to start looking through the resources available.

### **Additional Reference Guides**

- 1) 41-1207 Word Structure App Note
- 2) Magellan Motion Control IC Programmer's Command Reference
- 3) Magellan User-Defined Profile Mode Documentation rev1\_1



	<h1>SmartStage Linear System C-Motion Guide</h1>		
Document No. <b>41-1368</b>	Revision: <b>A</b>	Revision Date: <b>09/04/2020</b>	Sheet: <b>9 of 77</b>

## What is included?

The C-Motion SDK and DLLs offer a command formatting API for communicating with Dover stages and controllers. The library offers support for several devices and protocols, offering a variety of communications options with the stages.

Demo code and examples are written using the libraries as well, since they offer additional readability and clarity as to the commands sent to the controllers. For information on the command structure that is handled by the libraries, see the reference documents in Additional Reference Guides, particularly 41-1207 Word Structure App Note contains the formatting information for each protocol used.

The libraries also offer enums and error returns for each function call, making error handling far easier to implement into a larger program.

The code included with these examples is from a subset of the larger original code base that supports additional hardware and communications methods. See Original Source Code for more information on using the unmodified original source code.

## Definitions Used Within This Guide

This is a list of some of the terminology and definitions used within the guide to provide a better understanding and common language.

- 1) Host, host machine, host controller
  - a. This refers to the computer or external system that sends commands to the controller. This coordinates motion within the larger system program and handles telling the stage where to be and when.
- 2) Stage, controller, stage controller
  - a. This refers to the Dover hardware SmartStage linear system and is the Dover hardware. These devices are slave devices, and respond to a host controller or host machine that runs the larger system program.
- 3) USB Stick
  - a. Dover stages with built-in controllers should come with a USB storage device that has this guide, the guides referenced in Additional Reference Guides, example code, and the SDK and DLLs for C-Motion.
  - b. The code used within figures in this guide are pulling demo code from the included examples on these USB sticks, and the full examples can be found there. Some examples have additional comments in the code.
  - c. Resource locations
    - i. 36105-01\C-Motion

### What steps are needed to communicate with the stages?

Several different protocols are handled automatically through C-Motion so that the host does not have to format any commands sent to the controller. Because there are several protocols, it is important that the host configure C-Motion correctly depending on what the controller expects to receive.

#### Point-to-Point Serial

Point-to-Point serial communications is used with a single device connected at a time, generally over RS232. This communications packet formatting style ignores the address bits and assumes that those bits will be all zeros. This protocol cannot be used with multiple devices since they will all respond at the same time and cause communications errors. See 41-1207 Word Structure App Note for more information on the packet structure sent. C-Motion handles all formatting requirements and packet structure.

#### Multidrop Serial

This serial communications method is used with multiple devices, generally over RS485. This protocol requires a set address to independently connect to different controllers on the same network. The address is used when calculating the checksum. See 41-1207 Word Structure App Note for more information on the packet structure sent. C-Motion handles all formatting requirements and packet structure.

#### CAN

This protocol requires CAN addressing, and a Node ID for addressing controllers individually. There is one supported protocol. For more information on the packet structure, refer to 41-1207 Word Structure App Note. Note that the drivers used within the libraries are 32-bit.

### C-Motion Axis Handles

C-Motion uses an “Axis Handle” to configure communications settings and port information for the supported device drivers. At the beginning of the user code, a new handle should be initialized per Figure 1 - Axis Handle Declaration. Note that these commands are using the enum from C-Motion to initialize axis handles for axis 0, the primary setup on the controller.

```
PMDAxisHandle hAxis1;
```

Figure 1 - Axis Handle Declaration

Then communications settings should be initialized using one of the C-Motion initialization functions per Figure 2 - C-Motion Communications Functions.

```
PMDSetupAxisInterface_Serial(phAxis1, PMDAxis1, serialmode, address, port, baud);
PMDSetupAxisInterface_CAN(phAxis1, PMDAxis1, baud, address);
```

Figure 2 - C-Motion Communications Functions

Once the initial axis handle has been created any subsequent devices connected will need a separate axis handle. It is necessary to copy the axis interface over for any additional axes that will be connected. This should be done using the calls per Figure 3 - Multiple Device Initialization Calls.

```
PMDCreateMultiDropHandle(phAxis2, phAxis1, PMDAxis1, nAddress2);
PMDCreateMultiDropHandle_CAN(phAxis2, phAxis1, PMDAxis1, 1);
```

Figure 3 - Multiple Device Initialization Calls

Once an axis handle is set up for each axis, the user must determine the axes that any additional features are required on for configuration. Some features of the controllers are accessed by addressing “PMDAxis2” in the command formatting, a virtual second axis that allows configuration of the following features:

- Trigger on Position (TOP)
- User Defined Profile Mode (UDP)
- Auxiliary encoders
- Electronic Gearing Mode
- Additional I/O

In order to initialize an additional handle for addressing these features on a single controller, use the command per Figure 4 - Additional Features Handle Creation that copies over the communications settings from another axis handle. This command does not change based on the protocol that is being used.

```
PMDCopyAxisInterface(phAxis2, phAxis1, PMDAxis2);
```

Figure 4 - Additional Features Handle Creation

## Communicating with the Controllers

All of the controllers addressable using C-Motion respond to requests from the host machine, and handle trajectories and certain motion events in the background, but all other action must be handled on the host side. Some of the included demo and example code relies heavily on the host machine handling the events generated by the controller. Homing routines are a particular example where the controller handles the motion itself, but the host is still ultimately making all the decisions and configuring what the controller should be watching for.

### Handshake Communications

Handshake communications is a term used to describe the way these controllers operate. They only respond to requests from the host, and so always need a “handshake” to give any information. This means that the host must poll for status information from the controller if the host depends on a motion completing, or some other status information is required from the controller.

### Important Registers and Bits

Two main registers are used when controlling motions and verifying the condition the stage controller is in. The Activity Status Register and the Event Status Register. See Table 1 - Event Status Register and Table 2 - Activity Status Register for the more important status bits that can be monitored, and an explanation of each. See Magellan Motion Control IC Programmer’s Command Reference for a full reference to the status bits.

*Table 1 - Event Status Register*

Bit Number	Bit Name	Description
0	Motion Complete	The conditions that determine motion complete are described in Stall detection and Position Errors, the bit goes high when they are met
2	Breakpoint 1	Breakpoints are described in Advanced Features, Breakpoints bit goes high when the set condition is met
3	Capture Received	The capture mechanism allows high-speed and short duration signal capture used with index marks, the DOF-5 uses the home signal, the DMCM uses index
4	Motion Error	The conditions that determine motion error are described in Stall detection and Position Errors, the bit goes high when they are met
5	Positive Limit	This bit goes high with the measured signal
6	Negative Limit	This bit goes high with the measured signal
7	Instruction Error	Set high when a command sent would trigger an error or when a communication error occurs
11	Commutation Error	Set high when a commutation error occurs, generally when phase initialization fails, only relevant to 3 phase motors
12	Current Foldback	Current foldback is an error state triggered by outputting current greater than the max continuous specified current for too long a time (prevents damage to motor)
14	Breakpoint 2	Breakpoints are described in Advanced Features, Breakpoints the bit goes high when the set condition is met

Table 2 - Activity Status Register

Bit Number	Bit Name	Description
0	Phasing Initialized	Holds whether commutation or phasing has been initialized and completed correctly on the controller
2	Tracking	The tracking window (described in Stall detection and Position Errors) monitors the position error and goes high when a threshold is exceeded
7	Axis Settled	The conditions that determine what is settled are reviewed in Stall detection and Position Errors, this bit goes high when they are met
11	In Positive Limit	This bit goes high with the measured signal, and does not latch
12	In Negative Limit	This bit goes high with the measured signal, and does not latch

## Getting the stage to move

Before the stage can be moved, it must be in the correct operating mode, this depends on the motor type being controlled. See Table 3 - Motor Type Vs. Control Loops for a table of supported/fixed motor types for Dover stages, and the control loops to enable for each.

Table 3 - Motor Type Vs. Control Loops

Stage / Controller	Motor Type	Control Loops Needed	Phase Initialization Required?	Motor Current
Smart Stage	3 Phase	0x37 (Enable all: Axis Enabled, Motor Output, Current Loop, Position Loop, Trajectory Loop)	Yes	Controlled by position loop

Once the appropriate control loops have been enabled by sending the command in Figure 5 - Enabling Control Loops most stages will begin to hold position.

```
PMDSetOperatingMode(phAxis, 0x37);
```

Figure 5 - Enabling Control Loops

## Commutation or Phase Initialization

For all 3 phase motors, the encoder and motor must be synchronized, so that the controller knows where in travel to power which motor coil. This is done through the process of commutation, or phase initialization.

How it works: commutation or phase initialization engages one of the motor coils, and measures the position that the stage settles to. It then engages a different motor coil, and measures how far the stage moves from the previous position. From this, the controller can detect the direction moved, and where the motor coils are in relation to the encoder position so it can properly engage the three motor coils at the correct positions.

Errors that can result from poor commutation, or not commutating:

- Motion error event status
  - This is a common result of not commutating, since the motor engages the wrong coil first, which is out of phase with the magnets mounted in the stage, causing the stage to move in an unpredictable manner, often exceeding the tracking error limit.
- High current draw

- When the motor is not commutated, the stage may be able to maintain position if it happens to start close to a magnetic pole, but the motor will need to use higher current in order to maintain position properly.

Common problems encountered while commutating:

- Power cycled
  - The commutation process must be completed every time the stage is power cycled.
  - The process must also be re-run if the maximum velocity is exceeded, as encoder counts may have been lost – at 1.25nm this is 31.25mm/s.
- Restricted motion
  - The stage must be able to move freely when commutating in order to properly measure the motion induced by the commutation process.
  - If the stage is unbalanced (has external forces acting on it) and is biased to one of the mechanical limits, then the stage may not be able to travel far enough to properly complete the process.
- Not enough time to execute the process completely
  - These functions use a default time of 3 seconds to engage each of the 2 motor coils used, so a total of 6+ seconds should be allowed for the process to complete.
- Too low a motor output
  - If the stage is unbalanced (biased to move in one direction by external forces) or has a very large payload then the stage may need a higher motor output to drive it through one complete electrical cycle.
  - The default output is 10% of the set max, but this can be increased to help the motor locate the electrical poles. Increases as high as 30-40% are generally safe. Contact Dover if outputs higher than 40% are needed.
- Incorrect configuration
  - Reach out to Dover if the process cannot complete, things to check are the phase counts and phase prescaler, these should satisfy the following equation where the prescaler is 1, 64, 128 or 256. The electrical cycle is 25.4mm. The encoder resolution should be in counts/mm, so 5nm encoder has 200,000 counts/mm.

$$\text{Phase Counts} = \frac{\text{Encoder Resolution} * \text{Electrical Cycle}}{\text{Prescaler}}$$

Figure 6 – Calculating Phase Counts

Figure 7- Commutation Function shows an example function for commutation that has a simple check at the end to verify that the process completed successfully.

```
void commutation(PMDAxisHandle * phAxis, PMDuint16 commMethod, PMDint16 commTime,
PMDint16 motorOutPercent) {

    //comm time to cycles conversion:
    int commTimecycles = commTime * 1000 / 20.4;

    printf("Initializing Phase\n");
    PMDSetOperatingMode(phAxis, 0x0001);

    //delay a fixed 100ms here
    int end_time, current_time;
    current_time = clock();
    end_time = current_time + 100;
    while ((current_time < end_time) || (current_time < 5000))
        current_time = clock();
}
```

```

PMDSetCommutationMode(phAxis, 0);
PMDSetPhaseInitializeMode(phAxis, commMethod);
PMDSetPhaseCorrectionMode(phAxis, 0);
PMDSetPhaseInitializeTime(phAxis, commTimecycles);
PMDSetMotorCommand(phAxis, motorOutPercent * 32768 / 100);
//reset errors
PMDResetEventStatus(phAxis, 0xa000);
PMDResetEventStatus(phAxis, 0xffff);
PMDSetOperatingMode(phAxis, 0x0003);
//start the process of commutation, do not interfere with this process while it
is running, or the controller will not finish and will not trip any errors
PMDSetPhaseInitializeTime(phAxis, commTimecycles);
PMDResetEventStatus(phAxis, 0xf7ff);
PMDSetMotorCommand(phAxis, motorOutPercent * 32768 / 100);
PMDInitializePhase(phAxis);

//delay longer than the commutation operation is scheduled to take here
current_time = clock();
end_time = current_time + commTime * 2 + 1000;
while ((current_time < end_time) || (current_time < 5000))
    current_time = clock();

PMDSetMotorCommand(phAxis, 0);

PMDuint16 phinit = 0;
PMDGetActivityStatus(phAxis, &phinit);
if (phinit && 0x0001) {
    //phasing initialized successfully
    printf("Completed OK\n");
}
else {
    //phasing did not complete correctly - be sure to check that the physical
motion of the stage is not interfered with
    printf("Failed\n");
}
}

```

Figure 7- Commutation Function

## Basic Motion Commands

Once the control loops are enabled, and the stage is holding position, it is ready to accept motion commands. All controllers use absolute positioning for all commands sent, and the controllers only deal with encoder counts or output counts. See Figure 8 - Trajectory Parameters for the parameters to send. Dover demo code that is distributed with the controllers includes conversion formulas, and the conversion formulas can be found in the Magellan Motion Control IC Programmer's Command Reference.

```

PMDSetProfileMode(phAxis, PMDProfileModeTrapezoidal);
PMDSetJerk(phAxis, 1234);
PMDSetAcceleration(phAxis, 1234);
PMDSetDeceleration(phAxis, 1234);
PMDSetVelocity(phAxis, 1234);
PMDSetPosition(phAxis, 1234);
PMDUpdate(phAxis);

```

Figure 8 - Trajectory Parameters

Be aware that until a home position is defined, the stage may not be at 0, as that is wherever the controller happened to power on from. Caution should be exercised before homing the stage.

All commands issued are in absolute coordinates from the defined zero position of the controller, relative motions must read the current commanded position and add to it.

### Unit conversion

All commands issued to the controller must be sent in counts, the native units for the controllers. Real world unit conversion must happen at the host level. See Figure 9 - Unit Conversion Functions for the functions used with basic unit conversion for motion commands.

The functions referenced here use the encoder resolution in counts/mm to convert real world units in mm to counts. For stages with 5nm, that should be 200,000 counts/mm, 1.25nm resolution should be 800,000 counts/mm, etc. Time is measured in servo cycles, for all current controllers the servo cycle is 51.2 $\mu$ s, or 19531.25 cycles/second. Verify this conversion with the stage controller datasheet.

Current conversion must use the 0.611mA/count value for the controller, this should be found in the SmartStage user manual to ensure the correct value. This conversion is a direct multiplication of the value by the counts measured.

Time conversions are all measured in servo cycles, which again are 51.2 $\mu$ s for all controllers as of writing, verify this with the datasheet for the stage controller being used.



```
//converts mm/s/s/s to PMD units
PMDint32 CalcJerk(float mmpsecpsecsec_jerk, PMDuint32 enc_res) {
    /*
    * Jerk
    * 1 mm      Encoder Ticks   |      sec      | 3
    * ----- * ----- * | ----- | * 4294967296 = PMD Jerk command
    * sec              mm      | 19531.25 cycles |
    */

    return (PMDint32)(mmpsecpsecsec_jerk * enc_res / 19531.25 / 19531.25 / 19531.25
* 4294967296);
}

//converts mm/s/s to PMD units
PMDint32 CalcAccel(float mmpsecpsec_accel, PMDuint32 enc_res) {
    /*
    * Acceleration / Deceleration
    * 1 mm      Encoder Ticks   |      sec      | 2
    * ----- * ----- * | ----- | * 65536 = PMD Acceleration
command
    * sec              mm      | 19531.25 cycles |
    */

    return (PMDint32)(mmpsecpsec_accel * enc_res / 19531.25 / 19531.25 * 65536);
}

//converts mm/s to PMD units
PMDint32 CalcVel(float mmpsec_vel, PMDuint32 enc_res) {
    /*
    * Velocity
    * 1 mm      Encoder Ticks   |      sec      |
    * ----- * ----- * | ----- | * 65536 = PMD Veocity command
    * sec              mm      | 19531.25 cycles |
    */

    return (PMDint32)(mmpsec_vel * enc_res / 19531.25 * 65536);
}

//converts mm to PMD counts
PMDint32 CalcPos(float mm_pos, PMDuint32 enc_res) {
    return (PMDint32)(mm_pos * enc_res);
}
```

Figure 9 - Unit Conversion Functions

## Setting and Reading I/O

The controllers use a register to control and set the state of the input inversion, and output states. The hardware connected to any actual outputs on the stage should be verified in the appropriate user manual. SmartStage linear systems have multiplexed I/O, so the appropriate state must be set to use certain I/O features.

It is important not to invert the state of any inputs that are otherwise already configured. This could potentially result in motion errors or other configuration issues. When setting the state of the I/O, two commands should be issued, one to read the current state, and one that modifies the read state with updates to just the bits that need to be changed.

Figure 10 - Setting I/O Commands has the two commands used to read and set the register, and Figure 11 - Signal Sense Register has the available bits that can be set. Note that some features are on axis2, things like the pulse and direction inputs and auxiliary encoder inputs. Care must be taken not to invert the state of the limits, encoder A/B on axis 1, and the motor direction on axis 1.

```
PMDGetSignalSense(phAxis, &signalsense);
PMDSetSignalSense(phAxis, signalsense);
```

Figure 10 - Setting I/O Commands

Arguments	Name	Instance	Encoding	
	<i>axis</i>	<i>Axis1</i>	0	
		<i>Axis2</i>	1	
		<i>Axis3</i>	2	
		<i>Axis4</i>	3	
		<b>Indicator</b>	<b>Encoding</b>	<b>Bit Number</b>
	<i>sense</i>	<i>EncoderA</i>	0001h	0
		<i>EncoderB</i>	0002h	1
		<i>Encoder Index</i>	0004h	2
		<i>Capture Input</i>	0008h	3
		<i>Positive Limit</i>	0010h	4
		<i>Negative Limit</i>	0020h	5
		<i>AxisIn</i>	0040h	6
		<i>HallA</i>	0080h	7
		<i>HallB</i>	0100h	8
		<i>HallC</i>	0200h	9
		<i>AxisOut</i>	0400h	10
		<i>Step Output/SPI Enable</i>	0800h	11
		<i>Motor Direction</i>	1000h	12
		— (Reserved)		13–15

Figure 11 - Signal Sense Register

## Homing Routines

The controllers do not support any pre-programmed functionality, so homing operations must be handled by the host. Dover has several homing routines, all functioning similarly, checking initial conditions and beginning motion with breakpoints set to stop motion once expected inputs are received. The host ends up polling for the breakpoint status to move to the next step of homing. An explanation of one such routine is included in this document, the source code for each routine is distributed with the controllers on their USB sticks.

### SmartStage Linear Homing Routine:

The provided example homing routine for the DOF-5 takes two inputs specifically (see Figure 12 – SmartStage Linear Homing Function Declaration) axis handle for the controller and the encoder resolution for the DOF being addressed. The return is an int with states captured in Table 4 - Homing Function Returns.

```
int DOHomeV2(PMDAxisHandle * phAxis, PMDuint32 enc_res);
```

Figure 12 – SmartStage Linear Homing Function Declaration

Table 4 - Homing Function Returns

Function Return	Description	Possible Causes
0	No error	Homing completed successfully
1	Operating mode not set	The homing function does not handle setup of the stage, make sure to enable everything prior to calling the function
2	Error condition	There was an error state in the event status register prior to starting, be sure that the stage does not have any unhandled errors before calling the function
3	Timeout	The timeout period (default hardcoded 10 seconds) was exceeded looking for the home signal, something interfered with motion, the motor output may have been disabled
4	Error moving	Something caused a motion error while homing, check for obstructions to motion or that the stage is not unstable

The axis handle is whatever handle the user has initialized for the axis when connecting to the stage using C-Motion. The encoder resolution is the number of counts/mm that the SmartStage linear encoder is configured for. The most common resolution is 5nm (200,000 counts/mm).

This function can be called to move the stage to the 'home' position, at the center of travel, on the positive side of the home signal transition point. The routine always approaches this point at a fixed speed (the default is 1mm/s) from the top of travel, which should give a consistent homing position.

### Items tracked by the homing routine:

- SignalSense
  - The signal sense register is responsible for inverting/setting the read state for the controller input signals. If this register was inverted for the home signal, it could affect the logic needed for homing.
- Motion error, instruction error, overtemperature fault, drive exception, commutation error, current foldback

- Motion error – the position error limit was exceeded – make sure that the error limit is not set too tightly, and make sure the stage is normally stable, not oscillating and is unobstructed by obstacles
- Instruction error – there was a communications problem – make sure the hardware used to communicate with the stage controller is working reliably and that all library commands are issuing correctly.
- Current Foldback – the stage was drawing higher current than is safe for too long a period of time – make sure the stage is normally stable and operating correctly, no physical obstructions.

### Using the function:

This function returns error codes depending on what happens while running. It checks for most error states while running, and any pre-existing error states, and has a timeout using standard C calls when waiting to detect the home switch. This is a blocking function.

- Before running the function:
  - Make sure the operating mode of the controller has been set to 0x37 to enable all the control loops.
  - Make sure pre-existing errors are cleared using ResetEventStatus.
- While running the function:
  - Make sure there are no physical obstructions to the motion of the stage.
  - Make sure the stage can reach the home flag location (the center of stage travel)
- After running the function:
  - Check the return value of the function to determine if there were any errors.
  - To determine more detail about the error received, use GetEventStatus
  - The stage should be configured for trapezoidal motion profiles with default speed and acceleration. These can (and should) be overridden for future motions.

## Advanced Features

### Stall detection and Position Errors

The MC58113 controller has several settings that can be used for motion tracking and programmatically detecting when there are deviations in position from the generated trajectory. The main settings to be aware of are discussed below.

The Event Status Register:

The event status register holds the event statuses for the controller and contains more information that is discussed here. The two bits that matter in this register for motion are highlighted below in Figure 13 - Event Status Register

- Motion Complete
  - This bit is set to 1 once the motion is considered to be complete
  - This bit is set based off the settings in SetMotionCompleteMode to either commanded (purely trajectory based) or actual (based off the physical stage position, and the settling settings)
- Motion Error
  - This bit is set when the position error limit is exceeded, the controller will go into the state specified by SetEventAction

Name	Bit(s)	Description
Motion Complete	0	Set to 1 when motion has completed. <b>SetMotionCompleteMode</b> determines if this bit is based on the trajectory generator position or the encoder position.
Wrap-around	1	Set to 1 when the actual (encoder) position has wrapped from maximum allowed position to minimum, or vice versa.
Breakpoint 1	2	Set to 1 when breakpoint 1 has been triggered.
Capture Received	3	Set to 1 when a position capture has occurred.
Motion Error	4	Set to 1 when a motion error has occurred.
Positive Limit	5	Set to 1 when the axis has entered a positive limit switch.
Negative Limit	6	Set to 1 when the axis has entered a negative limit switch.
Instruction Error	7	Set to 1 when an instruction error has occurred.
Disable	8	Set to 1 when "disable" due to user /Enable line has occurred.
Overtemperature Fault	9	Set to 1 when overtemperature condition has occurred.
Drive Exception	10	An drive event occurred causing output to be disabled. This bit is used on ION products to indicate a bus voltage fault, and with an attached Atlas amplifier to indicate any disabling drive event.
Commutation error	11	Set to 1 when a commutation error has occurred.
Current Foldback	12	Set to 1 when current foldback has occurred.
—	13	Reserved; not used; may be 0 or 1.
Breakpoint 2	14	Set to 1 when breakpoint 2 has been triggered.
—	15	Reserved; not used; may be 0 or 1.

Figure 13 - Event Status Register

## Activity Status Register:

The activity Register contains information relevant to the current running actions of the controller. The bits of interest here are, and can be found in Figure 14 - Activity Status Register.

- Tracking (Window)
  - The tracking window is a user-set window that can be used as a pre-position error warning or a notifier that there may be something unexpected happening.
- In-motion
  - This bit is set as long as the controller is moving in a generated profile and is based off the commanded trajectory.

Name	Bit(s)	Description																				
Phasing Initialized	0	Set to 1 if phasing is initialized (brushless DC axes only).																				
At Maximum Velocity	1	Set to 1 when the trajectory is at maximum velocity. This bit is determined by the trajectory generator, not the actual encoder velocity.																				
Tracking	2	Set to 1 when the axis is within the tracking window.																				
Current Profile Mode	3-5	Contains trajectory mode encoded as follows: <table><tr><th>bit 5</th><th>bit 4</th><th>bit 3</th><th>Profile Mode</th></tr><tr><td>0</td><td>0</td><td>0</td><td>Trapezoidal</td></tr><tr><td>0</td><td>0</td><td>1</td><td>Velocity Contouring</td></tr><tr><td>0</td><td>1</td><td>0</td><td>S-curve</td></tr><tr><td>0</td><td>1</td><td>1</td><td>Electronic Gear</td></tr></table>	bit 5	bit 4	bit 3	Profile Mode	0	0	0	Trapezoidal	0	0	1	Velocity Contouring	0	1	0	S-curve	0	1	1	Electronic Gear
bit 5	bit 4	bit 3	Profile Mode																			
0	0	0	Trapezoidal																			
0	0	1	Velocity Contouring																			
0	1	0	S-curve																			
0	1	1	Electronic Gear																			
—	6	Reserved; not used; may be 0 or 1.																				
Axis Settled	7	Set to 1 when the axis is settled.																				
Position Loop Enabled	8	Set to 1 when position loop or trajectory is enabled.																				
Position Capture	9	Set to 1 when a value has been captured by the high speed position capture hardware but has not yet been read.																				
In-motion	10	Set to 1 when the trajectory generator is executing a profile.																				
In Positive Limit	11	Set to 1 when the positive limit switch is active.																				
In Negative Limit	12	Set to 1 when the negative limit switch is active.																				
Profile Segment	13-15	When the profile mode is S-curve, it contains the profile segment number 1-7 while profile is in motion, and contains a value of 0 when the profile is at rest. This field is undefined when using the Trapezoidal and Velocity Contouring profile modes.																				

Figure 14 - Activity Status Register

## User Defined Settings, Flags and Actions:

The user can define the behavior, and when that behavior will occur within the controller based off the settings discussed below. Essentially the intended behavior is that the tracking window is used as an early warning system that there is some following error, and the position error limit is used as the crash or stall detection.

- Error/Warning Settings
  - Position Error Limit
  - Event Action (Specifically for motion errors)
  - Tracking Window
- Motion Complete Settings
  - Settle Time
  - Settle Window
  - Motion Complete Mode (commanded or actual)

Demo Code:

Figure 15 - Configuring the Tracking Window Settings is some C++ demo code using the C-Motion libraries to configure these settings programmatically. Note that these can be saved to NVRAM.

```
//Setting the position error limit and action
PMDSetPositionErrorLimit(&hAxis1, CalcPos(0.01, enc_res));
PMDSetEventAction(&hAxis1, PMDEventActionEventMotionError,
PMDEventActionDisableMotorOutputAndHigherModules);

//Setting the tracking window (early warning for position error)
PMDSetTrackingWindow(&hAxis1, CalcPos(0.005, enc_res));

//Setting the motion complete settings
//generally use the actual position mode, this is based off encoder data instead of the
generated trajectory
PMDSetMotionCompleteMode(&hAxis1, PMDMotionCompleteModeActualPosition);
//settle time is set in servo cycles (51.2us)
float pmdsettletime = 100;//in milliseconds
PMDSetSettleTime(&hAxis1, (PMDuint16)(pmdsettletime/51.2));
//the settling window must be maintained for the settle time for "motion to be
complete"
PMDSetSettleWindow(&hAxis1, CalcPos(0.001, enc_res));
```

Figure 15 - Configuring the Tracking Window Settings

The event status register and activity status register can then be polled for during motion to observe if any unusual behavior is resulting in undesired motion or unpredictable behavior.

### Breakpoints

Breakpoints provide a convenient way to trigger different actions at reasonably precise timings with the controllers. Each controller has two configurable breakpoints. Breakpoints need several commands to be configured to work:

#### 1) Reset the event status

In order to work properly the event status must be cleared for the breakpoint. See Figure 16- Resetting Breakpoint Event Status for the commands to reset the event statuses.

```
PMDResetEventStatus(phAxis1, 0xFFFE);
PMDResetEventStatus(phAxis1, 0xBFFF);
```

Figure 16- Resetting Breakpoint Event Status

#### 2) Set the breakpoint value

Breakpoint values are set depending on what triggers are being used by the breakpoint. Some are very straightforward like position-based triggers, where the value set is the position (always be sure to convert the units to counts, or the base controller units).

Breakpoints that trigger off signals or other registers need to be configured with a selection mask and sense mask. This allows the user to control which signals will cause the breakpoint to trigger and what is considered “active” for a trigger. In Figure 17 - Breakpoint Value Set, a value of 0x00400040 is used.

- The first 4 bytes (0x0040) select the Axis In signal
- The second 4 bytes (0x0040) select the active high/low for the incoming signal

```
PMDSetBreakpointValue(phAxis, PMDBreakpoint1, 0x00400040);
```

Figure 17 - Breakpoint Value Set

#### 3) Set the breakpoint action and trigger

There are a lot of actions and triggers available, read through the below list to see what can be used. The names listed here are also the enumerated names within the SDK, and an example call is in Figure 18 - Setting the Breakpoint.

```
PMDSetBreakpoint(phAxis1, PMDBreakpoint1, sourceaxis, PMDBreakpointActionUpdate,
PMDBreakpointTriggerSignalStatus);
```

Figure 18 - Setting the Breakpoint



- Breakpoint Actions
  - PMDBreakpointActionUpdate
  - PMDBreakpointActionAbruptStop
  - PMDBreakpointActionSmoothStop
  - PMDBreakpointActionMotorOff
  - PMDBreakpointActionDisablePositionLoopAndHigherModules
  - PMDBreakpointActionDisableCurrentLoopAndHigherModules
  - PMDBreakpointActionDisableMotorOutputAndHigherModules
  - PMDBreakpointActionAbruptStopWithPositionErrorClear
- Breakpoint Triggers
  - PMDBreakpointTriggerDisable
  - PMDBreakpointTriggerGreaterOrEqualCommandedPosition
  - PMDBreakpointTriggerLessOrEqualCommandedPosition
  - PMDBreakpointTriggerGreaterOrEqualActualPosition
  - PMDBreakpointTriggerLessOrEqualActualPosition
  - PMDBreakpointTriggerCommandedPositionCrossed
  - PMDBreakpointTriggerActualPositionCrossed
  - PMDBreakpointTriggerTime
  - PMDBreakpointTriggerEventStatus
  - PMDBreakpointTriggerActivityStatus
  - PMDBreakpointTriggerSignalStatus
  - PMDBreakpointTriggerDriveStatus

## Absolute positioning with Stepper Motors (Open Loop Control)

A very helpful use of breakpoints is detecting the home signal. See Figure 19 - Breakpoint Code Example for a C++ example of these commands that could be used to stop motion after detecting the signal state change, this is pulled from the homing routine demo function.

```
//Next we will reset the event status for:
PMDResetEventStatus(phAxis, 0xFFFB); //0xFFFB - breakpoint 1
PMDResetEventStatus(phAxis, 0xBFFF); //0xBFFF - breakpoint 2

//we want to allow the stage to move into the positive side of travel slightly so we
can find the tripping point from the positive direction
//so we will use a breakpoint, configured for smooth stopping so the stage decelerates
into the positive half of travel
//the first half of this value is the selection mask for WHAT signals will cause the
breakpoint, the second half is the sense mask for WHAT STATES will trigger the
breakpoint
//So, with 0x0008 0008 we are expecting a capture source to go high will trigger the
breakpoint
PMDSetBreakpointValue(phAxis, 0, firstBPValue);
PMDSetBreakpoint(phAxis, 0, 0, PMDBreakpointActionSmoothStop, 0x00080008);
```

Figure 19 - Breakpoint Code Example

### Electronic Gearing (Step and Direction)

Electronic gearing or step and direction inputs allow the controller to be commanded to move by an external source without needing to send any communications commands to the controller. Two inputs to the controller can be configured to act as an auxiliary set of inputs.

NOTE: A second axis handle is required to access the auxiliary encoder inputs, refer to C-Motion Axis Handles for more information on copying the axis interface handle to a second handle on 'axis 2' in order to configure the inputs.

There are two primary commands used to configure this profile mode and operation of the stage, see Figure 20 - Electronic Gearing Set Gear Ratio and Figure 21 - Electronic Gearing Set Gear Master for the syntax for the commands.

Setting the gear ratio allows finer or more coarse control of position from the same input source. This can be helpful if larger, fast motions are needed with fewer pulses from the master. This function scales everything such that 65,536 is a 1:1 ratio and allows negative numbers to invert the direction. So, increasing the number to 131,072 would be a ratio of 2:1, one master count or pulse moves the stage by 2 encoder counts. Setting it to 32,767 halves that ratio, such that two pulses of input signal move the stage by one encoder count.

```
PMDSetGearRatio(phAxis1, (PMDint32)egearratio);
```

Figure 20 - Electronic Gearing Set Gear Ratio

Setting the master sets up the input and control, the master axis should always be set to 1, and the source should be set to commanded, so this function call should always resemble Figure 21 - Electronic Gearing Set Gear Master.

```
PMDSetGearMaster(phAxis1, 1, 1);
```

Figure 21 - Electronic Gearing Set Gear Master

See Figure 22 - Electronic Gearing Function Example for a reference on configuring electronic gearing mode. Once configured, the host should occasionally check for error states from the controller to ensure that nothing unexpected occurred. Note that electronic gearing mode is another profile mode that is selected using the same commands as setting the profile in the trajectory generator. However, the controller does not follow a trajectory generated profile once this mode is engaged, and the controller will 'snap' between positions. It becomes the responsibility of the external controller to handle acceleration, deceleration and profile generation.

Two modes of input can be configured, step/direction or a quadrature encoder input. Where step/direction uses one pin to increment position and one pin to determine the polarity of the increment, the direction. Quadrature input the pin states alternate between high and low 90 degrees out of phase, and direction is determined by the order of their changing.

```
void ElectronicGearing(PMDAxisHandle * phAxis1, PMDAxisHandle * phAxis2, float
egearratio, int dirpolarity) {

    //So be sure to set these when using electronic gearing mode (step/direction)

    //here we are setting the signal sense bit to the appropriate value
    //this will reverse the direction of travel for a high/low signal on the
direction input
    PMDuint16 signalsense = 0;//active high/low control
    PMDSignalSense(phAxis2, &signalsense);
    if(dirpolarity)
        PMDSignalSense(phAxis2, signalsense & ~0x0002);//unset
    else
        PMDSignalSense(phAxis2, signalsense | 0x0002);//set
    //DoverSetSignalSenseBit(phAxis2, 0x0002, dirpolarity);

    //ensure that the second encoder source is set to pulse and direction mode
    //PMDAuxiliaryEncoderModePulseAndDirection
    //PMDEncoderSourcePulseAndDirection
    PMDEncoderSource(phAxis2, PMDEncoderSourcePulseAndDirection);

    //setting the gear ratio
    //this has a unity scale factor of 65536, where one input pulse is 1 output
pulse
    //this is a fixed decimal point number that is 16.16, so 0x00010000 or 65536 is
unity (1)
    //a ratio of 1 input to 10 output has a ratio of 65536
    //a ratio of 10 input to 1 output has a ratio of 655
    //here we are converting the sent decimal ratio
    egearratio *= 65536;
    PMDSetGearRatio(phAxis1, (PMDint32)egearratio);

    //make sure both axes are enabled
    stageEnable(phAxis1);
    PMDSetOperatingMode(phAxis2, PMDOperatingModeAxisEnabled);

    //set the master and source axis
    PMDSetGearMaster(phAxis1, 1, 1);

    //set up the profile mode last, right before the update that makes all this go
into effect
    PMDSetProfileMode(phAxis1, PMDProfileModeElectronicGear);
    PMDUpdate(phAxis1);
}
```

Figure 22 - Electronic Gearing Function Example

## TOP – Trigger on Position

Trigger on position is an output mode that allows high precision triggering off the raw encoder position. The output is two signals, a gate signal that either allows or prevents the trigger signal from reaching the output of the controller, and the trigger signal itself. These signals are “and” gated together within the controller, so no false triggers will be possible without the gate signal. The gate signal can also serve as a useful signal to drive illumination sources, and things of that nature within a system.

### Usage notes:

- SmartStage linear systems have multiplexed I/O lines, and must be set to the correct GPIO state for these signals to reach the main connector. See Setting the GPIO State for more detail on this.
- The trigger signal is a 250ns pulse that outputs on an output pin from the controller, see the controller/stage datasheet for more information on wiring details. This pulse duration is fixed and cannot be modified.
- Trigger on position is not compatible with a stage/controller using SPI encoder data. Contact Dover for more information, SmartStage Linear systems ship with SPI encoder data used by default. Run GetEncoderSource() to determine if this feature is enabled on the controller, if the returned value is decimal 11, or 0xB the encoder is set for SPI data.
- Tabled trigger on position mode requires usage of memory space on the controller, which the MC58113 has a fixed user-configurable memory buffer size of 4096 32-bit data points. This memory space is also shared with the trace buffer space, so using this feature reduces the amount of data that can be captured.

Trigger on position features two main use cases, incremental triggering (firing an output pulse every X encoder counts) and table-based positions to fire the pulse. Example code for each use case is reviewed below.

Incremental triggering can be configured using the example function in Figure 23 - Incremental TOP Example. There are a set of parameters to configure for the operation of the output, some of which configure things on the second axis handle which must be defined in order to address these features. This example uses the enum documented in TriggerOnPosition.h to make the example code easier to read and understand.

```
//NOTE TOP does not work in SPI encoder mode
void IncrementalTOPgated(PMDAxisHandle * phAxis1, PMDAxisHandle * phAxis2, PMDUint32
increment, PMDint32 start_pos, PMDint32 stop_pos)
{
    //ensure we disable everything first before changing values
    DisableTOP(phAxis1, phAxis2);

    //2 < max pos < 32bit
    PMDSetFeedbackParameter(phAxis2, TOPincrement, increment); //this is the distance
between output pulses in encoder counts
    PMDSetFeedbackParameter(phAxis2, TOPtrigger, 0); //trigger
    PMDSetFeedbackParameter(phAxis2, TOPmode, TOPincremental); //periodic/incremental
mode (3)

    PMDSetFeedbackParameter(phAxis1, TOPmode, TOPdisabled); //Disable
    PMDSetFeedbackParameter(phAxis1, TOPoutputStart, start_pos); // triggering
enabled between the start position
    PMDSetFeedbackParameter(phAxis1, TOPoutputStop, stop_pos); // and the stop
position
    PMDSetFeedbackParameter(phAxis1, TOPmode, TOPenabled); //level mode (0)
}
```

Figure 23 - Incremental TOP Example

Tabled TOP must be enabled after first configuring the tabulated data in controller memory. The example function in TriggerOnPosition.cpp demonstrates this in either tabledTOP() or tabledTOPfile(). The code snippet in Figure 24 - Tabled TOP Example shows configuring the controller memory once the data points have been determined, loaded or put into an array. The commands at the end enable TOP mode to run, and return the length of the tabled memory buffer space since this is shared with traces and the host machine should be aware of the now limited trace memory space.

```
//Buffer setup
//create the TOP table in controller buffer space
PMDSetBufferStart(phAxis1, TOPBUFFID, topbuffmemloc); //set up the memory address
and buffer ID
PMDSetBufferLength(phAxis1, TOPBUFFID, TOPBUFFERLENGTH); //set up the buffer
length
PMDSetBufferWriteIndex(phAxis1, TOPBUFFID, topbuffmemloc); //ensure a write index
of 0 to write from the beginning
for (int i = 0; i < TOPBUFFERLENGTH; i++) {
    PMDWriteBuffer(phAxis1, TOPBUFFID, toppoints[i]);
}
PMDSetBufferReadIndex(phAxis1, TOPBUFFID, topbuffmemloc);

//This code reads the TOP buffer to verify that it was set correctly.
/*PMDSetBufferReadIndex(phAxis1, TOPBUFFID, 0);
for (int i = 0; i < TOPBUFFERLENGTH; i++) {
    PMDint32 blarf = 0;
    PMDReadBuffer(phAxis1, TOPBUFFID, &blarf);
    printf("ibuff %i  %i\n", i, blarf);
    //PMDWriteBuffer(phAxis1, pbuffID, ppoints[i]);
}*/

//make sure to close the file
fclose(stream);

//enable TOP output
PMDSetFeedbackParameter(phAxis1, TOPmode, TOPdisabled); //Disable
PMDSetFeedbackParameter(phAxis1, TOPtabledBuffID, TOPBUFFID); // Begin
Triggering
PMDSetFeedbackParameter(phAxis1, TOPmode, TOPtabled); //Enable Mode

//return the size of the memory buffer used
return buffcount;
}
```

Figure 24 - Tabled TOP Example

### UDP – User Defined Profile Mode

User defined profile mode allows external signals to drive motion in a defined way, specified by the host controller. This allows for things like coordinated motion, triggered motion, and pre-planned sets of positions to be loaded into memory and run on the controller.

Notes about this section:

- This guide will cover setting up and configuring coordinated motion between axes, as this is the more common use case for this feature. See Additional Reference Guides for additional material on using this feature.
- This profile mode generally requires usage of memory space on the controller, which the MC58113 has a fixed user-configurable memory buffer size of 4096 32-bit data points. This memory space is also shared with the trace buffer space, so using this feature reduces the amount of data that can be captured.

To configure this feature to run properly, users must configure the data in the controllers' memory buffer, then enable the feature. This can be done using the demo code in Figure 25 – Configure UDP Example Function. Essentially this function sets up the memory buffer, then calls the enable demo function, both of which are discussed in more detail below.

```
int configureUDPfile(PMDAxisHandle * phAxis1, PMDAxisHandle * phAxis2, const char  
*filename, int enc_res);  
void enableUDPmode(PMDAxisHandle * phAxis1, PMDAxisHandle * phAxis2);
```

Figure 25 – Configure UDP Example Functions

The example function `configureUDPfile()` is an example that handles reading a file, and loading in the lists of input positions and output positions into the appropriate memory locations on the controller. It also includes a call to enable UDP profile mode. Figure 26 - `configureUDPfile()` Example below shows the important sections of the code that send the appropriate arrays of data points to the memory locations on the controller.

Since the controller has a fixed memory size, we return the size of the data space used by this process so the host can be aware of the remaining memory size if traces are used.

```
//MC58113 has a maximum buffer size of 0x1000, so we just create arrays that size plus
one

//NVRAM starts well outside of the trace buffer area
//trace buffer is 0x0000 to 0x1000 (4096)
int ipoints[4097];
int ppoints[4097];

char line[1024];
while (fgets(line, 1024, stream))//&& buffcount <= buffsize)
{
    char* tmp = _strdup(line);
    char * tmp2;
    ipoints[buffcount] = atoi(strtok_s(tmp, ",", &tmp2));
    ppoints[buffcount] = atoi(strtok_s(NULL, "\n", &tmp2));
    // NOTE strtok destroys tmp - we need to clear it
    free(tmp);
    buffcount++;
}

uint32_t ibuffmemloc = 0x0000;//this memory location can be set somewhat
arbitrarily, generally start from 0
uint32_t pbuffmemloc = ibuffmemloc + buffcount + 1;//this needs to start right
after the ibuffer without overlapping

//Buffer setup
//create the I (input) table
PMDSetBufferStart(phAxis1, IBUFFID, ibuffmemloc);//set up the memory address and
buffer ID
PMDSetBufferLength(phAxis1, IBUFFID, buffcount);//set up the buffer length
PMDSetBufferWriteIndex(phAxis1, IBUFFID, 0);//ensure a write index of 0 to write
from the beginning
//create the P (position) table
PMDSetBufferStart(phAxis1, PBUFFID, pbuffmemloc);//set up the memory address and
buffer ID
PMDSetBufferLength(phAxis1, PBUFFID, buffcount);//set up the buffer length
PMDSetBufferWriteIndex(phAxis1, PBUFFID, 0);//ensure a write index of 0 to write
from the beginning
for (int i = 0; i < buffcount; i++) {
    PMDWriteBuffer(phAxis1, IBUFFID, ipoints[i]);
    PMDWriteBuffer(phAxis1, PBUFFID, ppoints[i]);
}

//This code reads the UDP buffer to verify that it was set correctly.
//uncomment this to debug and make sure the buffer is getting written to the
controller correctly
/*PMDSetBufferReadIndex(phAxis1, PBUFFID, 0);
```

```

    for (int i = 0; i < buffcount; i++) {
        PMDint32 blarf = 0;
        PMDReadBuffer(phAxis1, PBUFFID, &blarf);
        printf("pbuff %i  %i\n", i, blarf);
    }
    PMDSetBufferReadIndex(phAxis1, IBUFFID, 0);
    for (int i = 0; i < buffcount; i++) {
        PMDint32 blarf = 0;
        PMDReadBuffer(phAxis1, IBUFFID, &blarf);
        printf("ibuff %i  %i\n", i, blarf);
    }*/

    enableUDPmode(phAxis1, phAxis2);

    //note that we need to double the UDP buffer count size since we have an input
    and output buffer
    return buffcount * 2 + 1;
}

```

Figure 26 - configureUDPfile() Example

Then enabling the UDP profile mode requires a little more setup to ensure that everything is properly configured and set up. When configuring everything, the parameters are all buffered but should only be modified when in a different profile mode to avoid unexpected behavior. So first we put the controller in a known profile mode (trapezoidal is used in the example).

The auxiliary encoder inputs are configured appropriately, the example functions are setting up for coordinated motion between two axes, and so are expecting a quadrature input signal from the encoder pass-through from another stage. After that, most parameters for the profile mode are set using the new enum defined in UDPmode.h, making them human-readable and easier to set and track. Finally, the profile mode is set to 10, or UDP mode, and an update command is issued to initialize everything. See the example code in Figure 27 - Enable UDP Mode Example for additional comments on each function and the parameters set.

```

void enableUDPmode(PMDAxisHandle * phAxis1, PMDAxisHandle * phAxis2) {
    printf("Enabling UDP mode from controller buffer.\n");

    //Note that this function resets the position in the memory index and resets the
    input encoder position. The output profile will completely restart.
    //To avoid this, comment out the following lines from the code below:
    //PMDSetProfileParameter(phAxis1, PMDProfileParameterStartIndex, 0);
    //PMDSetActualPosition(phAxis2, 0);

    PMDResetEventStatus(phAxis1, 0); //literally all event statuses will be reset
    PMDResetEventStatus(phAxis1, 0xDFFF); //reset runtime errors in UDP mode

    PMDSetProfileMode(phAxis1, PMDProfileModeTrapezoidal);
    PMDUpdate(phAxis1);

    //for testing purposes, disable motion error event actions
    PMDSetEventAction(phAxis1, PMDEventActionEventMotionError, PMDEventActionNone);

    //ensure that the second encoder source is set to incremental to read AqB inputs
    PMDSetEncoderSource(phAxis2, 0x0000);
    //ensure that the starting encoder position is 0 to prevent and erratic starting
    behavior
}

```



```

PMDSetActualPosition(phAxis2, 0);

//configure profile parameter values
PMDSetProfileParameter(phAxis1, PMDProfileParameterSource, 0x0001); //set the
profile source to the encoder input from "axis 2"
//NOTE: Axis 2 in this instance does not refer to a second axis, but rather a
virtual "second axis" to source another encoder signal from
//Bits [7:0] specify the source axis, 0 means Axis1, 1 means Axis2, and so
forth.
//Bits[15:8] specify the source type, 0 means actual position, 1 means commanded
position, and 2 means time(axis field ignored). All other values are reserved.
PMDSetProfileParameter(phAxis1, PMDProfileParameterRateScalar,
0x00010000); //sets the ratio between input and output
//The Rate Scalar is a signed quantity scaled by 1/65536, that is, a 16.16 fixed
point quantity.
//(first 16 bits are the whole number, second 16 bits are the fractional)
//0x00010000 is 1.0
PMDSetProfileParameter(phAxis1, PMDProfileParameterStartValue, 0); //sets the
starting value for the contour (position index?) MUST BE 0
PMDSetProfileParameter(phAxis1, PMDProfileParameterStartIndex, 0); //sets the
starting index for the contour (memory index?)
PMDSetProfileParameter(phAxis1, PMDProfileParameterStopValue, 0xFFFFFFFF); //sets
the value within the contour that it should stop following
//as long as this is outside the bounds of the contour it should be fine,
otherwise it will stop where this is set
PMDSetProfileParameter(phAxis1, PMDProfileParameterIBufferID, IBUFFID); //I
buffer memory address
PMDSetProfileParameter(phAxis1, PMDProfileParameterPBufferID, PBUFFID); //P
buffer memory address

PMDSetProfileMode(phAxis1, PMDProfileModeUserDefinedProfile); //Profile mode 10
(0xA)

//make sure all updates are done before sending an update
PMDUpdate(phAxis1);
}

```

Figure 27 - Enable UDP Mode Example

## Setting the GPIO State

SmartStage linear systems have multiplexed outputs, so the host machine must choose which I/O will be accessible to the main connector of the system.

### Usage notes:

- These are custom firmware features only available on SmartStage linear systems.
- These functions use some commands that are not part of the original SDK and DLL, so if these are rebuilt, they will need to be included. See Figure 29 - Set GPIO Custom Function Definitions

This feature uses the custom command `PMDSetCustomParameter()` to set specific parameters to enable and set the pin modes for multiplexing signals within the controller. See Figure 28 - SetGPIO Example for an example function that handles setting the mode to a known state. See the SmartStage linear system (or appropriate controller) guide for a reference on which I/O state connects which I/O to the main connector.

```
void configureGPIO(PMDAxisHandle * hAxis1, PMDuint16 gpiomode) {
    //See each GPIO case for a description of each output signal, and what other
    //modes to pair them with.
    //lotsa reading, sorry

    //this function simplifies the process of setting the GPIO mux
    //3 pin states and one enable pin control the mux
    //0000WXYZ - the bytes sent in SetCustomParameter set each mux input state
    accordingly
    //each of the following states allows different inputs/outputs to be passed
    through to the appropriate location

    //GPO state set - sets all GPO to either high/low
    //16(0x10)
    //GPO set high - sets selected GPO high, 0 means no change to individual state
    //17(0x11)
    //GPO clear state - sets the selected GPO low, 0 means no change to individual
    state
    //18(0x12)
    //GPIO set state - this configures these pins (which default to inputs where all
    I/O is disabled) into an input or output state
    //19(0x13)

    PMDSetCustomParameter(hAxis1, GPIOSetMode, 0x0F); //this configures all pins as
    outputs, enabling them to set up an output mode

    switch (gpiomode) {
        //Enable/disable
        case GPIO_Disabled:
            PMDSetCustomParameter(hAxis1, GPIOSetLow, 0x08);
            break;
        case GPIO_Enabled:
            PMDSetCustomParameter(hAxis1, GPIOSetHigh, 0x08);
            break;
        case GPIO_State0:
            //IOA1 is general purpose input, step/direction, Quad A (coordinated
            motion)
            //IOA2 is general purpose input, step/direction, Quad B (coordinated
            motion)
    }
}
```

```

//IOA3 is PEG output, LUT PEG output, general purpose output
//Secondary axis is passed through
PMDSetCustomParameter(hAxis1, GPIOSetState, 0x08);
break;
case GPIO_State1:
//IOA1 is general purpose input, step/direction, Quad A (coordinated
motion)
//IOA2 is general purpose input, step/direction, Quad B (coordinated
motion)
//IOA3 is Axis out is general purpose output
//Secondary axis is passed through
PMDSetCustomParameter(hAxis1, GPIOSetState, 0x0C);
break;
case GPIO_State2:
//IOA1 is Quad A (from IOB2, the secondary axis)
//IOA2 is Quad B (from IOB3, the secondary axis)
//IOA3 is Axis out is general purpose output
//IOB1 is passed through
//IOB2 is mapped to IOA1
//IOB3 is mapped to IOA2
//This state is for coordinated motion, and reads position from the
secondary axis which acts as a master
//Primary axis in state 2 should use a secondary axis in state 4 or 5
PMDSetCustomParameter(hAxis1, GPIOSetState, 0x0A);
break;
case GPIO_State3:
//IOA1 is general input
//IOA2 is Enc A from primary axis
//IOA3 is Enc B from primary axis
//IOB1 is Enc A out from primary
//IOB2 is Enc B out from primary
//IOB3 is passed through
//Primary axis driving coordinated motion - secondary axis should be in
mode 0 or 1
PMDSetCustomParameter(hAxis1, GPIOSetState, 0x0E);
break;
case GPIO_State4:
//IOA1 is general input
//IOA2 is Enc A out
//IOA3 is Enc B out
//Secondary axis is passed through
PMDSetCustomParameter(hAxis1, GPIOSetState, 0x09);
break;
case GPIO_State5:
//IOA1 is PEG output
//IOA2 is Enc A out
//IOA3 is Enc B out
//Secondary axis is passed through
//This should be used as a secondary axis when interested in both PEG and
coordinating motion using encoder signal
PMDSetCustomParameter(hAxis1, GPIOSetState, 0x0D);
break;
case GPIO_State6:
//IOA1 is PEG output
//IOA2 is PEG gate signal

```

```

        //IOA3 is Axis out is general purpose output
        //Secondary axis is passed through
        PMDSetCustomParameter(hAxis1, GPIOSetState, 0x0B);
        break;
    case GPIO_State7:
        //encoder programming mode
        //Secondary axis is passed through
        PMDSetCustomParameter(hAxis1, GPIOSetState, 0x0F);
        break;
    default:
        //default state is to disable the outputs, without losing whatever was
        previously configured
        PMDSetCustomParameter(hAxis1, GPIOSetLow, 0x08);
        break;
    }
}

void getGPIO(PMDAxisHandle * hAxis1, PMDuint16 * readgpiothing) {
    PMDGetCustomParameter(hAxis1, GPIOGetState, readgpiothing);
}

```

Figure 28 - SetGPIO Example

```

PMDresult PMDSetCustomParameter(PMDAxisInterface axis_intf, PMDuint16 param, PMDint16
value)
{
    return SendCommandWordWord(axis_intf, 0xCA, param, value);
}

PMDresult PMDGetCustomParameter(PMDAxisInterface axis_intf, PMDuint16 param, PMDuint16
*value)
{
    return SendCommandWordGetWord(axis_intf, 0xCB, param, value);
}

```

Figure 29 - Set GPIO Custom Function Definitions

## Error Codes

The library supports a number of error code returns to help diagnose issues that occur with the controllers. See Figure 30 - C-Motion Error Codes for a list of the various error codes that can be returned. These are defined in an enum in PMDecode.h.

```
typedef enum PMDErrorCodesEnum {

    // processor error codes
    PMD_NOERROR                = 0,
    PMD_ERR_OK                 = 0,
    PMD_ERR_Reset              = 0x01,
    PMD_ERR_InvalidInstruction = 0x02,
    PMD_ERR_InvalidAxis        = 0x03,
    PMD_ERR_InvalidParameter   = 0x04,
    PMD_ERR_TraceRunning        = 0x05,
    PMD_ERR_BlockOutOfBounds    = 0x07,
    PMD_ERR_TraceBufferZero     = 0x08,
    PMD_ERR_BadSerialChecksum   = 0x09,
    PMD_ERR_InvalidNegativeValue = 0x0B,
    PMD_ERR_InvalidParameterChange = 0x0C,
    PMD_ERR_LimitEventPending   = 0x0D,
    PMD_ERR_InvalidMoveIntoLimit = 0x0E,
    PMD_ERR_InvalidOperatingModeRestore = 0x10,
    PMD_ERR_InvalidOperatingModeForCommand = 0x11,
    PMD_ERR_BadState            = 0x12,
    PMD_ERR_AtlasNotDetected     = 0x14,
    PMD_ERR_HardFault            = 0x13,
    PMD_ERR_BadSPIChecksum       = 0x15,
    PMD_ERR_InvalidSPIprotocol   = 0x16,
    PMD_ERR_InvalidTorqueCommand = 0x18,
    PMD_ERR_BadFlashChecksum     = 0x19,
    PMD_ERR_InvalidFlashModeCommand = 0x1A,
    PMD_ERR_ReadOnly             = 0x1B,
    PMD_ERR_InitializationOnlyCommand = 0x1C,
    PMD_ERR_IncorrectDataCount    = 0x1D,
    PMD_ERR_MoveInError          = 0x1E,
    PMD_ERR_WaitTimedOut         = 0x1F,

    // non-processor errors
    PMD_ERR_InvalidOperation     = 0x7FD0,
    PMD_ERR_NotConnected         = 0x7FD1,
    PMD_ERR_NotResponding        = 0x7FD2,
    PMD_ERR_CommPortRead        = 0x7FD3,
    PMD_ERR_CommPortWrite       = 0x7FD4,
    PMD_ERR_InvalidSerialPort    = 0x7FDB,
    PMD_ERR_InterfaceNotInitialized = 0x7FDF,
    PMD_ERR_OpeningPort          = 0x7FE0,
    PMD_ERR_Driver               = 0x7FE1,
    PMD_ERR_NoDPRAM              = 0x7FE2,
    PMD_ERR_DPRAM                = 0x7FE3,
    PMD_ERR_Timeout              = 0x7FE4,
    PMD_ERR_WaitCancelled         = 0x7FE5,
    PMD_ERR_CommunicationsError   = 0x7FFC,
    PMD_ERR_CommTimeoutError      = 0x7FFD,
    PMD_ERR_ChecksumError         = 0x7FFE,
    PMD_ERR_CommandError         = 0x7FFF

} PMDErrorCode;
```

Figure 30 - C-Motion Error Codes

### Overall Controller Behavior Notes

Notes about interesting controller behaviors to be aware of when programming:

- 1) Trapezoidal profile mode resets the velocity to 0 when a limit is encountered, so a new set of profile parameters should be issued (velocity, acceleration, deceleration, etc.)
- 2) Velocity contouring mode supports positive and negative values, but other profile modes do not, this is because velocity profile mode uses the polarity of the velocity to determine the travel direction, where other profile modes use absolute position.

### Known C-Motion Issues:

#### Serial Connection Desynchronization

A known issue with asynchronous communications is when the host and controller become desynchronized. Since all communications with the controller work off of expected command sizes and waits for a buffer to be filled before processing anything from the host. If a byte is missed then this can be interrupted, causing a lapse in communications where the controller always immediately returns an error, or stops responding to the host. To solve this, the host should send bytes of 0's to the controller until the controller responds with something. Then normal communications can resume.

The latest release of C-Motion should handle these cases and has a function internal to the DLL and SDK called `PMDSerial_Sync`. This should not normally be needed by user code.

#### Driver Dependency Issues

Note that the libraries have support for a variety of device drivers, and some are key to include and have installed on the host machine. Serial drivers come with Windows, but the IXXAT CAN drivers do not, and if they are needed, they must be downloaded from their website. The standard libraries have support for more devices and hardware as well, so these dependencies will need to be removed as necessary until support for installed devices is included.

## Starting Projects in Visual Studio

### Visual Studio 2019 C++ Project Setup

#### *Finding the files:*

The files referenced in the instructions below should be included on the USB stick that came with the controller/stage. See Definitions Used Within This Guide, USB Stick, Resource locations.

#### *VS2019 C++ Project Notes:*

Users must configure the project and compiler properly to get the SDK to build error-free. There are some files that will cause the code not to compile or run depending on the hardware drivers and communications devices installed on the host machine.

When compiling for x64bit machines, the CAN communications .c and .h files must be removed from the project. The files that are not needed are listed below. The CAN libraries are built for x86 (32bit) machines.

- PMDCAN.c
- PMDCAN.h
- PMDIXXATCAN3.c
- PMDIXXATCAN3.h

When not using a communications protocol, users must be certain to exclude files that use libraries and hardware that do not have installed drivers.

#### *VS2019 Starting a New C++ Project:*

If build errors are encountered, please read through the VS2019 C++ Common Errors section below to reference some common solutions.

- 1) Start a new project in Visual Studio 2017, a C/C++ Console App per Figure 31 - VS2019 C++ New Project

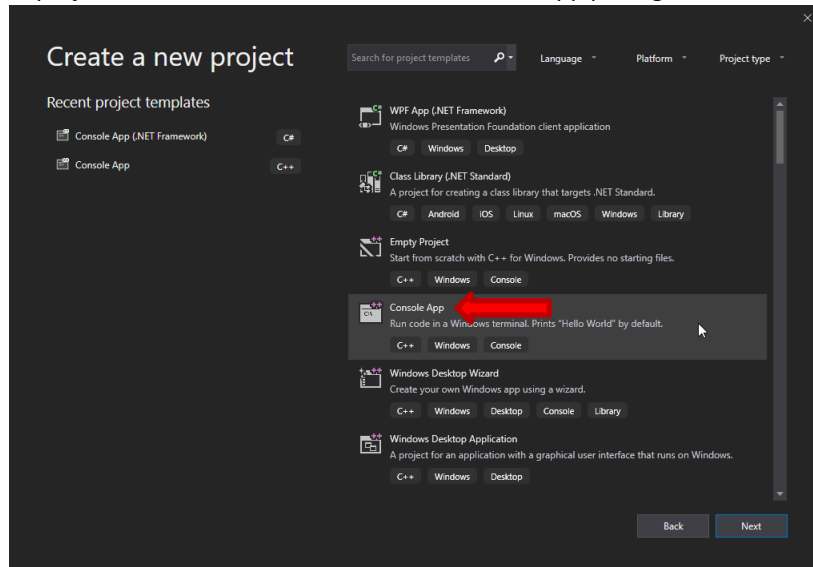


Figure 31 - VS2019 C++ New Project

- 2) Copy and paste local copies of the “C” and “Include” folders from C-Motion inside the project directory, at the same level as the projectname.vcxproj file
- 3) Right click the “Header Files” folder in the solution explorer view and select “Add”, then “Existing Item...” and select all the files from the “Include” directory in the project folder per Figure 32 - VS2019 Adding Header Files
- 4) Right click the “Source Files” folder in the solution explorer view and select “Add”, then “Existing Item...” and select all the files from the “C” directory in the project folder

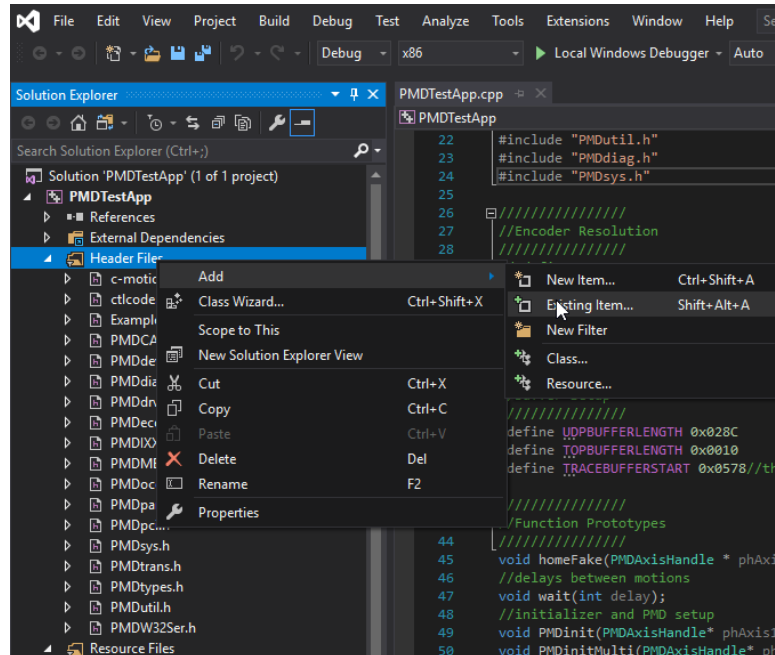


Figure 32 - VS2019 Adding Header Files



- 5) Right click the “Resource Files” folder in the solution explorer view and select “Add”, then “Existing Item...” and select the following items from the sub-folders in “Include”:
  - a. ..\Include\IXXAT\vcisdsk.lib
  - b. ..\Include\PLX\PlxApi.lib
  - c. ..\Include\NI\ni845x.lib

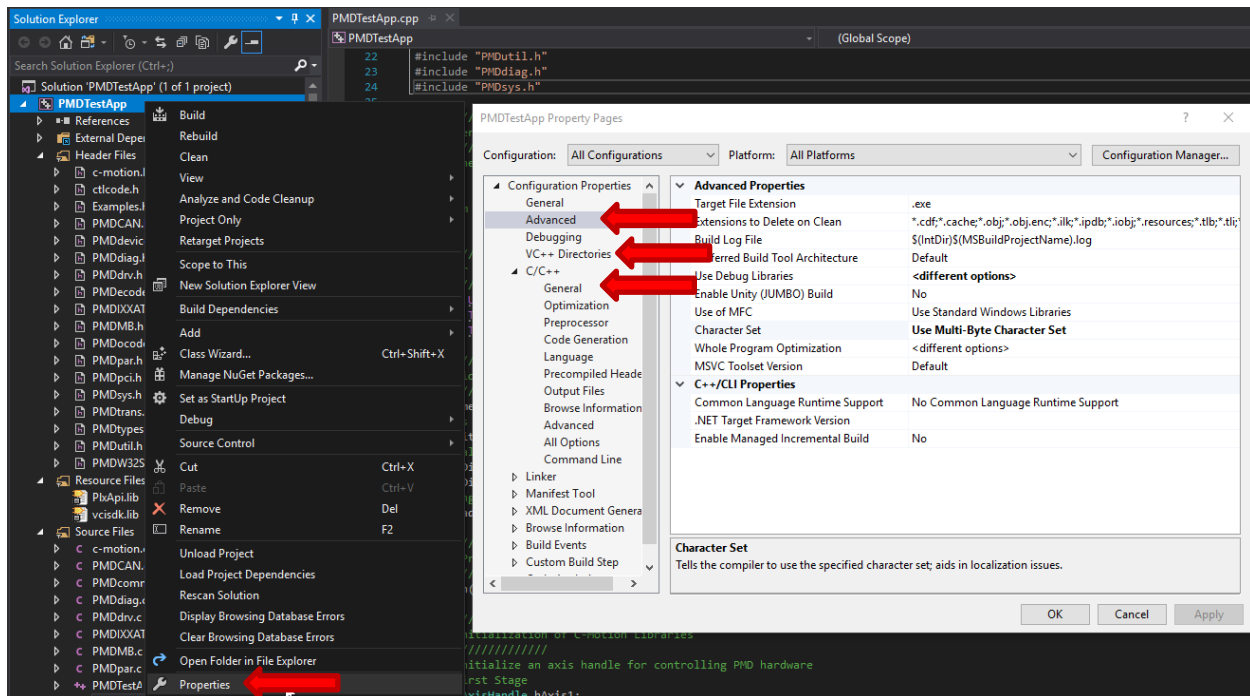


Figure 33 - VS2019 Project Properties

- 6) Project Properties need to be modified, right click on the project name in the solution explorer view and select properties – be sure to select all configs and all platforms per Figure 33 - VS2019 Project Properties
- C/C++ - General – SDL Checks – No
  - Advanced – Character Set – Use Multi-Byte Character Set
  - VC++ directories – include directories – add the following folders per Figure 34 - VS2019 VC++ Directories
    - ..\Include
    - ..\Include\IXXAT
    - ..\Include\PLX
    - ..\Include\NI

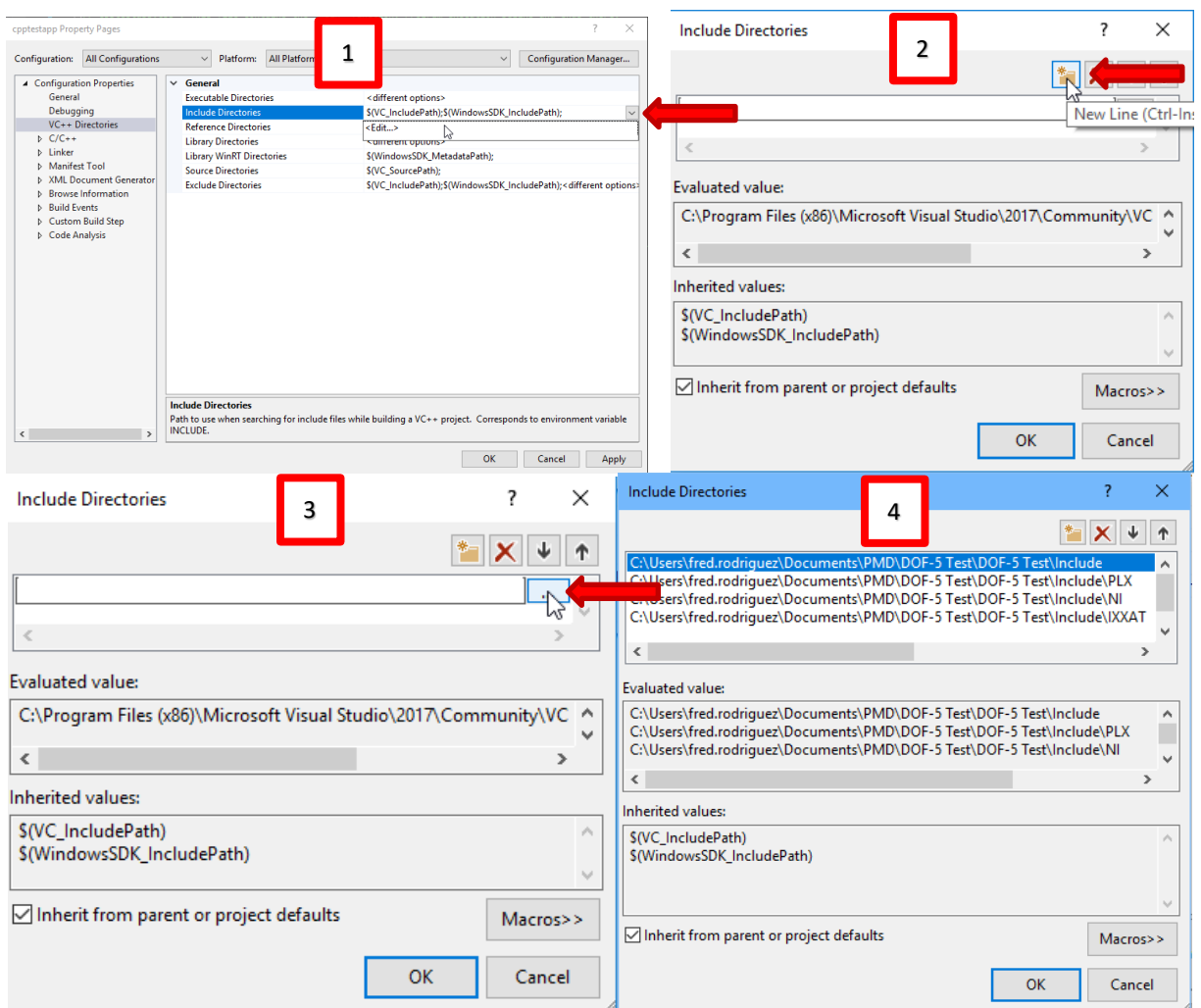


Figure 34 - VS2019 VC++ Directories

- 7) Exclude the .c files that have uninstalled dependencies or required devices. The following interfaces have device specific driver requirements and will not compile unless those drivers are installed. See Figure 35 - VS2019 Excluding Uninstalled Dependencies
- a. PMDNISPI.c
    - v. For use with SPI communications
    - vi. Install the drivers for an NI845x device
  - b. PMDCAN.c
    - vii. For use with CAN communications
    - viii. Install the drivers for an IXXATUSB-to-CAN adapter

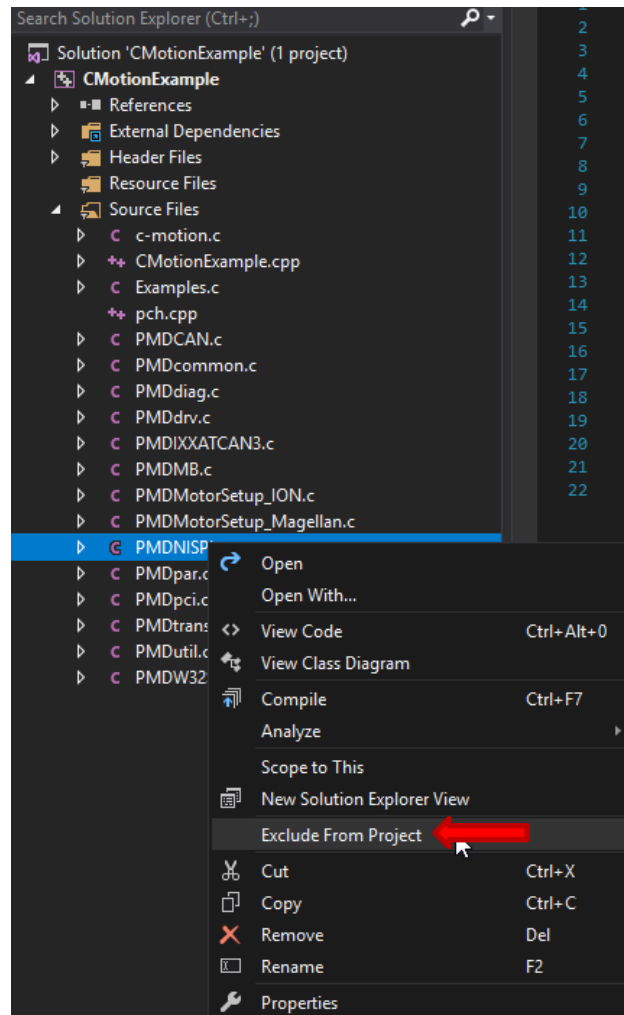


Figure 35 - VS2019 Excluding Uninstalled Dependencies

- 8) Build the solution to ensure that everything has linked correctly per Figure 36 - VS2019 Rebuild
  - a. See VS2017 C++ Common Errors below to troubleshoot errors when building the project solution. VCINLP.dll is a common CAN driver problem, SAFESEH is a linker exception handler problem, etc.

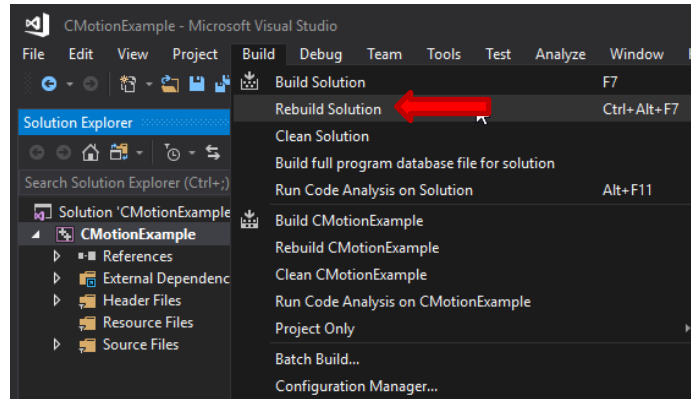


Figure 36 - VS2019 Rebuild

- 9) The program should now be set up, and ready to run, the “main entry point” to the program, is projectname.c
- 10) Running the program can be done by selecting the debug menu, and selecting either “Start Debugging” or “Start Without Debugging”, see Figure 37 - VS2019 Run Debugging
  - a. Begin programming the stage control program or continue reading for where to find example code to start moving the stage with. Note that the examples are explained below the instructions on how to use them, and each example includes well documented functions and setup code.

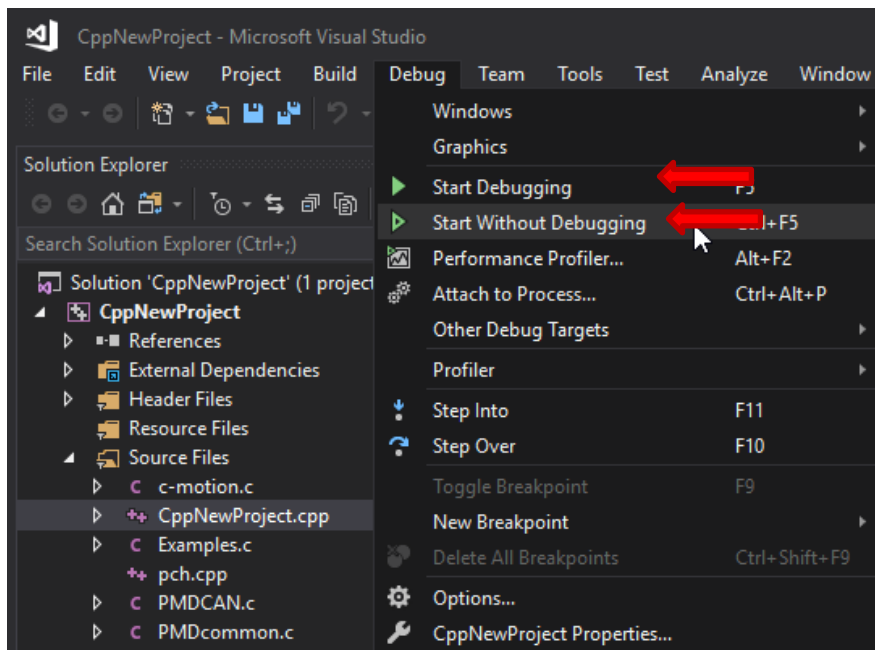


Figure 37 - VS2019 Run Debugging

- 11) Once the project has been “Built” or compiled, it can also be run by using the generated “projectname.exe” file found at the debug output location, the typical default is below, and can also be found by browsing through project properties, in “General” and selecting the output directory, per Figure 38 - VS2019 Output Directory
- b. ..\projectname\Debug

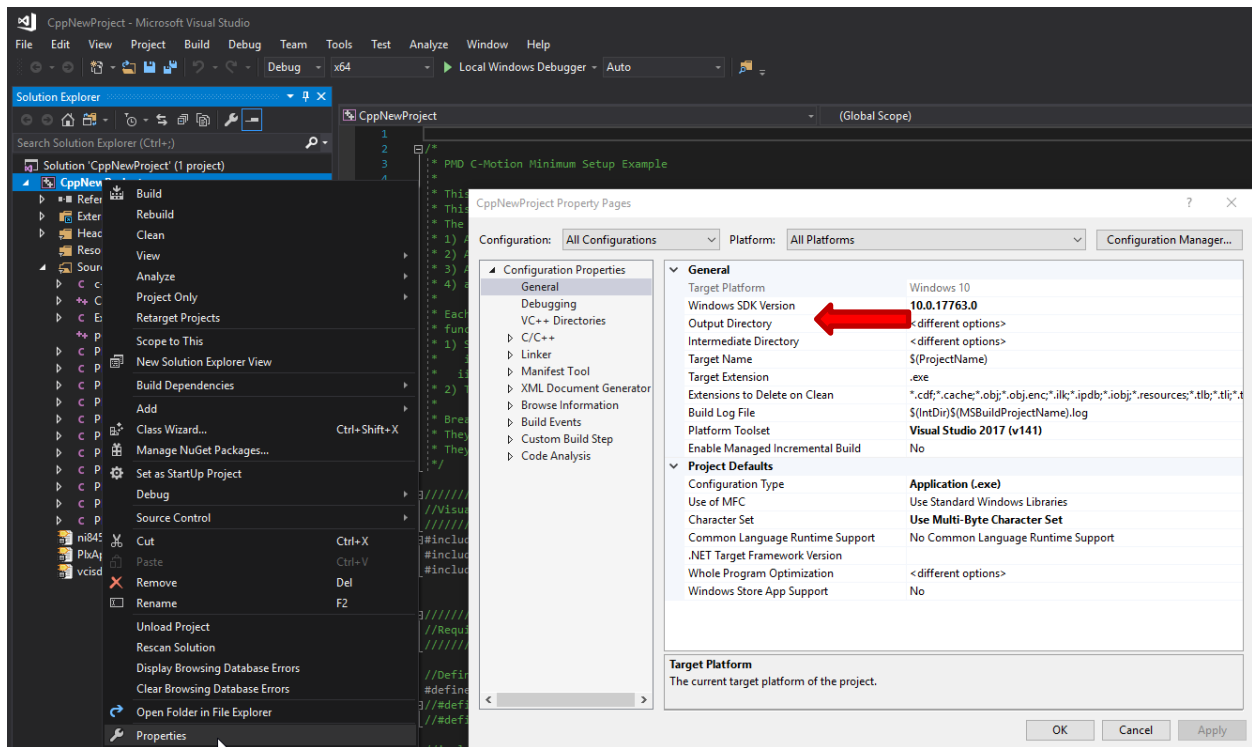


Figure 38 - VS2019 Output Directory

### VS2019 C++ Common Errors

#### VS2019 C++ SAFESEH Warning:

If the build is throwing a SAFESEH warning, or Link 2026 errors, then change the “Image has safe exception handlers” setting in the project properties window under Linker -> Advanced, down at the bottom of the list per Figure 39 - VS2019 SAFESEH Warning

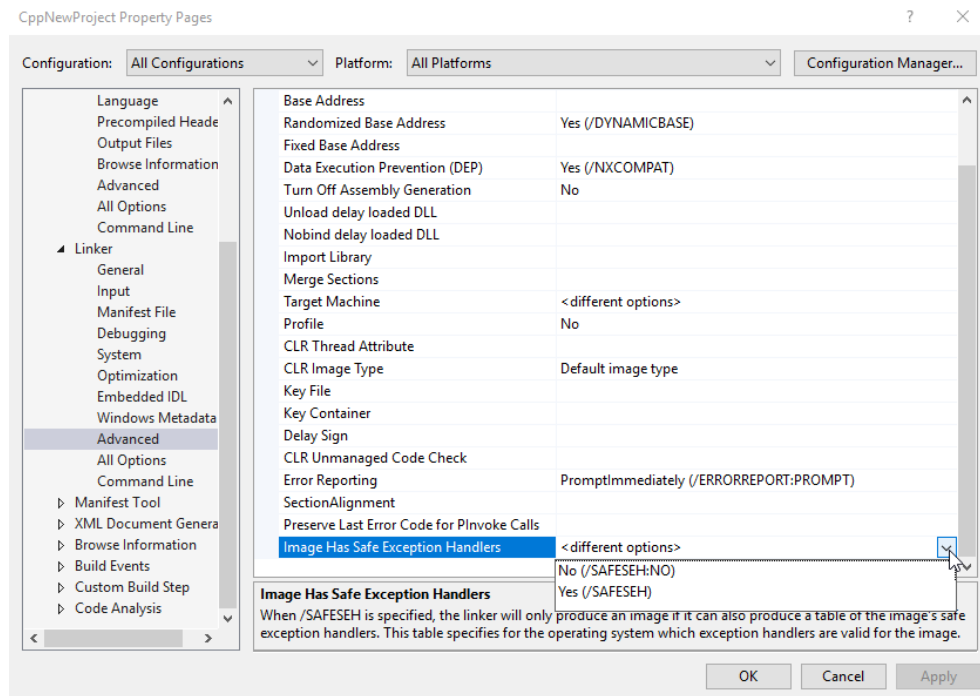


Figure 39 - VS2019 SAFESEH Warning

#### VS2019 C++ Missing Drivers:

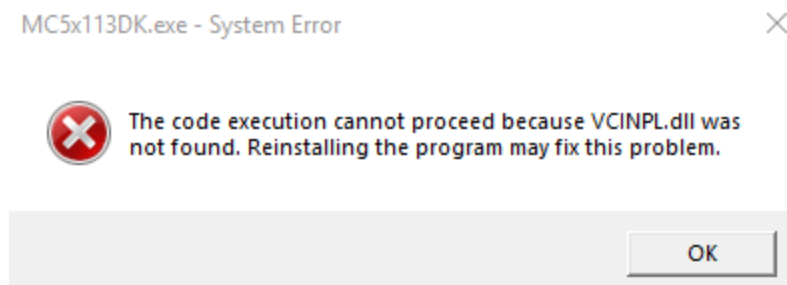


Figure 40 - VS2019 Missing Driver Error

This error pops up when the correct drivers are not installed for the USB device needed, specifically for the IXXAT USB-to-CAN adapter. Ensure that the drivers for the devices in use are installed or remove the dependent files that require the drivers. See Figure 40 - VS2019 Missing Driver Error

### VS2019 C++ Runtime Library Errors:

A long string of link errors in the build output could likely indicate an incorrect selection for the runtime library. Note that this selection is build dependent, and must be either debug or release, otherwise a long string of build errors can occur. See Figure 41 - VS2019 Linker Error

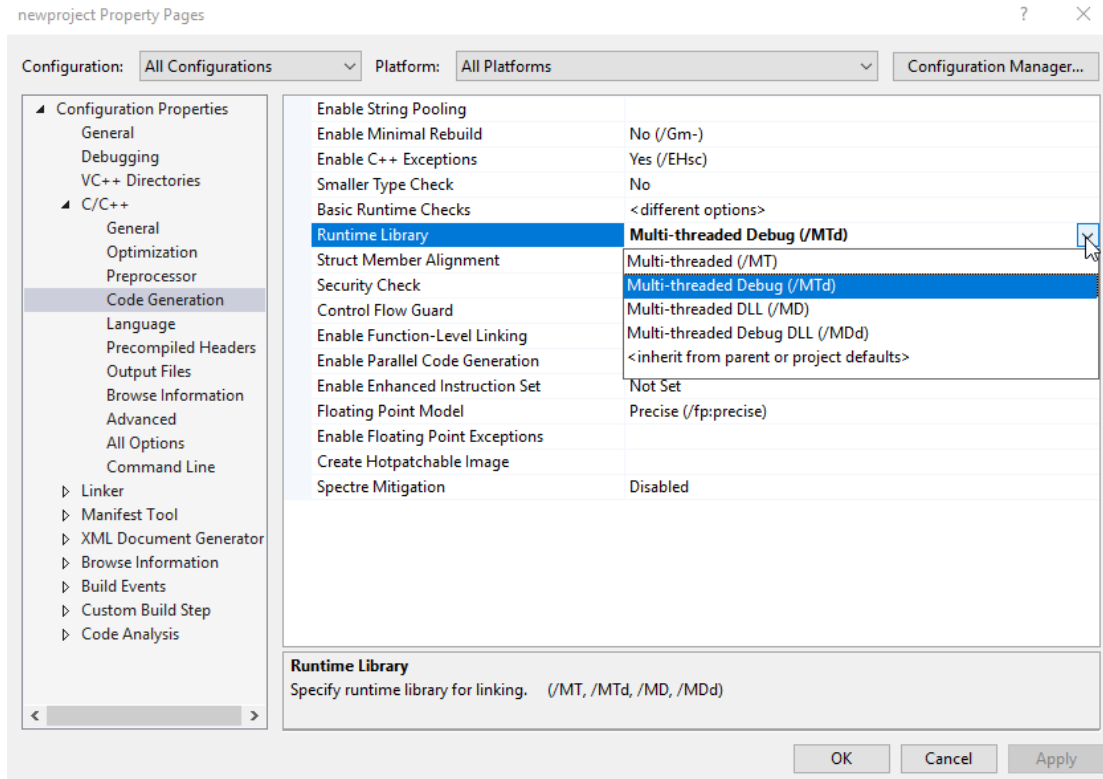


Figure 41 - VS2019 Linker Error

### VS2019 C++ Communications Timeout:

Many Dover stages and stage controllers allow multiple communications protocols and use DIP-switches to select which protocol is currently in use. Make sure these switches are properly set, refer to the documentation supplied with the stage or controller to set these, or contact Dover Motion for more information.

Also make sure that your baud rates match for serial and CAN communications. These protocols require the host/device be running at the same data rates.

## Visual Studio 2019 C# Project Setup

### **Finding the files:**

The files referenced in the instructions below should be included on the USB stick that came with the controller/stage. See Definitions Used Within This Guide, USB Stick, Resource locations.

### **VS2019 C# Usage notes:**

C-Motion has been pre-packaged for Windows 32-bit and 64-bit machines into a set of dll files. Two files are needed, a wrapper and the source dll. PMDLibrary.dll and C-Motion.dll are needed to set up a C# program.

Creating a new method should be done using the commands listed below. They will initialize a given device over the selected communications protocol. Be sure to initialize both the peripheral and the device appropriately, then initialize the axis handle with both items. See Figure 42 - VS2019 C# Connection Initialization.

- Peripheral Initialization Options
  - PMDPeripheralCOM
    - This initializes a Point-to-Point serial connection with a single controller
  - PMDPeripheralMultiDrop
    - This initializes a multi-drop serial connection that addresses one of many controllers on a shared network
  - PMDPeripheralCAN
    - This initializes a CAN connection with the controller at a hard-coded 1M baud rate.
- Device Initialization Options
  - PMDDeviceType.MotionProcessor
- Axis Initialization
  - PMDAxisNumber.Axis1 – This is used for most features on the controller
  - PMDAxisNumber.Axis2 – This is used to access the auxiliary encoders and TOP settings

```
periph = new PMD.PMDPeripheralCAN(580, 600, 0);
device = new PMD.PMDDevice(periph, PMD.PMDDeviceType.MotionProcessor);
PMD.PMDAxis axis1 = new PMD.PMDAxis(device, PMD.PMDAxisNumber.Axis1);
```

Figure 42 - VS2019 C# Connection Initialization

Whenever a library C procedure returns a non-zero status code (PMDresult.NO\_ERROR) an exception will be thrown. The data member of the exception will contain a data property with the key "PMDresult".

C-Motion commands that return a single value become class properties and drop the "set" and "get" from the command. Commands can be used as below:

```
pos = axis1.ActualPosition;
```

Signal and event bitmasks for returned status registers can be found in the SDK if needed. See PMDtypes.c within the SDK.



## **VS2019 Starting a New C# Project:**

If build errors are encountered, please read through the VS2019 C# Common Errors section below to reference some common solutions.

- 1) Open MS Visual Studio 2019 and start a new C# Console Application for the .NET Framework 4 per Figure 43 - VS2019 New Project C#.

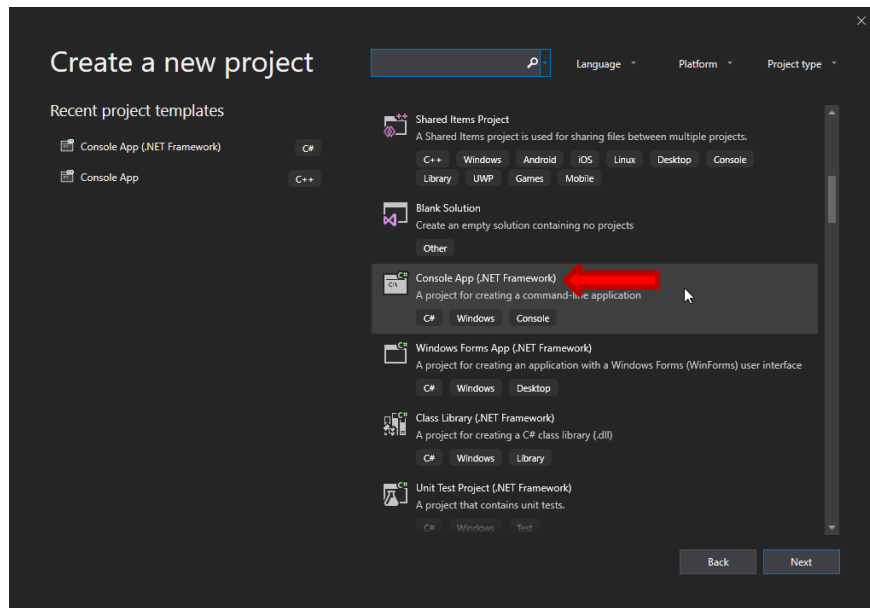


Figure 43 - VS2019 New Project C#

- 2) The appropriate (32bit (x86) or 64bit (x64)) dll files that are needed for the project should be pasted into the source folder of the project, in the same folder as Program.cs and Projectname.csproj, see Figure 44 - VS2019 DLL Location
- a. 32-bit Release:
    - i. ..\ C-Motion.zip\C-Motion\DLLs\Release\x86
      1. PMDLibrary.dll
      2. C-Motion.dll
  - b. 32-bit Debug:
    - i. ..\ C-Motion.zip\C-Motion\DLLs\Debug\x86
      1. PMDLibrary.dll
      2. C-Motion.dll
  - c. 64-bit Release:
    - i. ..\ C-Motion.zip\C-Motion\DLLs\Release\x64
      1. PMDLibrary.dll
      2. C-Motion.dll
  - d. 64-bit Debug:
    - i. ..\ C-Motion.zip\C-Motion\DLLs\Debug\x64
      1. PMDLibrary.dll
      2. C-Motion.dll

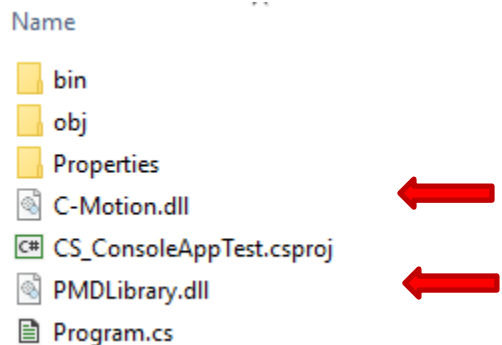


Figure 44 - VS2019 DLL Location

- 3) Right click on References in the Solution Explorer view and select “Add Reference” per Figure 45 - VS2019 Add DLL to Project
  - a. Through the menu select browse and locate the PMDLibrary.dll file just included in the source folder
  - b. Why only include the one file? PMDLibrary.dll is the wrapper called by the program, C-Motion.dll is called by PMDLibrary.dll.

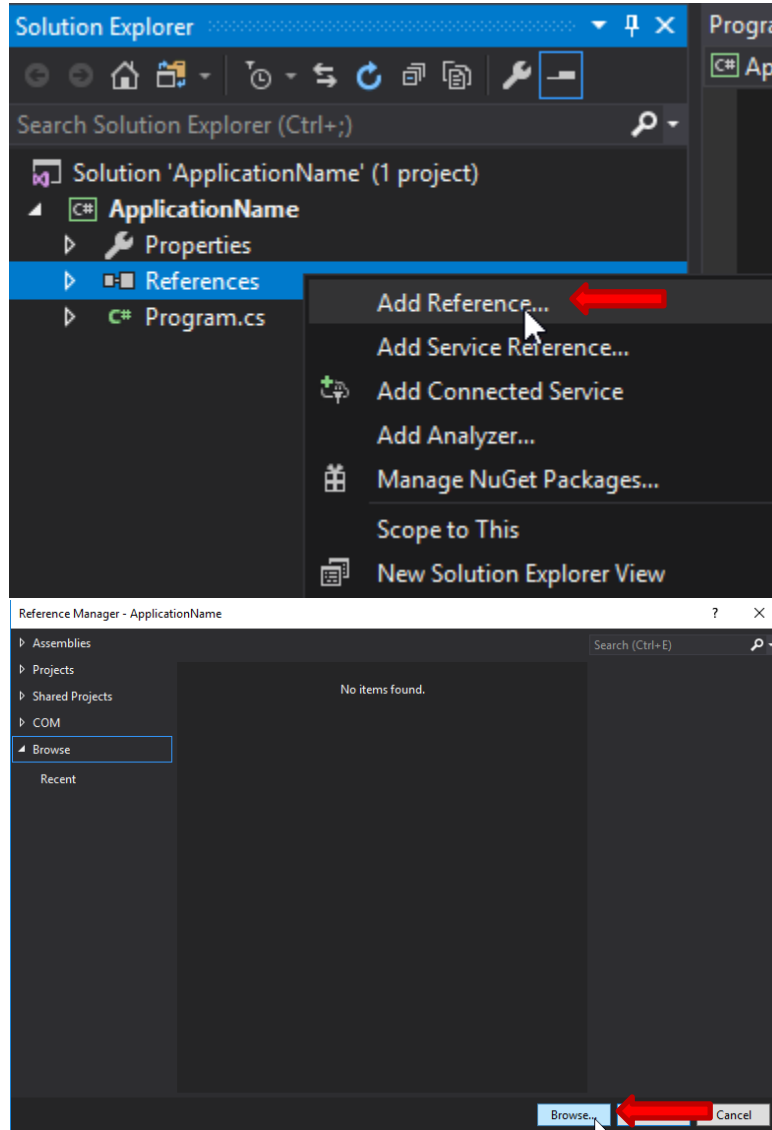


Figure 45 - VS2019 Add DLL to Project

- 4) Open the Debug Configuration Manager, and change the Active Solution Platform to either x86, or x64 depending on the project requirements per Figure 46 - VS2019 Choose Solution Platform

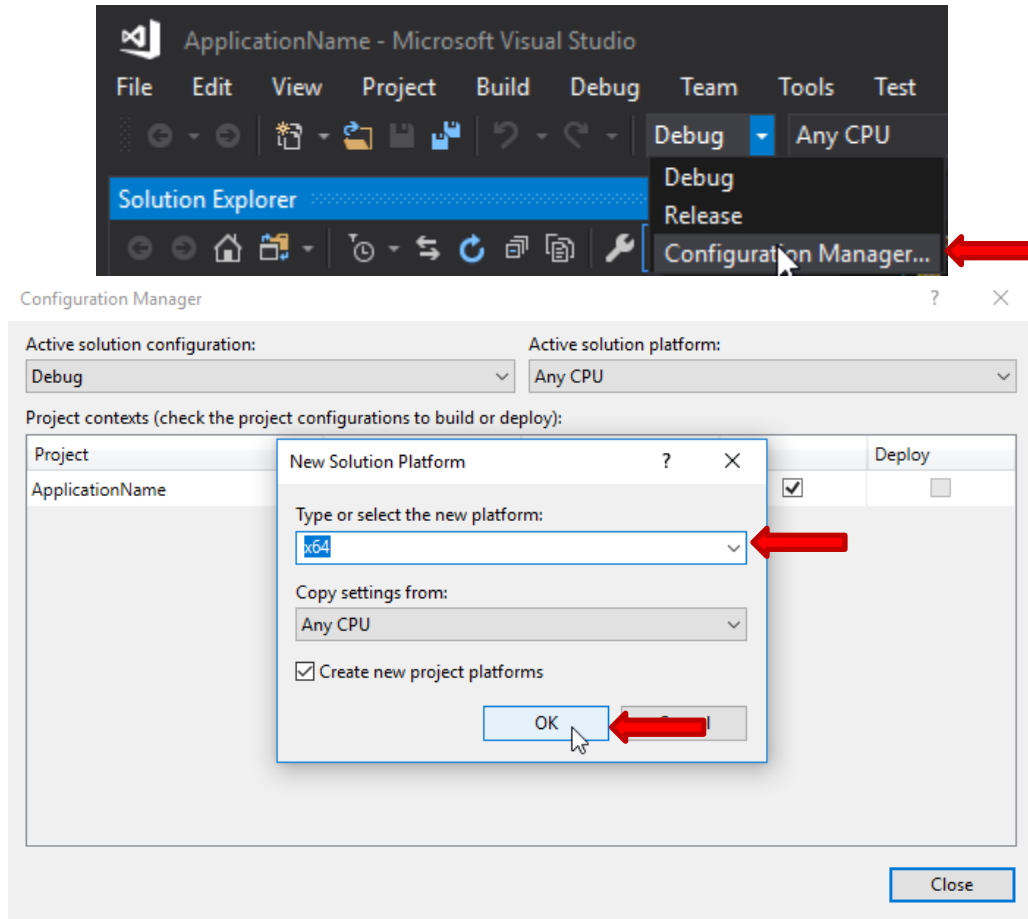


Figure 46 - VS2019 Choose Solution Platform

- 5) Build the solution by choosing Build – Rebuild Solution and navigate to the output folder, make sure that the c-motion.dll file ended up in the output directory, if not, paste them in that location so that the code can find them when running, this is needed for debugging code to run as well. See Figure 47 - VS2019 C# Build Directory
  - a. Default project build/compile locations (depends on build settings, see the output options in the Project Manager -> General -> Output Directory):
    - i. ..\Projectname\bin\x64\Debug
    - ii. ..\Projectname\bin\x64\Release
  - b. C-Motion.dll
  - c. C-Motion.lib
  - d. PlxApi.lib
  - e. PlxApi720.dll
  - f. PMDLibrary.dll
  - g. Vcisdk.lib

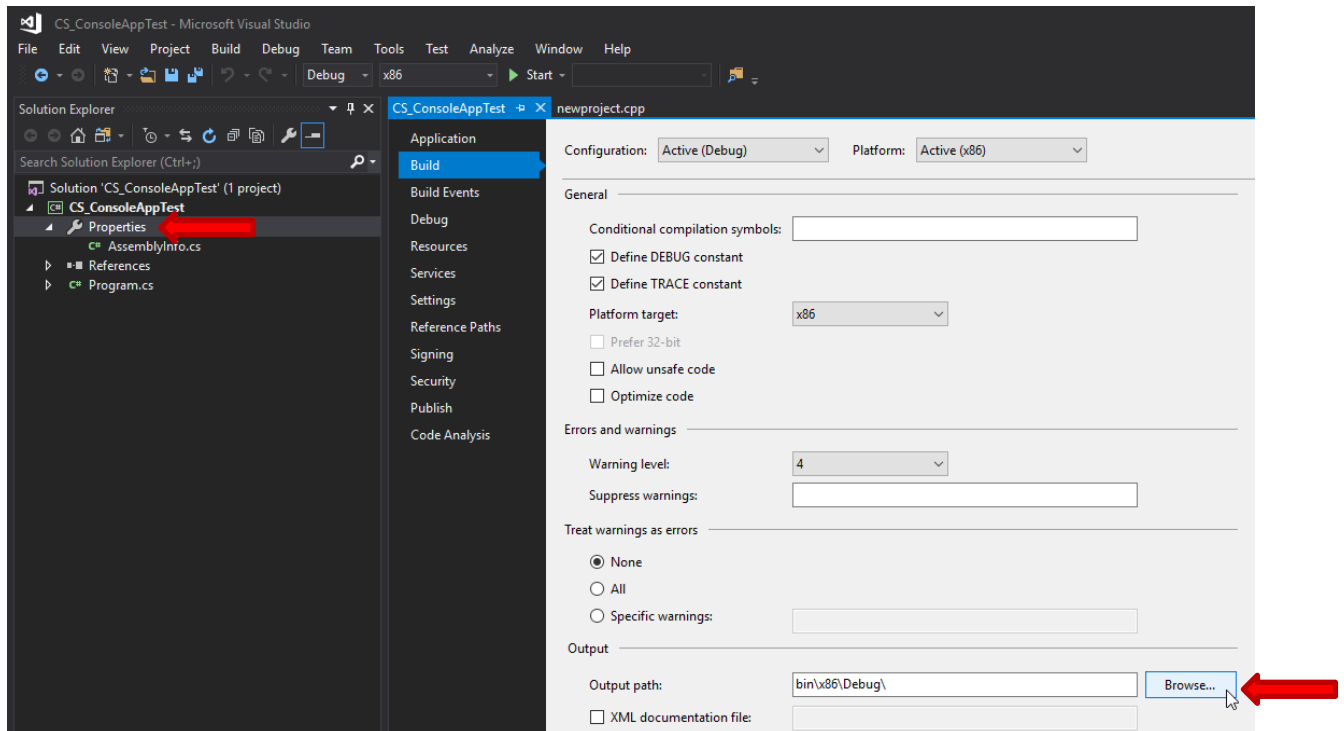


Figure 47 - VS2019 C# Build Directory

- 6) To debug the program, select debug, and either Start Debugging, or Start Without Debugging per Figure 48 - VS2019 Debugging C#

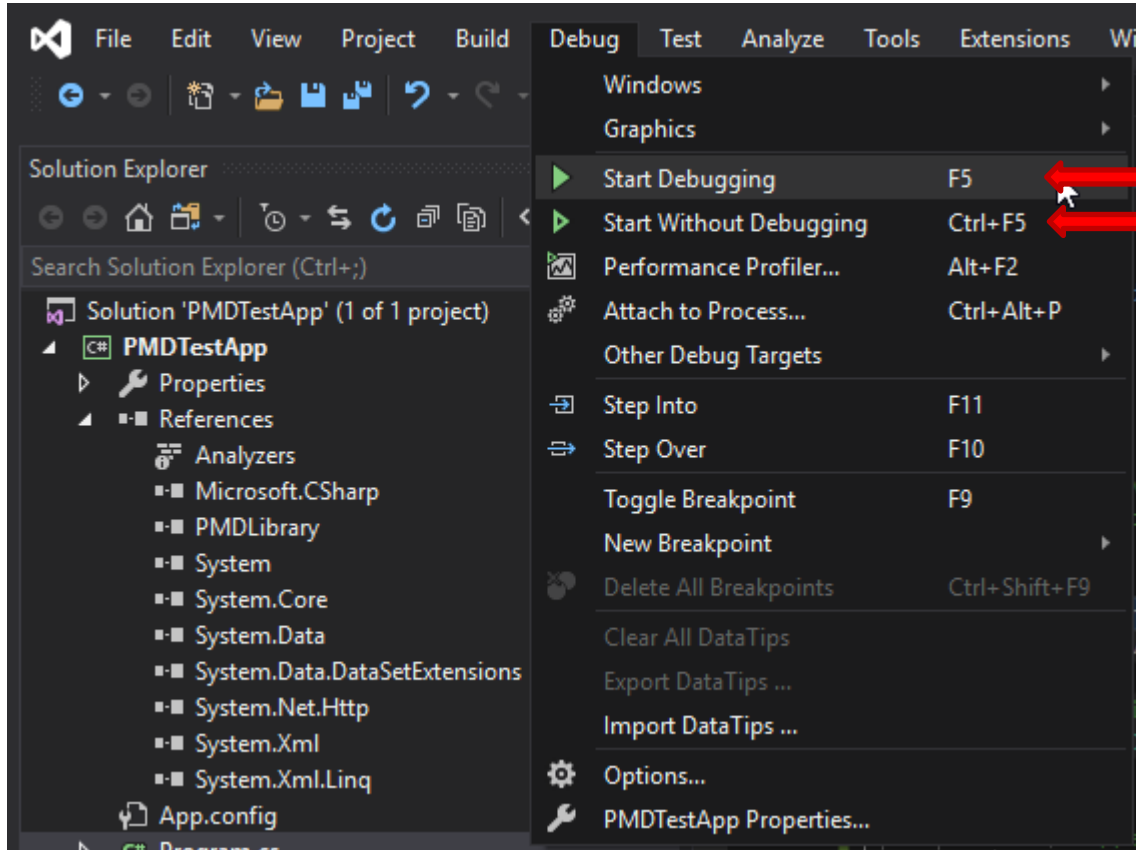


Figure 48 - VS2019 Debugging C#

- 7) The .exe result should run without problems, and should not give any errors for missing files, to troubleshoot, see VS2017 C# Common Errors below.

## VS2019 C# Common Errors

### VS2019 C# DLLs Could Not Be Included:

The wrapper PMDLlibrary.dll is called by the C# program, and C-Motion.dll is called by PMDLlibrary.dll. So only PMDLlibrary.dll needs to be added as a reference to the project. Make sure that c-motion.dll is included in the output directory though, as it is still a dependency for the program.

### VS2019 C# DLLs Could Not Be Found:

Sometimes Visual Studio puts the build output for debugging in the default file repository for all Visual Studio projects, double check that the build output directory is where you think it is – check the full file path per Figure 49 - VS2019 C# Output Directory Verification

Visual Studio can put the output directory in:

C:\Users\Username\source\repos\Projectname

It is fine to work out of this default repository; however, the code will be more manageable if it all stays in the same location. It is advisable to update the build output to stay within the project source folder.

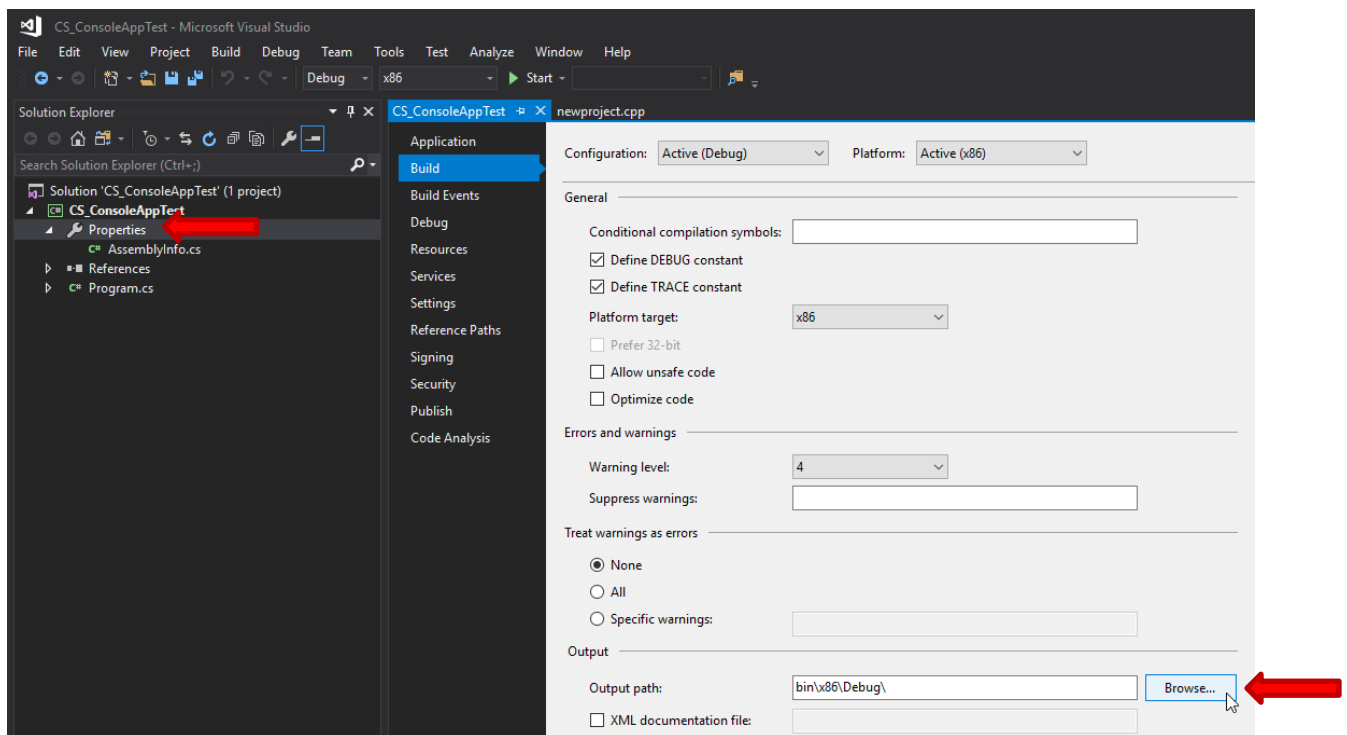



Figure 49 - VS2019 C# Output Directory Verification

 A division of Invetech	<h1>SmartStage Linear System C-Motion Guide</h1>		
Document No. <b>41-1368</b>	Revision: <b>A</b>	Revision Date: <b>09/04/2020</b>	Sheet: <b>56 of 77</b>

### **VS2019 C# 64bit Build Target Error:**

Error message:

An attempt was made to load a program with an incorrect format. (Exception from HRESULT: 0x8007000B)

This error can occur when a 64bit OS project is set to use “Any CPU” in the Configuration Manager. Make sure the build target architecture is properly specified for the build solution, see Figure 50 - VS2019 HRESULT: 0x8007000B

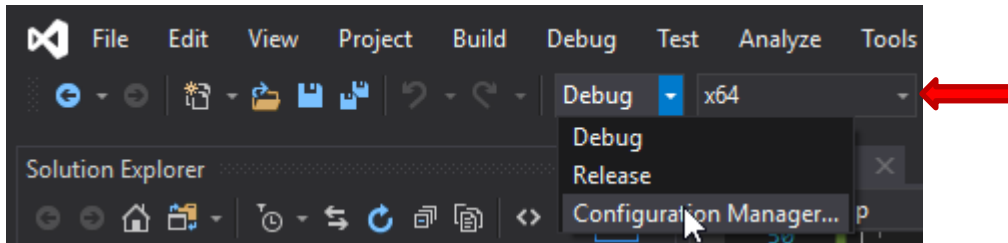


Figure 50 - VS2019 HRESULT: 0x8007000B

### **VS2019 C# Serial Port Connection Error:**

The C-Motion DLLs start COM port numbering at 0, so use the COM port number found in the Device Manager minus one. So, if the stage controller shows up as COM2, enter 1 into the function to initialize the axis.


### **VS2019 C# CAN Communications Timeouts/Errors:**

The DLLs use a fixed baud rate of 1M. The baud rate of the controller, DOF or otherwise must be set to this baud rate to connect properly. Contact Dover for assistance changing the CAN baud rate on your stage controller or modify and rebuild the DLL distribution to allow configuration of the CAN baud rate.

### **VS2019 C# Communications Timeout:**

Many Dover stages and stage controllers allow multiple communications protocols and use DIP-switches to select which protocol is currently in use. Make sure these switches are properly set, refer to the documentation supplied with the stage or controller to set these, or contact Dover Motion for more information.



 A division of Invetech	<h1>SmartStage Linear System C-Motion Guide</h1>		
Document No. <b>41-1368</b>	Revision: <b>A</b>	Revision Date: <b>09/04/2020</b>	Sheet: <b>57 of 77</b>

## Visual Studio 2017 C++ Project Setup

### Finding the files:

The files referenced in the instructions below should be included on the USB stick that came with the controller/stage. See Definitions Used Within This Guide, USB Stick, Resource locations.

### VS2017 C++ Project Notes:

Users must configure the project and compiler properly to get the SDK to build error-free. There are some files that will cause the code not to compile or run depending on the hardware drivers and communications devices installed on the host machine.

When compiling for x64bit machines, the CAN communications .c and .h files must be removed from the project. The files that are not needed are listed below. The CAN libraries are built for x86 (32bit) machines.

- PMDCAN.c
- PMDCAN.h
- PMDIXXATCAN3.c
- PMDIXXATCAN3.h

When not using a communications protocol, users must be certain to exclude files that use libraries and hardware that do not have installed drivers.

### VS2017 Starting a New C++ Project:

If build errors are encountered, please read through the VS2017 C++ Common Errors section below to reference some common solutions.

- 1) Start a new project in Visual Studio 2017, a C/C++ Console App per Figure 51 - VS2017 C++ New Project

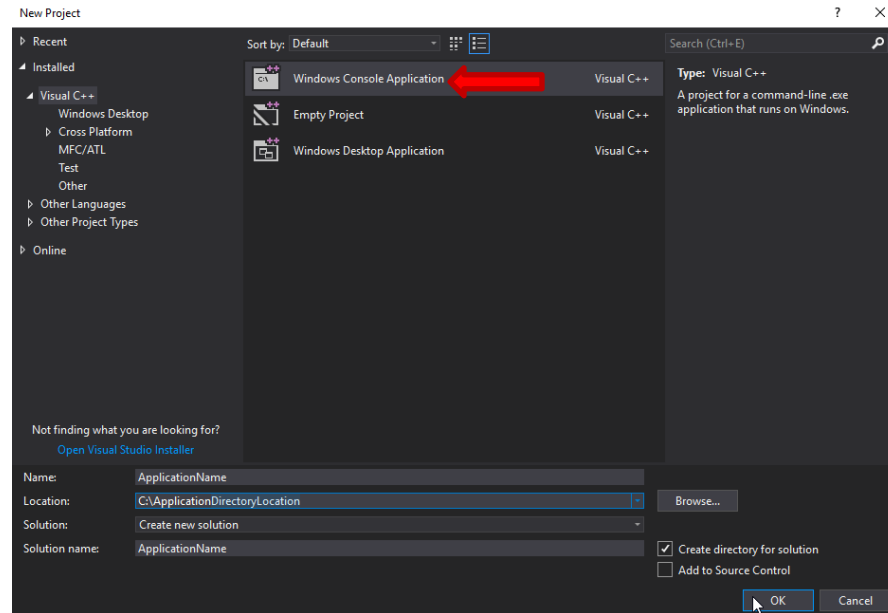


Figure 51 - VS2017 C++ New Project

- 2) Copy and paste local copies of the “C” and “Include” folders from C-Motion.zip inside the project directory, at the same level as the projectname.vcxproj file

- 3) Right click the project in the solution explorer view, select add – existing item and select the following items:
  - a. ..\Include\IXXAT\vcisdsk.lib
  - b. ..\Include\NI\ni845x.lib
  - c. ..\Include\PLX\PlxApi.lib
  - d. All items from ..\Include
- 4) Right click the project in the solution explorer view, select add – existing item and select all the .c files from the C folder.
- 5) Project Properties need to be modified, right click on the project name in the solution explorer view and select properties – be sure to select all configs and all platforms per Figure 52 - VS2017 Project Properties
  - a. General – character set – multibyte character set
  - b. C/C++ - General – SDL Checks – No
  - c. C/C++ - Code Generation – Runtime Library – Multi-threaded Debug (/MTd)
    - i. This setting is debug/release dependent, so if a list of errors linker errors comes up when compiling, be sure this settings is /MTd for a debug build, and /MT for a release build
  - d. C/C++ - Precompiled Headers - Not using precompiled headers

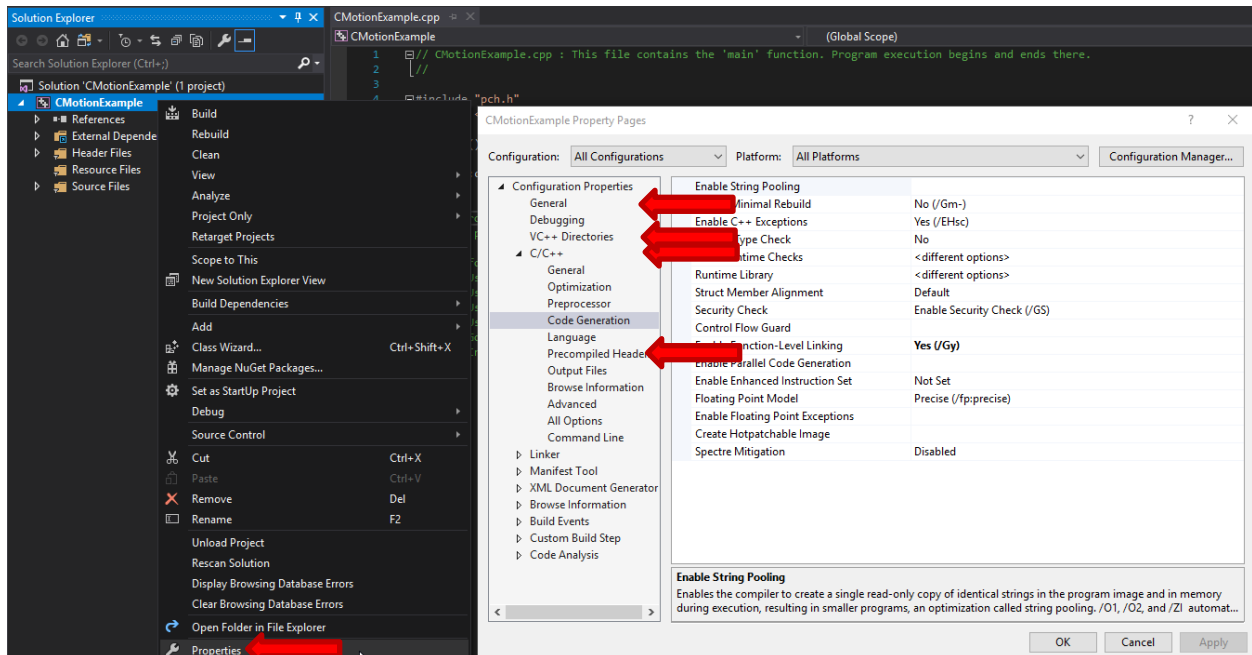


Figure 52 - VS2017 Project Properties

- e. VC++ directories – include directories – add the following folders per Figure 53 - VS2017 VC++ Directories

- ix. ..\Include
- x. ..\Include\IXXAT
- xi. ..\Include\PLX
- xii. ..\Include\NI

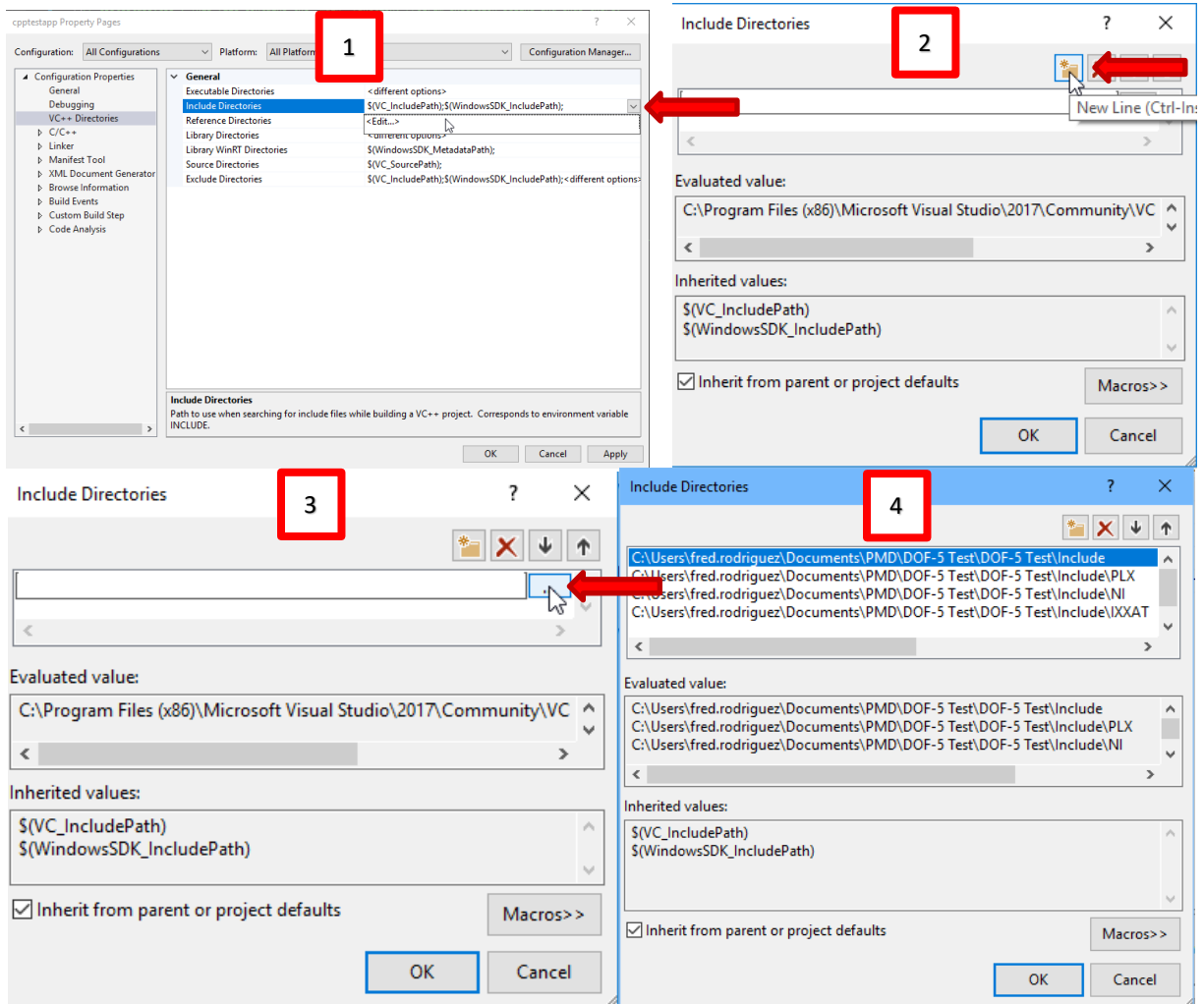


Figure 53 - VS2017 VC++ Directories

- 6) Exclude the .c files that have uninstalled dependencies or required devices. The following interfaces have device specific driver requirements and will not compile unless those drivers are installed. See Figure 54 – VS2017 Removing Uninstalled Dependencies
- c. PMDNISPI.c
    - xiii. For use with SPI communications
    - xiv. Install the drivers for an NI845x device
  - d. PMDCAN.c
    - xv. For use with CAN communications
    - xvi. Install the drivers for an IXXATUSB-to-CAN adapter

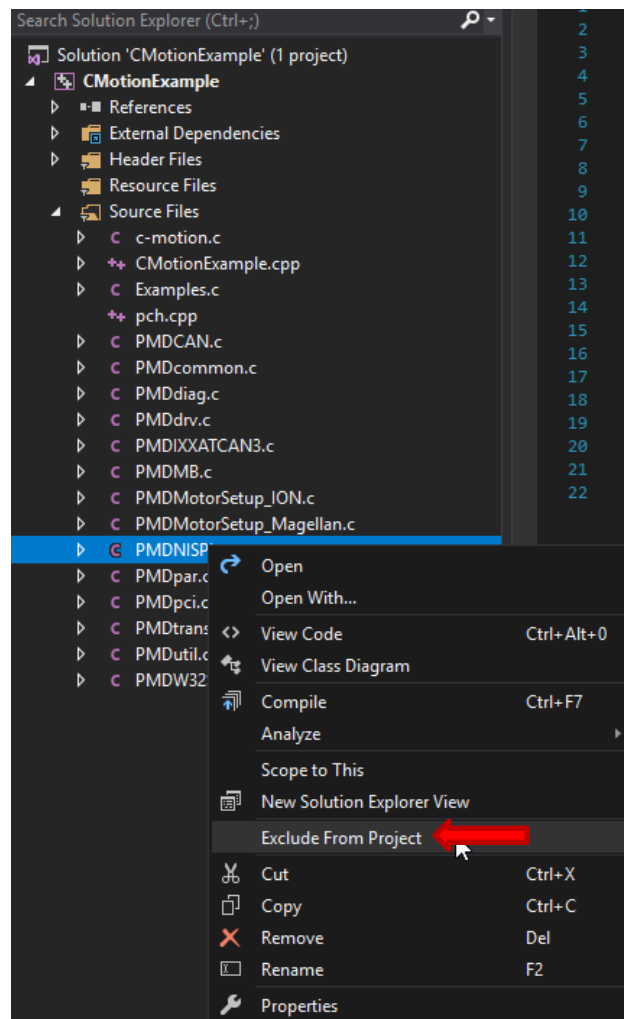


Figure 54 – VS2017 Removing Uninstalled Dependencies

- 7) Build the solution to ensure that everything has linked correctly per Figure 55 - VS2017 Rebuild
  - a. See VS2017 C++ Common Errors below to troubleshoot errors when building the project solution. VCINLP.dll is a common CAN driver problem, SAFESEH is a linker exception handler problem, etc.

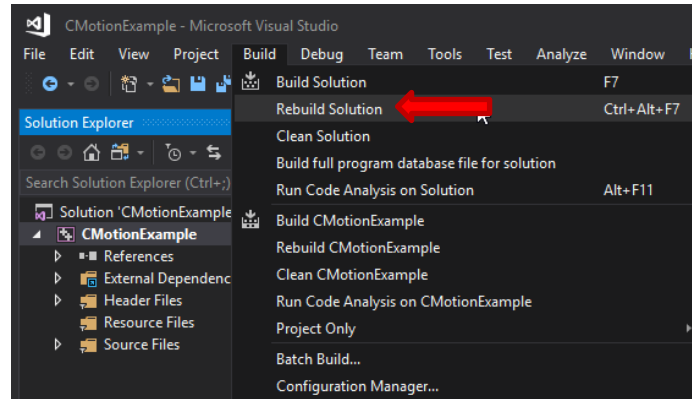


Figure 55 - VS2017 Rebuild

- 8) The program should now be set up, and ready to run, the “main entry point” to the program, is projectname.c
- 9) Running the program can be done by selecting the debug menu, and selecting either “Start Debugging” or “Start Without Debugging”, see Figure 56 - VS2017 Run Debugging
  - c. Begin programming the stage control program or continue reading for where to find example code to start moving the stage with. Note that the examples are explained below the instructions on how to use them, and each example includes well documented functions and setup code.

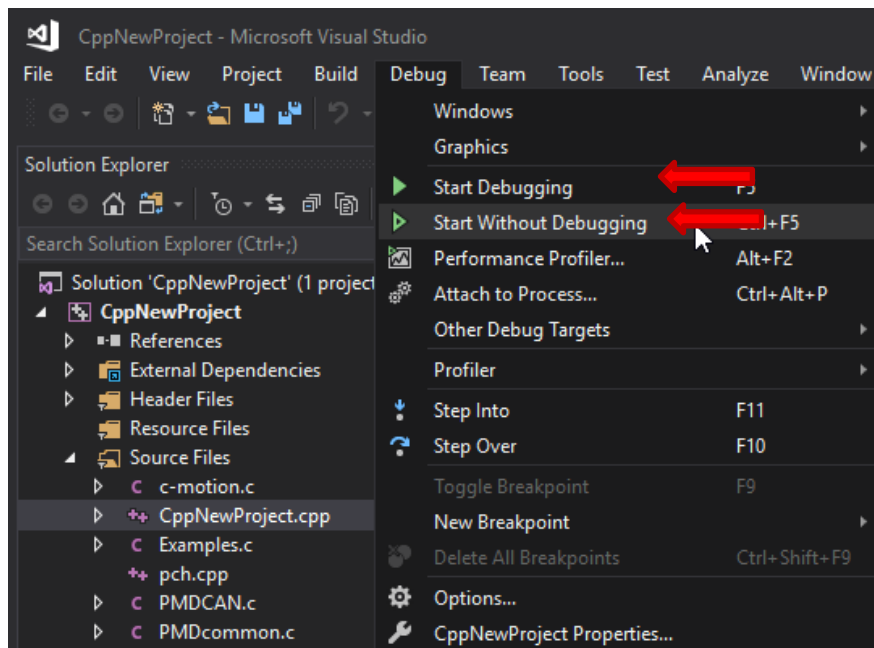


Figure 56 - VS2017 Run Debugging

- 10) Once the project has been “Built” or compiled, it can also be run by using the generated “projectname.exe” file found at the debug output location, the typical default is below, and can also be found by browsing through project properties, in “General” and selecting the output directory, per Figure 57 - VS2017 Output Directory
- d. ..\projectname\Debug

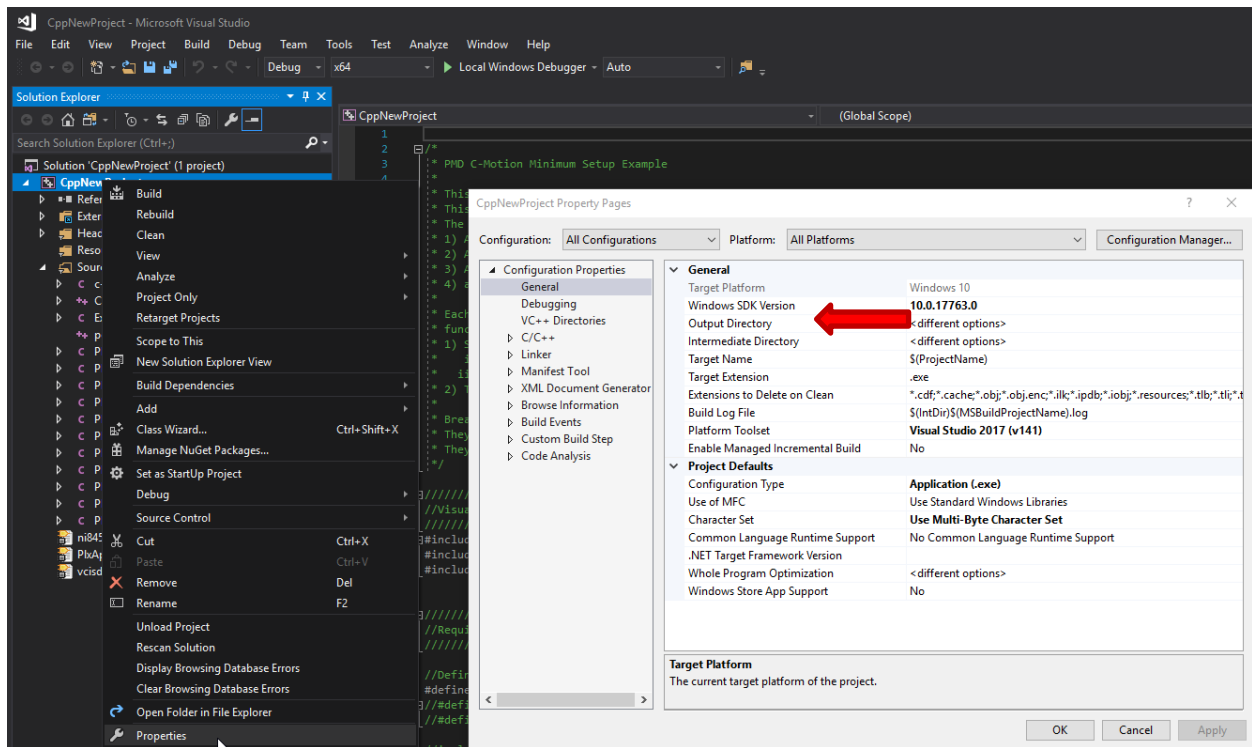


Figure 57 - VS2017 Output Directory

### VS2017 C++ Common Errors

#### VS2017 C++ SAFESEH Warning:

If the build is throwing a SAFESEH warning, or Link 2026 errors, then change the “Image has safe exception handlers” setting in the project properties window under Linker -> Advanced, down at the bottom of the list per Figure 58 - VS2017 SAFESEH Warning

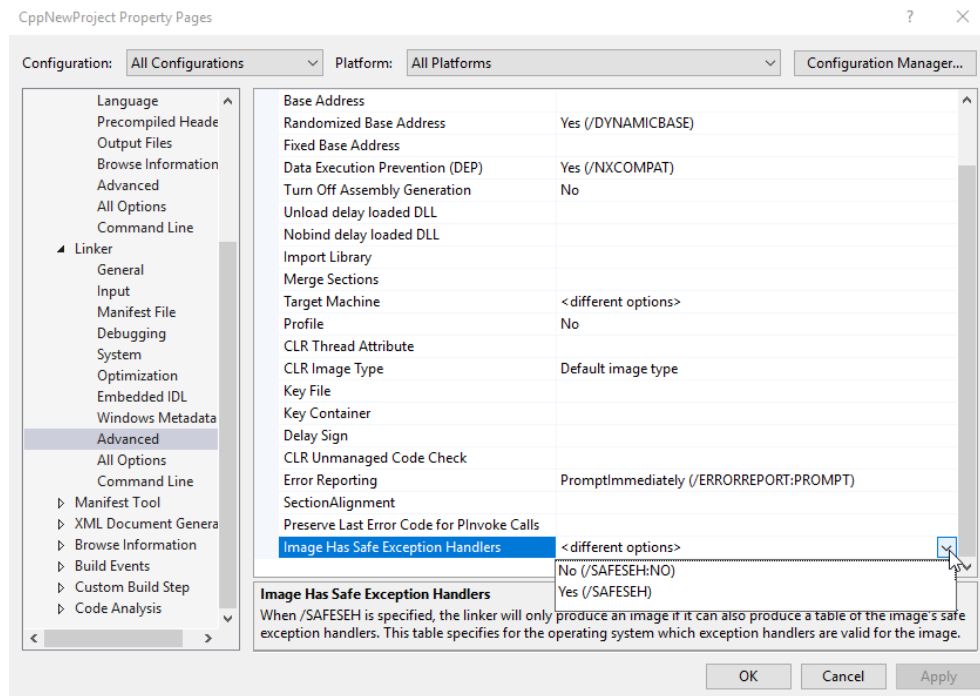


Figure 58 - VS2017 SAFESEH Warning

#### VS2017 C++ Missing Drivers:

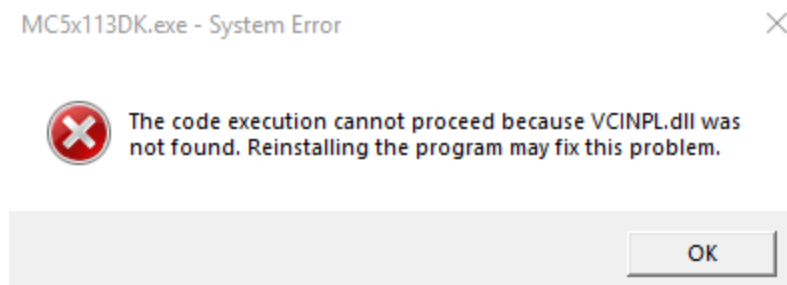


Figure 59 - VS2017 Missing Driver Error

This error pops up when the correct drivers are not installed for the USB device needed, specifically for the IXXAT USB-to-CAN adapter. Ensure that the drivers for the devices in use are installed or remove the dependent files that require the drivers. See Figure 59 - VS2017 Missing Driver Error.

## VS2017 C++ Runtime Library Errors:

A long string of link errors in the build output could likely indicate an incorrect selection for the runtime library. Note that this selection is build dependent, and must be either debug or release, otherwise a long string of build errors can occur. See Figure 60 - VS2017 Linker Error

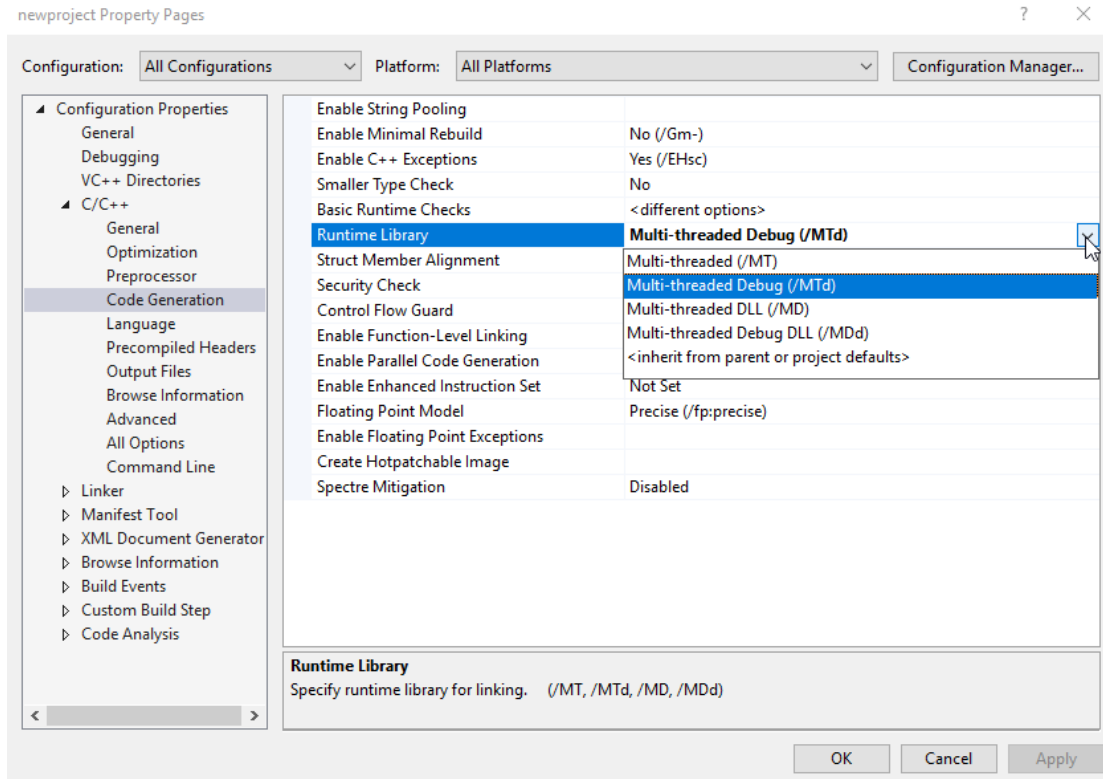


Figure 60 - VS2017 Linker Error

## VS2017 C++ Communications Timeout:

Many Dover stages and stage controllers allow multiple communications protocols and use DIP-switches to select which protocol is currently in use. Make sure these switches are properly set, refer to the documentation supplied with the stage or controller to set these, or contact Dover Motion for more information.

Also make sure that your baud rates match for serial and CAN communications. These protocols require the host/device be running at the same data rates.



## Visual Studio 2017 C# Project Setup

### *Finding the files:*

The files referenced in the instructions below should be included on the USB stick that came with the controller/stage. See Definitions Used Within This Guide, USB Stick, Resource locations.

### *VS2017 C# Usage notes:*

C-Motion has been pre-packaged for Windows 32-bit and 64-bit machines into a set of dll files. Two files are needed, a wrapper and the source dll. PMDLibrary.dll and C-Motion.dll are needed to set up a C# program.

Creating a new method should be done using the commands listed below. They will initialize a given device over the selected communications protocol. Be sure to initialize both the peripheral and the device appropriately, then initialize the axis handle with both items. See Figure 61 - VS2017 C# Connection Initialization.

- Peripheral Initialization Options
  - PMDPeripheralCOM
    - This initializes a Point-to-Point serial connection with a single controller
  - PMDPeripheralMultiDrop
    - This initializes a multi-drop serial connection that addresses one of many controllers on a shared network
  - PMDPeripheralCAN
    - This initializes a CAN connection with the controller at a hard-coded 1M baud rate.
- Device Initialization Options
  - PMDDeviceType.MotionProcessor
- Axis Initialization
  - PMDAxisNumber.Axis1 – This is used for most features on the controller
  - PMDAxisNumber.Axis2 – This is used to access the auxiliary encoders and TOP settings

```
periph = new PMD.PMDPeripheralCAN(580, 600, 0);
device = new PMD.PMDDevice(periph, PMD.PMDDeviceType.MotionProcessor);
PMD.PMDAxis axis1 = new PMD.PMDAxis(device, PMD.PMDAxisNumber.Axis1);
```

Figure 61 - VS2017 C# Connection Initialization

Whenever a library C procedure returns a non-zero status code (PMDresult.NO\_ERROR) an exception will be thrown. The data member of the exception will contain a data property with the key "PMDresult".

C-Motion commands that return a single value become class properties and drop the "set" and "get" from the command. Commands can be used as below:

```
pos = axis1.ActualPosition;
```

Signal and event bitmasks for returned status registers can be found in the SDK if needed. See PMDtypes.c within the SDK.

## VS2017 Starting a New C# Project:

If build errors are encountered, please read through the VS2017 C# Common Errors section below to reference some common solutions.

- 8) Open MS Visual Studio 2017 and start a new C# Console Application for the .NET Framework 4 per Figure 62 - VS2017 New Project C#

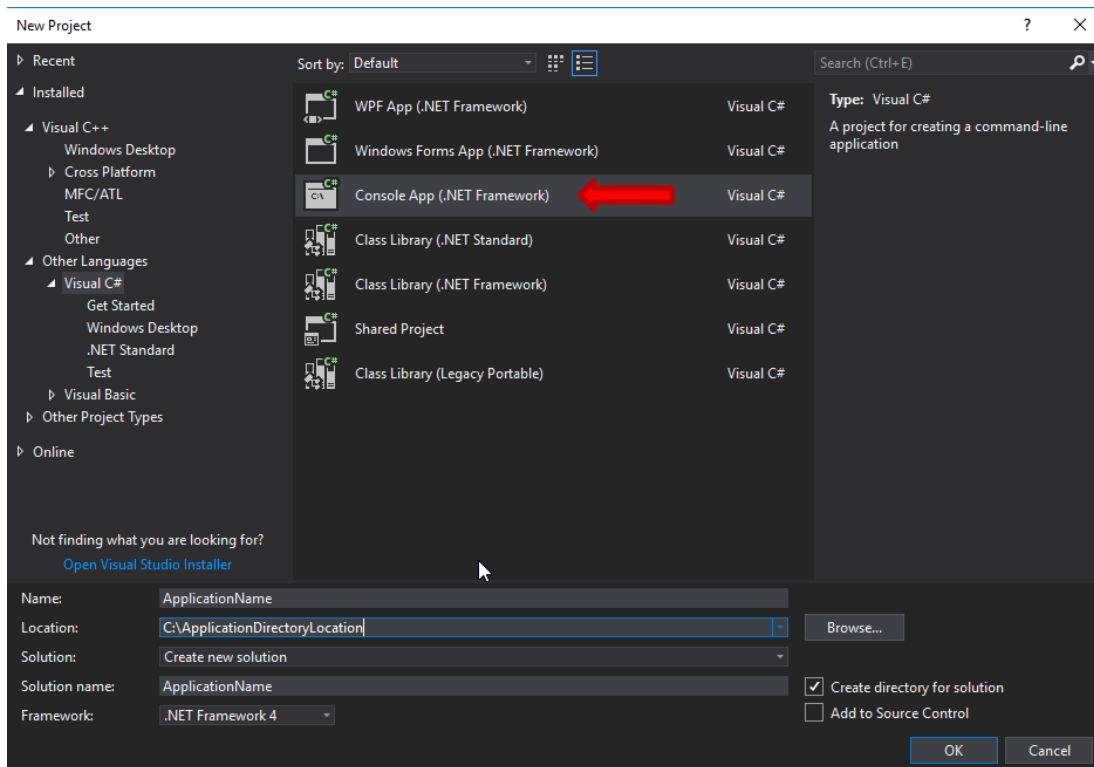


Figure 62 - VS2017 New Project C#

9) The appropriate (32bit (x86) or 64bit (x64)) dll files that are needed for the project should be pasted into the source folder of the project, in the same folder as Program.cs and Projectname.csproj, see Figure 63 - VS2017 DLL Location

a. 32-bit Release:

i. ..\ C-Motion.zip\C-Motion\DLLs\Release\x86

1. PMDLibrary.dll
2. C-Motion.dll

b. 32-bit Debug:

i. ..\ C-Motion.zip\C-Motion\DLLs\Debug\x86

1. PMDLibrary.dll
2. C-Motion.dll

c. 64-bit Release:

i. ..\ C-Motion.zip\C-Motion\DLLs\Release\x64

1. PMDLibrary.dll
2. C-Motion.dll

d. 64-bit Debug:

i. ..\ C-Motion.zip\C-Motion\DLLs\Debug\x64

1. PMDLibrary.dll
2. C-Motion.dll

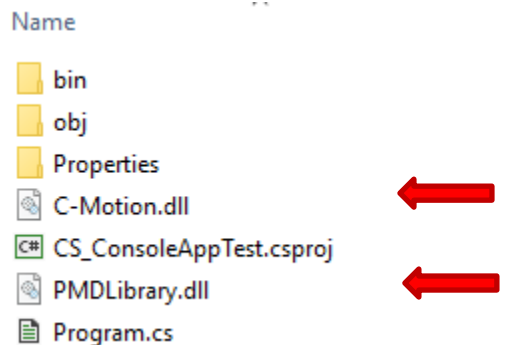


Figure 63 - VS2017 DLL Location

10) Right click on References in the Solution Explorer view and select “Add Reference” per Figure 64 - VS2017  
Add DLL to Project

- a. Through the menu select browse and locate the PMDLibrary.dll file just included in the source folder
- b. Why only the one file? PMDLibrary.dll is called by the program, C-Motion.dll is called by PMDLibrary.dll.

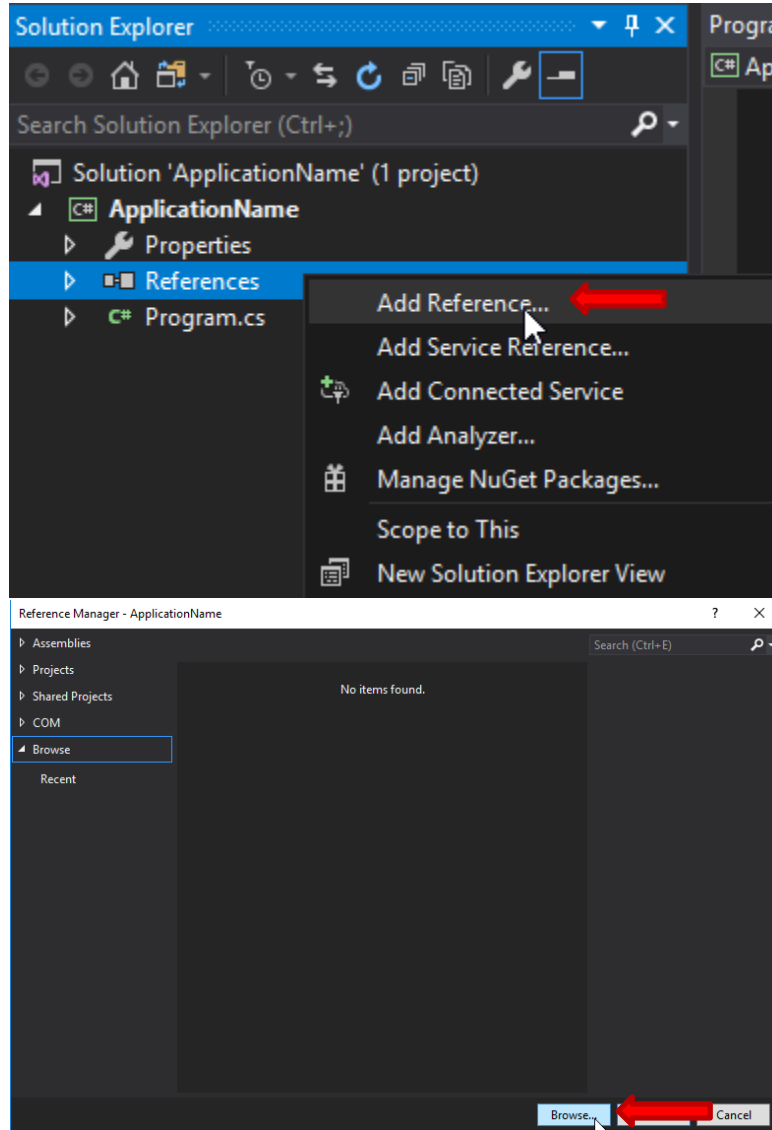


Figure 64 - VS2017 Add DLL to Project

- 11) Open the Debug Configuration Manager, and change the Active Solution Platform to either x86, or x64 depending on the project requirements per Figure 65 - VS2017 Choose Solution Platform.

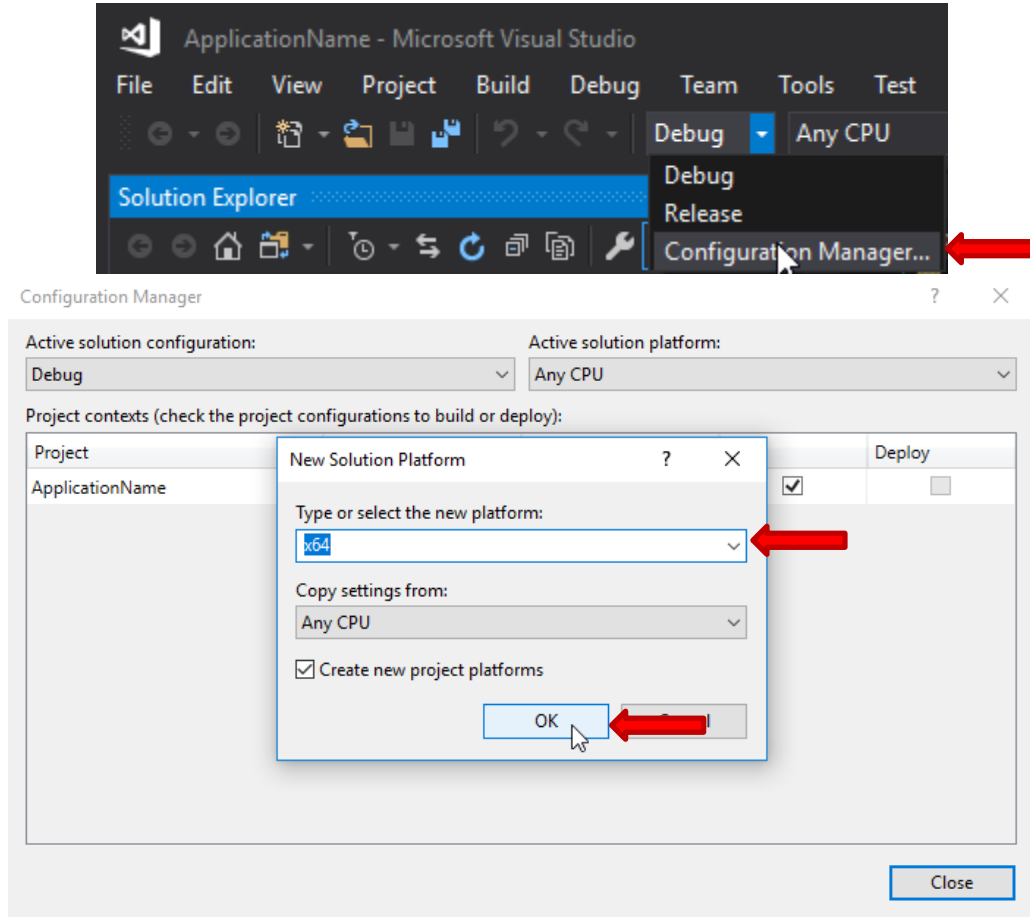


Figure 65 - VS2017 Choose Solution Platform

12) Build the solution by choosing Build – Rebuild Solution and navigate to the output folder, make sure that the dll files ended up in the output directory, if not, paste them in that location so that the code can find them when running, this is needed for debugging code to run as well. See Figure 66 - VS2017 C# Build Directory

- a. Default project build/compile locations (depends on build settings, see the output options in the Project Manager -> General -> Output Directory):
  - i. ..\Projectname\bin\x64\Debug
  - ii. ..\Projectname\bin\x64\Release
- b. C-Motion.dll
- c. C-Motion.lib
- d. PlxApi.lib
- e. PlxApi720.dll
- f. PMDLibrary.dll
- g. Vcisdk.lib

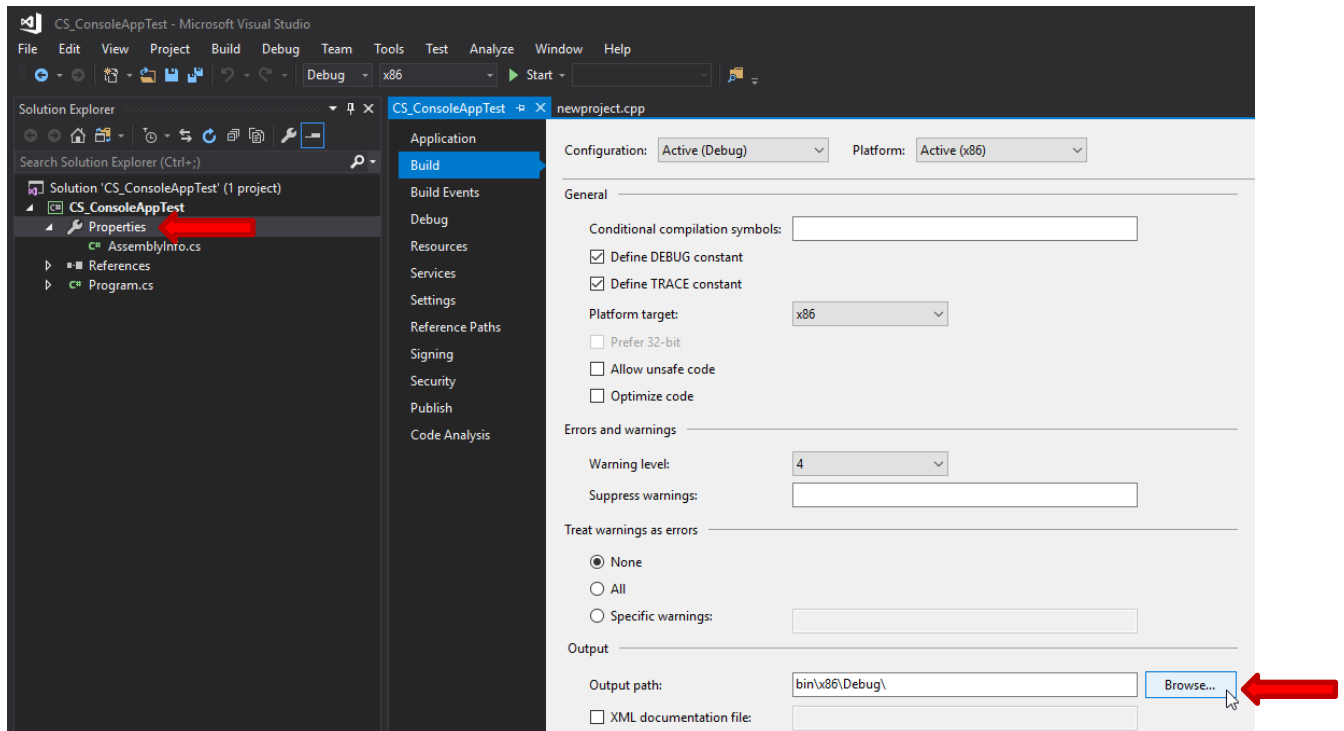


Figure 66 - VS2017 C# Build Directory

- 13) To debug the program, select debug, and either Start Debugging, or Start Without Debugging per Figure 67 - VS2017 Debugging C#

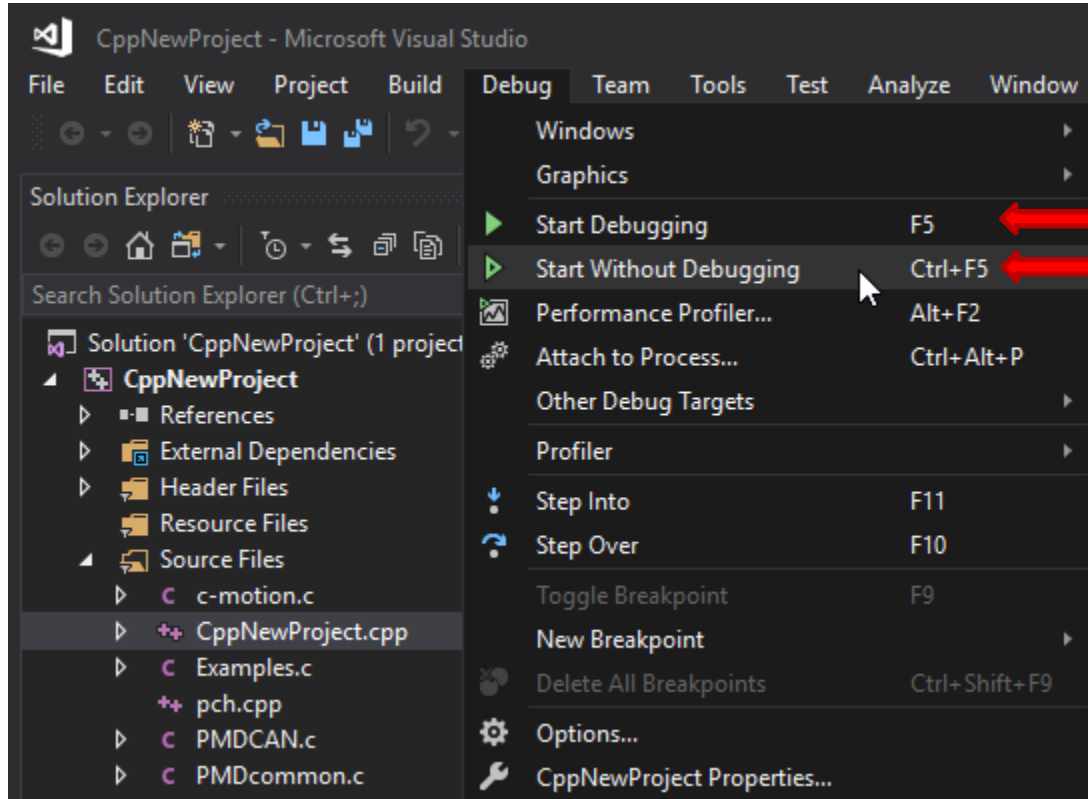


Figure 67 - VS2017 Debugging C#

- 14) The .exe result should run without problems, and should not give any errors for missing files, to troubleshoot, see VS2017 C# Common Errors below.

## VS2017 C# Common Errors

### VS2017 C# DLLs Could Not Be Included:

PMDLibrary.dll is called by the C# program, and C-Motion.dll is called by PMDLibrary.dll. So only PMDLibrary.dll needs to be added as a reference to the project.

### VS2017 C# DLLs Could Not Be Found:

Sometimes Visual Studio puts the build output for debugging in the default file repository for all Visual Studio projects, double check that the build output directory is where you think it is – check the full file path per Figure 68 - VS2017 C# Output Directory Verification.

Visual Studio can put the output directory in:  
C:\Users\Username\source\repos\Projectname

It is fine to work out of this default repository; however, the code will be more manageable if it all stays in the same location. It is advisable to update the build output to stay within the project source folder.

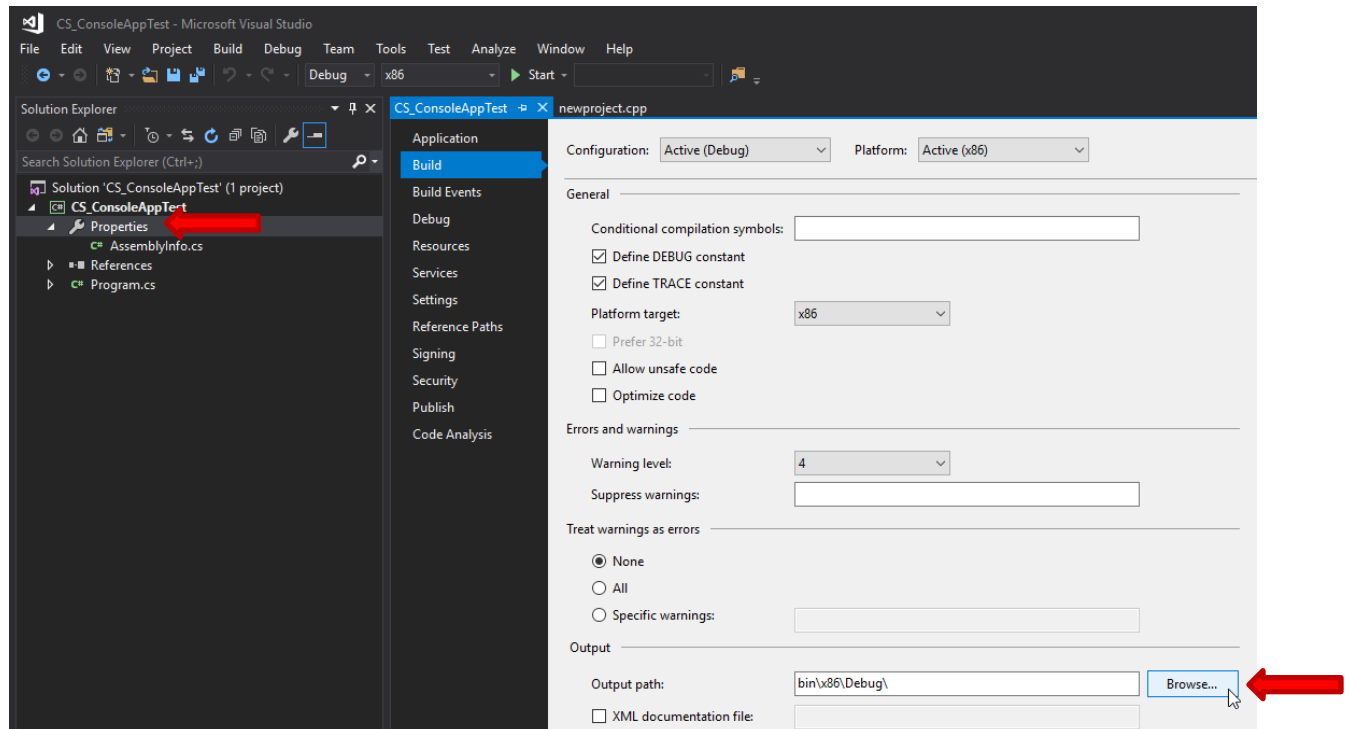



Figure 68 - VS2017 C# Output Directory Verification



 A division of Invetech	<b>SmartStage Linear System C-Motion Guide</b>		
Document No. <b>41-1368</b>	Revision: <b>A</b>	Revision Date: <b>09/04/2020</b>	Sheet: <b>73 of 77</b>

#### **VS2017 C# 64bit Build Target Error:**

Error message:

An attempt was made to load a program with an incorrect format. (Exception from HRESULT: 0x8007000B)

This error can occur when a 64bit OS project is set to use “Any CPU” in the Configuration Manager. Make sure the build target architecture is properly specified for the build solution, see Figure 69 - VS2017 HRESULT: 0x8007000B.

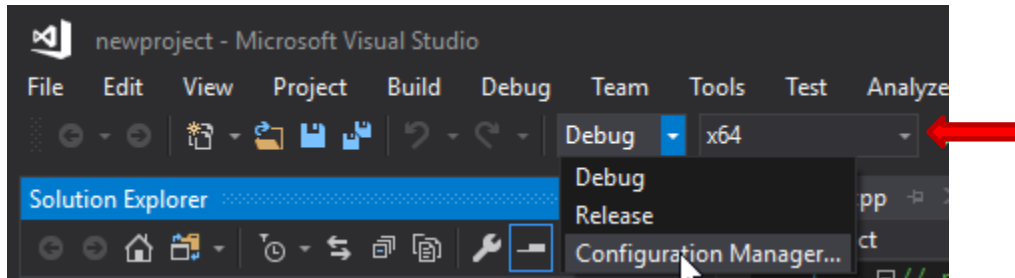


Figure 69 - VS2017 HRESULT: 0x8007000B

#### **VS2017 C# Serial Port Connection Error:**

The C-Motion DLLs start COM port numbering at 0, so use the COM port number found in the Device Manager minus one. So, if the stage controller shows up as COM2, enter 1 into the function to initialize the axis.

#### **VS2017 C# CAN Communications Timeouts/Errors:**

The DLLs use a fixed baud rate of 1M. The baud rate of the controller, DOF or otherwise must be set to this baud rate to connect properly. Contact Dover for assistance changing the CAN baud rate on your stage controller, or modify and rebuild the DLL distribution to allow configuration of the CAN baud rate.

#### **VS2017 C# Communications Timeout:**

Many Dover stages and stage controllers allow multiple communications protocols and use DIP-switches to select which protocol is currently in use. Make sure these switches are properly set, refer to the documentation supplied with the stage or controller to set these, or contact Dover Motion for more information.

## Original Source Code

### The SDK

The SDK (Source Development Kit) contains all the source code that C-Motion utilizes, and allows users to make edits and update the code as needed. Dover has put together some examples on how to use the libraries, and has a slightly modified distribution from the original source (added configuration options for communications settings primarily).

The original source code can be installed to the Documents folder on a Windows machine by running (as Admin) PMDSDK550.exe. This will install the source files to the folder location listed below. The original source supports all the original communications methods, PCIE cards, SPI, CAN, Windows and Linux Serial. Missing dependencies/drivers will prevent the source code from compiling currently, so unused or uninstalled dependencies should be removed.

...\Documents\PMD\SDK

### The DLLs

The C-Motion DLLs include support for a variety of features and communications protocols. They are provided pre-compiled for Windows 10 machines, and the source code is provided by an installation file.

There are several features that may become cumbersome dependencies for those trying to minimize a code footprint, and these can be selectively removed from the project as needed. There exists by default support for serial, CAN and PCI card drivers. Most applications only require one communications protocol, so the unused code may be removed, and the libraries rebuilt. Instructions for rebuilding the DLLs can be found in the section: Rebuilding the DLLs.

Importing these DLLs often requires a header file. The header files that need to be used with these DLLs are included with the C-Motion source for the DLLs. It is best to take these header files from the CMESDK192.exe file drop location (found below). This should help import the DLLs into most other tools and software packages that support them like LabVIEW and Matlab.

...\Documents\PMD\CMESDK\C-Motion\Include

### Rebuilding the DLLs

- 1) If the CMESDK192.exe has already been installed, just delete the PMD folder contents.
  - a. Go to Documents/PMD and delete everything
- 2) Run CMESDK192.exe as an administrator
- 3) Navigate to the installed folder location at Documents/PMD/CMESDK/HostCode/DLLBuild
- 4) Open the Visual Studio project file DLLBuild.sln (See Figure 70 – DLL File Structure)

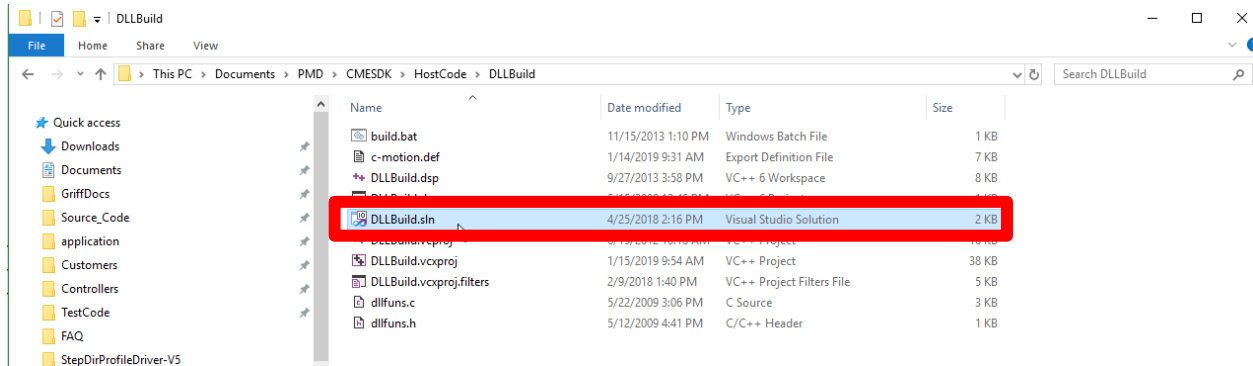


Figure 70 – DLL File Structure

- 5) Visual Studio will ask to retarget the solution, select “OK” (See Figure 71 - Visual Studio Retargeting Solution)

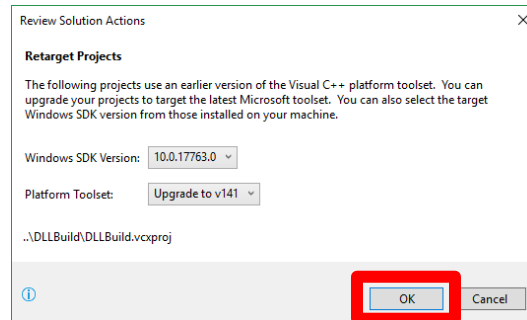


Figure 71 - Visual Studio Retargeting Solution

- 6) Select the build conditions that are needed (debug/release and 32/64 bit) (See Figure 72 - Build Conditions)

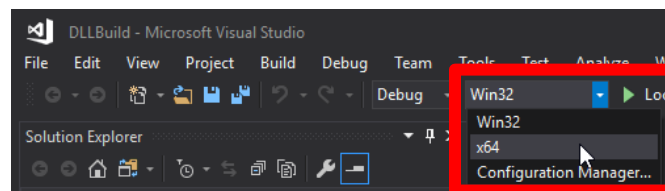


Figure 72 - Build Conditions

- 7) Build the project
  - a. The configured project build directory is Documents/PMD/CMESDK/HostCode/...
  - b. Where “...” is Debug or Release depending on the build settings

- 8) Take the output files (listed below) and put them in the project source folder, include PMDLlibrary.dll in the project references (C-Motion.dll is referenced by PMDLlibrary.dll and should not be included as a reference)
- a. PMDLlibrary.dll - include this in the project
  - b. CoMotion.dll - don't include this in the project

A common build error is that Visual Studio can't find plxapi.lib or vciapi.lib, these can be found in:

Documents/PMD/CMESDK/C-Motion/Include/PLX

Documents/PMD/CMESDK/C-Motion/Include/IXXAT

Take these files (plxapi.lib and vciapi.lib) and put them into the output folder for the respective build that you are trying to compile – these are located at:

Documents/PMD/CMESDK/HostCode/Release

Documents/PMD/CMESDK/HostCode/Debug



## SmartStage Linear System C-Motion Guide

Document No. **41-1368**

Revision: **A**

Revision Date: **09/04/2020**

Sheet: **77 of 77**

### ***Review/Revision History***

Revision	Date	Summary	ECO Number	Writer/Reviser
A	09/04/20	Initial Release	DM11185	Griffin Whittredge