VSI Mach4 Registers
User Guide

Document Revision 1.01

(Updated January 24, 2019)

Atlanta, GA USA

For more information please visit the product web page:

www.vitalsystem.com/motion/sdk

Contents

License Agreement

Before using this software, please take a moment to go thru this License agreement. Any use of this software indicate your acceptance to this agreement.

It is the nature of all machine tools that they are dangerous devices. In order to be permitted to use this software on any machine you must agree to the following license:

I agree that no-one other than the owner of this machine, will, under any circumstances be responsible, for the operation, safety, and use of this machine. I agree there is no situation under which I would consider Vital Systems, or any of its distributors to be responsible for any losses, damages, or other misfortunes suffered through the use of this software. I understand that this software is very complex, and though the engineers make every effort to achieve a bug free environment, that I will hold no-one other than myself responsible for mistakes, errors, material loss, personal damages, secondary damages, faults or errors of any kind, caused by any circumstance, any bugs, or any undesired response by the board and its software while running my machine or device.

I fully accept all responsibility for the operation of this machine while under the control of this software, and for its operation by others who may use the machine. It is my responsibility to warn any others who may operate any device under the control of DSPMC board of the limitations so imposed.

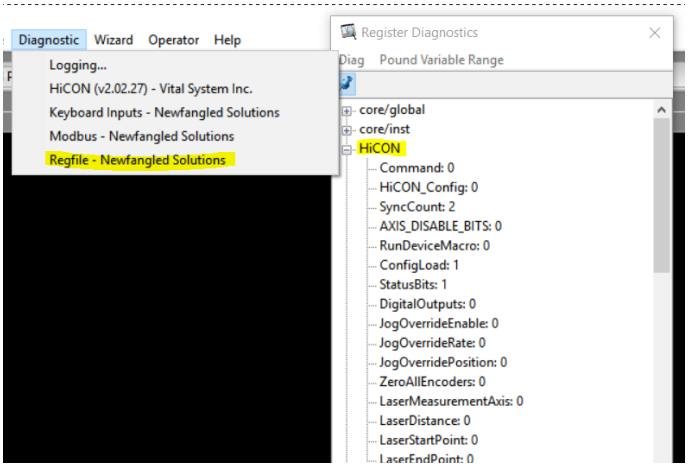
I fully accept the above statements, and I will comply at all times with standard operating procedures and safety requirements pertinent to my area or country, and will endeavor to ensure the safety of all operators, as well as anyone near or in the area of my machine.

WARNING: Machines in motion can be extremely dangerous! It is the responsibility of the user to design effective error handling and safety protection as part of the system. VITAL Systems shall not be liable or responsible for any incidental or consequential damages. By Using the VSI motion controller, you agree to the license agreement.

Introduction

Mach4 software allows the creation of globally accessible variables called registers. These registers can be accessed and changed using the scripting functionality in the screen or script editor built into Mach4. The VSI Motion Plugin contains registers designed for use with specific functions.

The registers of the VSI Motion controller can be manually viewed and changed by accessing the Mach4 Register Diagnostics window.



The full name of a VSI device register is the register prefix followed by the register name.

The register prefix is the name of the VSI Motion Controller (ie, HiCON, DSPMC).

Values in a register are written to or read from, depending on the use case. Some registers are read only.

To learn more about editing the screen in Mach4, download the Mach4 CNC Controller Screen Editing Guide here.

VSI Mach4 Registers

These registers are globally accessible in Mach4 and were created for specific functionality related to VSI motion controllers.

Command

Send custom commands to the VSI motion controller. Some commands return a result. Some commands accept a value followed by a ':' after the command. It is recommended to clear the command register by sending an empty string before sending a command.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
mc.mcRegSendCommand(commandReg, "")
```

Available Commands:

- **MOTION_SYNC** – Syncs the motor position between Mach4 planner and motion controller. This script will only work if Mach4 is in an idle state, ie no motion occurring.

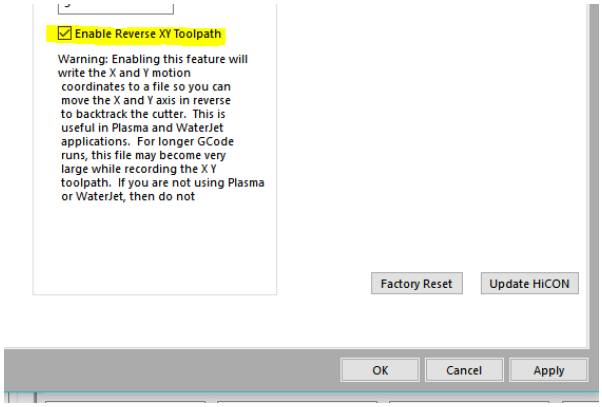
```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
mc.mcRegSendCommand(commandReg, "motion_sync")
```

- **RELOAD_PROFILE_SETTINGS** – Reloads the plugin configuration from the last saved profile.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
mc.mcRegSendCommand(commandReg, "reload_profile_settings")
```

- **REVERSE_TRAJECTORY_START** – allows the user to move along the reversed path or trajectory of a recent move. Follow the instructions below to use this function.

  Enable the reverse XY toolpath feature in the VSI plugin configuration window.



Create a button using the Screen Editor and add the following to the Left Down Script:

```
local inst = mc.mcGetInstance()
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. " /Command")
mc.mcRegSendCommand(commandReg, "REVERSE_TRAJECTORY_START")
```

Add the following to the Left Up Script in the same button:

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
mc.mcRegSendCommand(commandReg, "motion_sequence_stop")
```

Run a gcode file that has XY motion. Once the gcode stops or is stopped by the user, press and hold down the button created in the previous steps and the XY motion will move in the reverse trajectory from the end of the last gcode motion. The machine must be enabled when using this button script.

- **MOTION_SEQUENCE_STOP** – performs stop motion

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
mc.mcRegSendCommand(commandReg, "motion_sequence_stop")
```

- **CLEAR_CUSTOM_REGS** – removes all values in the VSI custom register

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
mc.mcRegSendCommand(commandReg, "clear_custom_regs")
```

- **FW_VERSION** – returns the value of the current firmware

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
local fwVersion = mc.mcRegSendCommand(commandReg, "fw_version")
```

- **SETUP_ABSENC_AXIS_PARAM:<value>** - setup an axis that uses an absolute encoder. See example code.

- **EXEC_ABSENC_READ_CMD:<value>** - start reading the absolute encoder. See example code.

- **SET_ABS_ENC_HOMEPOS** – set the current home position of the absolute encoder. See example code.

- **CLEAR_ABS_ENC_HOMEPOS** – clear the home position of the absolute encoder. See example code.

- **CREATE_REG:<value>** - Creates a custom register. The custom register prefix will be "VSIRegisters" and the name of the register will be the value passed in the argument. The following example creates a custom VSIRegister named customReg and assigns a value of 77.7 to it.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
mc.mcRegSendCommand(commandReg, "create_reg:customReg")
mc.mcRegSetValue(commandReg, 77.7)
```

- **RECOVER_PART:MILL – Currently Under Development.** This command is intended to be used on a 3-axis mill where the Z axis is the router. On machine disarm, the last positions of the XYZ axes are recorded. Sending this command register will move XYZ axes to the last known position before the machine was disarmed. The spindle command 'M3' will turn on after XY motion and before Z motion.

HiCON_Config or DSPMC_Config

Opens or closes the plugin configuration window. Open = 1, close = 0.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/" .. motionDevice .. "_Config")
mc.mcRegSetValue(commandReg, 1)
```

SyncCount

Displays the number of MotionSync calls made by the plugin. Read only

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local syncCountHandle = mc.mcRegGetHandle(inst, motionDevice .. "/SyncCount")
local syncCounts = mc.mcRegGetValueLong(syncCountHandle)
mc.mcCntlSetLastError(inst, tostring(syncCounts))
```

AXIS_DISABLE_BITS

Disable one or more axes. Value sent is integer converted from hex value: 0x17500 + axis id. The axis id starts at 1. The following example shows how to disable Y axis if it is mapped as motor 2 in Mach4. The machine must be enabled first before disabling the axis. Once the machine is disabled and re-enabled again, the disabled axis will become enabled again. Sending a value of 0 to this command register clears the disabled axis and re-enables it if it is currently disabled.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local axisDisableHandle = mc.mcRegGetHandle(inst, motionDevice .. "/AXIS_DISABLE_BITS")
local axis_2 = tonumber(string.format("%d", 0x17500)) + 2
mc.mcRegSetValueLong(axisDisableHandle, axis_2)
```

RunDeviceMacro

Runs the macro program downloaded on the VSI Motion Controller. Any value other than 0 will initiate the process. *Requires Macro Programming API Feature Activation.*

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local runMacroHandle = mc.mcRegGetHandle(inst, motionDevice .. "/RunDeviceMacro")
mc.mcRegSetValueLong(runMacroHandle, 1)
```

ConfigLoad

Increments every time the plugin profile configuration is loaded from the Mach4 machine.ini file. Read only.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local configLoadHandle = mc.mcRegGetHandle(inst, motionDevice .. "/ConfigLoad")
local loadCounts = mc.mcRegGetValueLong(configLoadHandle)
```

StatusBits

Debug information for support. Read only.

DigitalOutputs

Debug information for support. Read only.

JogOverrideEnable

Enables an unmapped motor for movement in velocity mode. Value sent is integer converted from hex value: 0x8D63FA || (axis id << 24). The axis id starts at 0. See example code.

Example value (motor 2):

```
local motor_2 = bit.bor(tonumber(string.format("%d", 0x8D63FA)), bit.lshift(2, 24))
```

JogOverrideRate

Defines the velocity of the motor in JogOverrideEnable mode. *Value is converted to a percentage of motor max velocity in the Mach4 configuration.* See example code.

JogOverridePosition

Defines the distance to move motor in JogOverrideEnable mode. A negative or positive value determines the direction. See example code.

LaserMeasurementAxis

Starts the laser measurement process on given axis. Value sent is integer: 10000 + axis id. The axis id starts at 0. Not available for DSPMC motion controllers. See example code.

LaserDistance

Distance found after laser measurement process completes. Read only. Not available for DSPMC motion controllers. See example code.

LaserMeasureSuccess

Notifies Mach4 that the laser measurement process completed successfully. 0 = false, 1 = true. Read only. Not available for DSPMC motion controllers. See example code.

ReservedOperation1

Debug register for support.

PowerMaxCommandReg

Defines the command to send to the PowerMax65/85. Not available for DSPMC motion controllers. See example code.

Possible register values:

- CUT_MODE_FORCE – 8339
- CURRENT_SET_FORCE – 8340
- PRESSURE_SET_FORCE – 8342
- FAULT_CODE – 8344
- CURRENT_SET_MIN – 8345
- CURRENT_SET_MAX – 8346
- PRESSURE_SET_MIN – 8348
- PRESSURE_SET_MAX – 8349
- ARC_TIME_LOW – 8350
- ARC_TIME_HIGH – 8351
- TORCH_INDEX_1 – 2056
- TORCH_INDEX_2 – 2057
- OUTPUT_PRESSURE – 8268

PowerMaxValueReg

Defines the value to send to the PowerMax65/85. Not available for DSPMC motion controllers. See example code.

PowerMaxReadReg

The value read from the PowerMax65/85 after successful command sent. Not available for DSPMC motion controllers. See example code.

Encoder(0-8)

Encoder counts for each motor. The following example shows how to read encoder at index 0 and print the results to the Mach4 history log.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local encoder0Handle = mc.mcRegGetHandle(inst, motionDevice .. "/Encoder0")
local encoder0Value = mc.mcRegGetValueLong(encoder0Handle)
mc.mcCntlSetLastError(inst, tostring(encoder0Value))
```

ZeroAllEncoders

Resets all encoder counts to zero. Any other value than 0 in the register will initiate the process.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local runMacroHandle = mc.mcRegGetHandle(inst, motionDevice .. "/ZeroAllEncoders")
mc.mcRegSetValueLong(runMacroHandle, 1)
```

EncoderVelocity(0-8)

Rate of change of encoder rotation. See example code.

VADC_(0-7)

Analog input voltage value (0-3300mV for HiCON, +-10V for DSPMCv2). Read only. The following example shows how to read VADC_0 and print the results to the Mach4 history log.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local vadc0Handle = mc.mcRegGetHandle(inst, motionDevice .. "/VADC_0")
local vadc0Value = mc.mcRegGetValueLong(vadc0Handle)
mc.mcCntlSetLastError(inst, tostring(encoder0Value))
```

DRO_To_Ctrl_(0-19)

Floating point registers that are written or read directly to/from the motion controller.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local dro3Handle = mc.mcRegGetHandle(inst, motionDevice .. "/DRO_To_Ctrl_3")
mc.mcRegSetValue(dro3Handle, 88.3) -- set the DRO value to 88.3
local dro3Value = mc.mcRegGetValue(dro3Handle) -- read the DRO value
```

DRO_From_Ctrl_(0-19)

Floating point registers that are read directly from the motion controller. Read only.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local dro12Handle = mc.mcRegGetHandle(inst, motionDevice .. "/DRO_From_Ctrl_12")
local dro12Value = mc.mcRegGetValue(dro12Handle) -- read the DRO value
```

LED_To_Ctrl_(0-31)

Binary registers that are written or read directly to/from the motion controller.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local led3Handle = mc.mcRegGetHandle(inst, motionDevice .. "/LED_To_Ctrl_3")
mc.mcRegSetValue(led3Handle, 1) -- set the LED value to 1
local led3Value = mc.mcRegGetValue(led3Handle) -- read the LED value
```

LED_From_Ctrl_(0-31)

Binary registers that are read directly from the motion controller. Read only.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local led12Handle = mc.mcRegGetHandle(inst, motionDevice .. "/LED_From_Ctrl_12")
local led12Value = mc.mcRegGetValue(led12Handle) -- read the LED value
```

RTapDeccelTime

The amount of time (in seconds) that it takes for the Maximum Spindle RPM to decelerate to the “Min Tap RPM”.

RTapUseIndexPulse

If enabled, this setting will allow the Z axis to wait for the index pulse trigger (from the spindle encoder) before initiating the tap cycle. This allows for consistent threading orientation on all tap cycles.

RTapIndexPulseRPM

The RPM of spindle when waiting for the Index Pulse. This has no effect if index pulse is not used for the tapping cycle.

RTapMinTapRPM

The minimum RPM that the spindle will decelerate to. This stabilizes the spindle as it approaches the target depth of the tapping cycle which allows the spindle to come to a complete halt when the target depth is reached.

RTapMinTapTime

The amount of time (in seconds) that the spindle will maintain the “Min Tap RPM” before completely turning off the spindle. Minimizing the RPM by the end of the tap cycle allows the spindle to take less time decelerating, which helps prevent “overshooting” the depth.

RTapDriftRevs

The estimated amount of revolutions that the spindle will make before coming to a complete halt from the “Min Tap RPM”.

RTapRetractDelay

The amount of time (in milliseconds) to delay before retracting the spindle after drilling.

RTapRetractRatio

This value is a multiplier for the RPM of the spindle when retracting. For example, if a spindle RPM of 500 is used for the tapping cycle and a “Retract Ratio” of 1.75 is set, then the spindle will retract at an RPM of 875.

RTapFeedForward

NOT RECOMMENDED TO USE. It is still preferable to perform feed forward tuning on servo drives. Set this to zero to disable it.
This parameter is used to minimize the following error between the spindle and Z motion by applying a feed forward multiplier. It is recommended to use small values (ex. 0.1), then gradually increase the value to the desired performance.

THC_ModeEnabled

Only used in ArcPro versions 1.xx. If enabled, puts the Z-axis into torch height control mode when running M3 script. To remember the last reference position after a probe and only run probe sequence once during a series of cuts, set the value of this register to 3. Otherwise, set the value to 1 to enable or 0 to disable. Refer to the ArcPro version 1.xx manual.

THC_ExtMode

Only used in ArcPro versions 2.xx. Secondary THC mode. Using this mode allows the user to bypass or program their own probe and ignition phase in the M3 script. This register should be set to 1 after receiving the ARCOK signal from the torch height device. The reference position is captured from analog input after the THC delay expires. Enabled = 1, disabled = 0. Refer to the ArcPro version 2.xx manual.

THC_MaxHeightCorrection

The maximum correction distance ***above*** the pierce height reference position. Refer to the ArcPro version 1.xx or 2.xx manual.

THC_MinHeightCorrection

The maximum correction distance ***below*** the pierce height reference position. Refer to the ArcPro version 1.xx or 2.xx manual.

THC_SpeedPercent

Modifies the velocity of the torch height controlled axis during a cut. Refer to the ArcPro version 1.xx or 2.xx manual.

THC_AccelPercent

Only used in ArcPro versions 2.xx. Modifies the acceleration of the torch height controlled axis during a cut. Refer to the ArcPro version 2.xx manual.

THC_FeedrateAntiDive

Feedrate Anti-Dive prevents the THC from dropping the torch into a cut hole, diving into corners, or diving at the end of a cut. When the XY cutting speed slows down, the plasma tip voltage increases, and as a result, the response from the THC is to lower the torch. When the actual cutting feedrate drops below the ***specified percentage*** of the commanded feedrate, Anti-Dive is engaged and the Z-Axis motion is disabled and stays locked in positon. Refer to the ArcPro version 1.xx or 2.xx manual.

THC_MaxTipVoltsRateOfChange

Only used in ArcPro versions 2.xx. Defines the maximum amount of tip volts change, from the reference voltage, within 50ms time period. The controller is constantly checking every 50ms during torch height control to verify that the tip voltage change has not exceeded this range. If the tip volts changes drastically and falls outside of this range, torch height motion is disabled until the tip volts comes back near the reference voltage. Refer to the ArcPro version 2.xx manual.

THC_PierceHeight

Only used in ArcPro versions 1.xx. The *Pierce Height* defines, in inches or millimeters, the height above the material that the cutter head will sit while conducting a pierce action. A good value for

pierce height can be found by looking in the consumables chart for the system's plasma cutter. ***This value is relative to the detected material surface height.*** Refer to the ArcPro version 1.xx manual.

THC_IgnitionHeight

ArcPro versions 1.xx

The Ignition Height is used on thick material where the pierce height is higher than the plasma head can easily establish an arc. On thinner material the Ignition Height can be set to zero and the system will skip the feature. ***This value is relative to the detected material surface height.*** Refer to the ArcPro version 1.xx manual.

ArcPro versions 2.xx

When the value of this register is set to 999999, the tip voltage reference point will automatically be found after THC_Delay expires. When the value is set to zero the tip voltage reference point is the value that is put into the DRO_To_Ctrl_18 register. Refer to the ArcPro version 2.xx manual.

THC_FloatingHeadOffset

Only used in ArcPro versions 1.xx. The value of this register should be set to zero when using ArcPro versions 2.xx. The value of this register is added to the THC_PierceHeight and THC_IgnitionHeight registers during THC setup. The value of this register is the offset from the torch height controlled axis floating head mount and the torch tip. Refer to the ArcPro version 1.xx manual.

THC_PierceDelay

Only used in ArcPro versions 1.xx. The time, in milliseconds, that the cutter head will pause while piercing. This gives time for the hole to go all the way through the material. Refer to the ArcPro version 1.xx manual.

THC_ControlDelay

Milliseconds to wait, after probing or THC setup sequence, before giving control of the Z axis to the torch height controller. Refer to the ArcPro version 1.xx or 2.xx manual.

THC_ProbeSpeed

Only used in ArcPro versions 1.xx. This value indicates at what percent of the max velocity the will Z move down to locate the material surface height during the THC probing sequence. Refer to the ArcPro version 1.xx manual.

THC_ReferenceFound

Only used in ArcPro versions 1.xx. Indicates whether or not the Z reference height has been found in a previous THC sequence. Enabled = 1, disabled = false. Read only. Refer to the ArcPro version 1.xx manual.

VSI Register Examples

PowerMax 65/85

Example send CUT_MODE_FORCE command to PowerMax65/85:

```
local inst = mc.mcGetInstance()
local powerMaxCmdReg = mc.mcRegGetHandle(inst, "HiCON/PowerMaxCommandReg")
local powerMaxCmdVal = mc.mcRegGetHandle(inst, "HiCON/PowerMaxValueReg")
local cut_mode_force = tonumber(string.format("%d", 0x2093))

mc.mcRegSetValueLong(powerMaxCmdReg, cut_mode_force)
mc.mcRegSetValueLong(powerMaxCmdVal, 1)

mc.mcRegSetValueLong(powerMaxCmdReg, 0)
mc.mcRegSetValueLong(powerMaxCmdVal, 0)
```

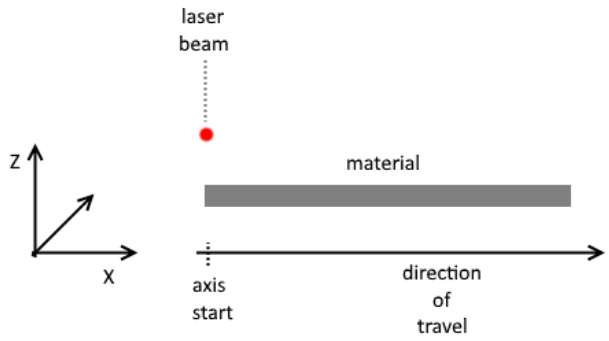
Example read result from PowerMax65/85:

```
local inst = mc.mcGetInstance()
local powerMaxReadReg = mc.mcRegGetHandle(inst, "HiCON/PowerMaxReadReg")
local powerMaxReadValue = mc.mcRegGetValue(powerMaxReadReg)
```

Measure a Part using a Laser Input Sensor

Overview

This example assumes the machine is equipped with a laser input sensor installed on an axis and is activated and deactivated as the axis moves along the side of the material to be measured.



The axis is set into position at the start of material before the move. At the start of measurement, the axis will move two units towards the end of material before starting to listening for the laser to deactivate. This ensures that any false material at beginning of the part is ignored. The axis continues to travel in the same direction until the laser is deactivated again (end of the material). This example requires the use of the VSI Command and Jog Override registers to control the axis. The input for the laser must be mapped to Input #63 in the Mach4 configuration. The scripting is done in three areas in Mach4: PLC Script, Screen Load Script, Screen Button Script.

Screen Load Script

These are the global variables and function that will be used in the other two scripts.

```
inst = mc.mcGetInstance()    -- may already be defined in the Screen Load script

laserMeasurementReg = mc.mcRegGetHandle(inst, "HiCON/LaserMeasurementAxis")
laserMeasureSuccessReg = mc.mcRegGetHandle(inst, "HiCON/LaserMeasureSuccess")
laserDistanceReg = mc.mcRegGetHandle(inst, "HiCON/LaserDistance")

jogOverrideEnableReg = mc.mcRegGetHandle(inst, "HiCON/JogOverrideEnable")
jogOverrideRateReg = mc.mcRegGetHandle(inst, "HiCON/JogOverrideRate")
jogOverridePositionReg = mc.mcRegGetHandle(inst, "HiCON/JogOverridePosition")

hiconCommandReg = mc.mcRegGetHandle(inst, "HiCON/Command")

motor_0 = bit.bor(tonumber(string.format("%d", 0x8D63FA)), bit.lshift(0, 24))

maxAxisMove = 48      -- max amount of units that the axis will move during measure
axisVelocity = 20         -- percentage of max velocity that the axis will move during measure
motionTimeout = 0
motionStopped, measureStarted, measureEnded, processComplete = false, false, false, false

function StartLaserMeasure(moveDirection)
   mc.mcCntlSetLastError(inst, "Measuring Part")
   local laserMeasureEnable = mc.mcRegGetValue(laserMeasurementReg)
   laserMeasureEnable = 10000
   mc.mcRegSetValueLong(laserMeasurementReg, laserMeasureEnable)
   mc.mcRegSetValue(jogOverrideEnableReg, motor_0)

   mc.mcRegSetValue(jogOverridePositionReg, maxAxisMove * moveDirection)
   mc.mcRegSetValueLong(jogOverrideRateReg, axisVelocity)
   motionTimeout = os.clock() + 5    -- timeout delay 5 seconds
   measureStarted = true
end
```

Button Clicked Script

Before pressing the button, the axis should be in position where the laser is not activated and the material to be measured is in the direction of travel.

```
local direction = 1  -- should be 1 or -1 depending on direction to travel
StartLaserMeasure(direction)
```

PLC Script

The PLC is constantly listening for when the controller reports to the plugin that the laser measurement process is complete. The process will timeout if motionTimeout expires before the controller reports that the process is complete.

```
ReadLaserDistance()
  mc.mcCntlSetLastError(inst, string.format("%.4f", mc.mcRegGetValue(laserDistanceReg)))
end

function LaserMeasureSuccess()
  return mc.mcRegGetValue(laserMeasureSuccessReg) ~= 0
end

function SyncMotion()
  mc.mcRegSendCommand(hiconCommandReg, "")
  mc.mcRegSendCommand(hiconCommandReg, "motion_sync")
  mc.mcRegSendCommand(hiconCommandReg, "")
end

function ClearLaserMeasureParameters()
  motionTimeout = 0
  measureStarted = false
  mc.mcRegSetValueLong(laserMeasurementReg, 0)
  mc.mcRegSetValueLong(laserMeasureSuccessReg, 0)
  mc.mcRegSetValue(jogOverrideEnableReg, 0)
  mc.mcRegSetValue(jogOverridePositionReg, 0)
  mc.mcRegSetValueLong(jogOverrideRateReg, 0)
end

function StopMoveAfterMeasure()
  ClearLaserMeasureParameters()
  SyncMotion()
  motionStopped = true
end

function RunProcess()
  if measureStarted then
    if os.clock() >  motionTimeout and motionTimeout ~= 0 then
      mc.mcCntlSetLastError(inst, "Laser did not detect part")
       measureEnded = false
       StopMoveAfterMeasure()
    end
    if LaserMeasureSuccess() then
      mc.mcCntlSetLastError(inst, "Laser Measure Read Success")
      measureEnded = true
      StopMoveAfterMeasure()
    end
  end

  if measureEnded then
    measureEnded = false
    StopMoveAfterMeasure()
  end

  if motionStopped then
    motionStopped  = false
    if measureEnded then
      measureEnded = false
      ReadLaserDistance()
    end
  end
end

RunProcess()
```

Control FeedRate/RapidRate Override Using Encoder Velocity

Overview

The velocity of each encoder is stored in VSI registers and can be used to override the feedrate while the machine is in motion. It's recommended to use an encoder that is NOT connected to a motor (ie, handwheel) for this mode. The following example will use an MPG handwheel. The scripting is done in three areas in Mach4: PLC Script, Screen Load Script, Screen Button Script.

Screen Load Script

These are the global variables and functions that will be used in the other two scripts.

```
inst = mc.mcGetInstance()   -- may already be defined in the Screen Load script

lastFeedRate = 0
lastRapidRate = 0
velOverrideMode = 0

-- returns the encoder index for the MPG at index 0
function GetMPGAxis()
   local hReg = mc.mcMpgGetEncoderReg(inst, 0)     -- value to use as 2nd parameter is the MPG id in Mach4 MPGs configuration, starts at zero
   local regName = mc.mcRegGetInfo(hReg)
   return tonumber(string.match(regName, "%d+"))   -- returns encoder index mapped in Mach4 MPGs configuration
end

mpgAxis = GetMPGAxis()

-- reset the feedrate overrides back to normal values on screen load
function ResetVelocityOverride()
   if lastFeedRate == 0 or lastRapidRate == 0 then
      lastFeedRate = 100
      lastRapidRate = 100
   end

   mc.mcCntlSetFRO(inst, lastFeedRate)
   mc.mcCntlSetRRO(inst, lastRapidRate)
end

ResetVelocityOverride()

function ToggleVelocityOverride()
   if velOverrideMode == 1 then
      if lastFeedRate == 0 or lastRapidRate == 0 then
         lastFeedRate = 100
         lastRapidRate = 100
      end
      mc.mcCntlSetFRO(inst, lastFeedRate)
      mc.mcCntlSetRRO(inst, lastRapidRate)
      velOverrideMode = 0
   else
      lastFeedRate = mc.mcCntlGetFRO(inst)
      lastRapidRate = mc.mcCntlGetRRO(inst)
      mc.mcCntlSetFRO(inst, 0)
      mc.mcCntlSetRRO(inst, 0)
      velOverrideMode = 1
   end
end
```

PLC Script

The PLC will listen for when the state of velOverrideMode is changed. If the velOverrideMode is enabled, the encoder velocity will override the current feedrate.

```
if velOverrideMode == 1 then
   local hReg = mc.mcRegGetHandle(inst, "HiCON/EncoderVelocity" .. tostring(mpgAxis))
   local value = mc.mcRegGetValue(hReg)
   mc.mcCntlSetFRO(inst, value/10)
   mc.mcCntlSetRRO(inst, value/10)
end
```

Button Clicked Script

```
ToggleVelocityOverride()
```

After successful compile of the above scripts, run motion in the MDI or with a gcode file and press the button to toggle the encoder velocity override mode. If the encoder is not moving, the feedrate should drop to zero. As you turn the encoder, the velocity will increase and the feedrate will change.

Axis Movement Override

Command a motor to move outside of Mach4 control. In the following example motor 1 is commanded to move 48 units in the positive direction at a rate of 20% of the maximum velocity for the motor. This function sends movement to the VSI motion controller directly. Moving the axis outside of Mach4 control requires a motion sync call afterwards to prevent a following error.

```
jogOverrideEnableReg = mc.mcRegGetHandle(inst, "HiCON/JogOverrideEnable")
jogOverrideRateReg = mc.mcRegGetHandle(inst, "HiCON/JogOverrideRate")
jogOverridePositionReg = mc.mcRegGetHandle(inst, "HiCON/JogOverridePosition")

motor_1 = bit.bor(tonumber(string.format("%d", 0x8D63FA)), bit.lshift(1, 24))
maxAxisMove = 48          -- max amount of units that the axis will move during measure
axisVelocity = 20         -- percentage of max velocity that the axis will move during measure

mc.mcRegSetValue(jogOverrideEnableReg, motor_1)
mc.mcRegSetValue(jogOverridePositionReg, maxAxisMove)   -- direction determined by the sign of maxAxisMove
mc.mcRegSetValueLong(jogOverrideRateReg, axisVelocity)
```

Absolute Encoder Feedback

Initialize Encoder Axis with Counts Per Rev (Mach4 Enable Button Down Script)

This command will set up the encoder at the axis id to be recognized as an absolute encoder in the motion controller's firmware. *The spaces and the 'CPR' text in the format of the command's value are required for the command to work.* The down action of the Mach4 enable button should be turned off and replaced by the following logic in the button down script function.

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
local fwVersion = mc.mcRegSendCommand(commandReg, "SETUP_ABSENC_AXIS_PARAM:0 CPR 32786")
wx.wxMilliSleep(1500) -- wait 1.5 seconds until sending command again
```

Read Absolute Encoder Position (Mach4 Enable Button Down Script)

In the following example, any axis initialized to record absolute encoder feedback is read and the motor position is synced to the actual feedback position. In the example, output 2 is used as the SEN signal for all motor drives. *This code will follow the above code.*

```
local val = mc.mcRegSendCommand(reg, "EXEC_ABSENC_READ_CMD:SEN " .. tostring(mc.OSIG_OUTPUT2));

if (val ~= "0") then
   wx.wxMessageBox("ABS Encoder Error Detected: " .. tostring(val));
   return
end

mc.mcCntlEnable(inst, 1);
scr.SetProperty('tbutton2', 'Button State', 'Down');
```

Clear Absolute Encoder Position (Mach4 Enable Button Down Script)

```
local inst = mc.mcGetInstance()
local motionDevice = mc.mcProfileGetString(inst, "Preferences", "MotionDevice", 'NO MOTION')
local commandReg = mc.mcRegGetHandle(inst, motionDevice .. "/Command")
local fwVersion = mc.mcRegSendCommand(commandReg, "CLEAR_ABS_ENC_HOMEPOS")
wx.wxMilliSleep(1500) -- wait 1.5 seconds until sending command again
```