Power PMAC Additional Advanced Training Material

Stepper Motor Setup

Types of Stepper Control

- **Fully Closed Loop**
  - Uses an encoder for position feedback and commutation
  - Acts virtually just like a 50 pole pair brushless DC motor (for 200 step stepper motors) or 100 pole pairs (for 400 step stepper motors)
  - Setup is identical to brushless servomotor

- **Direct Microstepping**
  - Does not involve an encoder
  - Integrate quadrature current command to use for simulated position feedback and commutation position by means of a special (**EncTable[n].Type = 11**) Encoder Conversion Table entry which converts floating point phase position into integer
  - Uses predetermined servo loop gains
  - Yields 102,400 motor counts per revolution for 200 step steppers

- **Hybrid**
  - Uses an encoder for position feedback, but Direct Microstepping for commutation
  - Fundamentally, the only difference between Hybrid and Direct Microstepping is that the Encoder Conversion Table entry is **Type = 1** and points to the encoder register, instead of integrating **IqCmd** for position feedback
  - Delta Tau does not recommend this method because there is almost always a mismatch between encoder resolution used for position feedback and microstepping resolution used for commutation, making tuning the velocity loop very difficult
  - Requires additional servo loop tuning (treat as velocity mode amplifier)

Typical Closed Loop Stepper Setup

```
// Motor Active
Motor[2].ServoCtrl = 1;

// Encoder Conversion Table
// This is default for quadrature encoders

// Use Two Phase Mode for all steppers
BrickLv.Chan[1].TwoPhaseMode = 1;

// Overtravel Limits, set = 0 to disable
Motor[2].pLimits = PowerBrick[0].Chan[1].Status.a;

// ADC Mask
Motor[2].AdcMask = $FFFC0000;

// Amplifier Fault Level
Motor[2].AmpFaultLevel = 1;

// Phase Offset
Motor[2].PhaseOffset = 512; // for 2-phase motors

// Phase Control
Motor[2].PhaseCtrl = 4;

// GLOBAL DcBusInput = 48; // DC Bus input voltage [VDC] –User Input, but defined earlier in our example setup
#define Mtr2DCVoltage 24 // Motor #2 DC rated voltage [VDC] –User Input
Motor[2].PwmSf = 0.95 * 16384 * Mtr2DCVoltage / DcBusInput;

// For Quadrature Encoder:
#define Mtr2StepAngle         1.8 // degrees per step, almost always the same for all steppers
#define Mtr2NumPolePairs      (360.0/(Mtr2StepAngle*4.0))
#define Mtr2CountsPerRev      10000.0        // -User Input
Motor[2].PhasePosSf = 2048.0 * Mtr2NumPolePairs / (256.0 * Mtr2CountsPerRev)
```

Power PMAC Script

Typical Closed Loop Stepper Setup

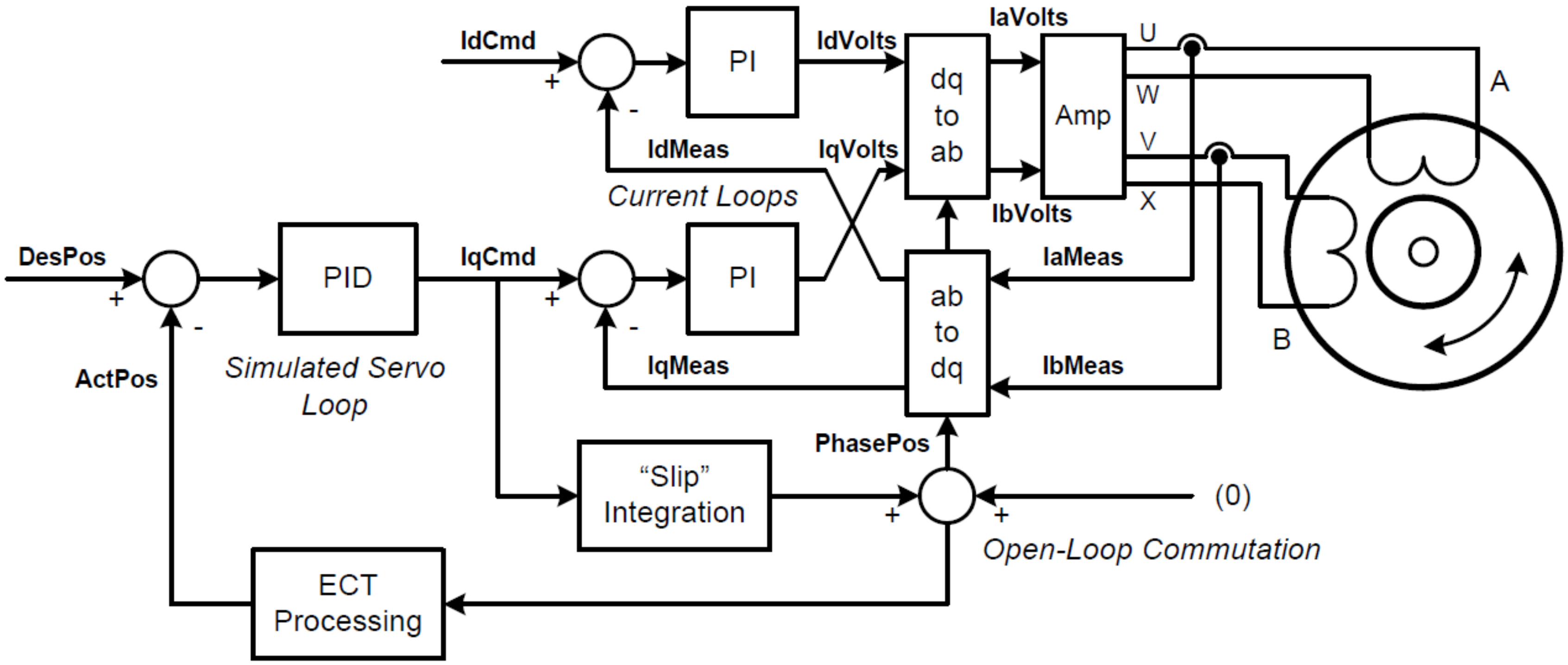
```
// Motor #2 I2T Settings Example
#define Ch2MaxAdc 33.85 // Max ADC reading [A rms] -–User Input
#define Ch2RmsPeakCur 4 // RMS Peak Current [A rms] -–User Input
#define Ch2RmsContCur 2 // RMS Continuous Current [A rms] --User Input
#define Ch2TimeAtPeak 1 // Time Allowed at peak [sec] --User Input
Motor[2].MaxDac = Ch2RmsPeakCur / Ch2MaxAdc * 32767 * 0.866;
Motor[2].I2TSet = Ch2RmsContCur / Ch2MaxAdc * 32767 * 0.866;
Motor[2].I2tTrip = (POW(Motor[2].MaxDac,2) - POW(Motor[2].I2TSet,2)) * Ch2TimeAtPeak;

// Current Loop Tuning
Motor[2].IiGain =  2;
Motor[2].IpbGain = 16;
Motor[2].IpfGain = 0;

// Phasing
#define Mtr2PhasingTime 1000 // Total phasing time [msec] --User Input
Motor[2].PhaseFindingTime = Mtr2PhasingTime * 0.5 / (Sys.ServoPeriod * (Sys.RtIntPeriod + 1))
Motor[2].PhaseFindingDac = Motor[2].I2TSet / 3 // Phasing search magnitude --User Input
Motor[2].AbsPhasePosOffset = 2048 / 6 // Qualifying motor movement
Motor[2].PowerOnMode = Motor[2].PowerOnMode & $5 // No power-on phasing
Gate3[0].Chan[1].EncCtrl = 7;

// Servo Loop Tuning
Motor[2].Servo.Kp=5.3379192
Motor[2].Servo.Kvfb=200
Motor[2].Servo.Kvff=200
Motor[2].Servo.Kaff=3500
Motor[2].Servo.Kfff=100
Motor[2].Servo.Ki=0
Motor[2].FatalFeLimit = 10000
```

Power PMAC Script

Notes on Closed Loop Stepper (FCLS)

- **FCLS Notes**
  - Need to tune current loop and position loop gains just like a Brushless DC Motor, recommend doing it interactively
  - Provides the most reliable control method for stepper motors because the position can always be verified (i.e. if the encoder is functional) and matches the commutation resolution

Direct Microstepping Operating Principle



Typical Direct Microstepping Setup

```
// Motor Activate
Motor[4].ServoCtrl = 1;

// Use Two Phase Mode for all steppers
BrickLv.Chan[3].TwoPhaseMode = 1;

// Encoder Conversion Table
// This is a special entry that integrates the IqCmd for feedback
// This entry reads a 64-bit floating point number and turns it into an integer for feedback
EncTable[4].type = 11
EncTable[4].pEnc = Motor[4].PhasePos.a
EncTable[4].index1 = 5 // Shift data up to the MSB of the word (32 bits - (11 + 8 + 8)),
// 11 bits from 2048 phase positions, first 8 bits comes from entry type 11
//which scales up by 256 by default, last 8 comes from index 5
// This generates more resolution from the data

EncTable[4].index2 = 0
EncTable[4].index3 = 0
EncTable[4].index4 = 0
EncTable[4].index5 = 255
EncTable[4].index6 = 1          //              Selects 64-bit double for reading
EncTable[4].ScaleFactor = 1 / (256 * (EncTable[4].index5 + 1) * EXP2(EncTable[4].index1)) // Scales double to int
Motor[4].pEnc = EncTable[4].a;
Motor[4].pEnc2 = EncTable[4].a;
// End result is in same units as commutation counts and we have 50*2048/4 = 102,400 cts per rev
```

Power PMAC Script

Typical Direct Microstepping Setup

```
// ADC Mask
Motor[4].AdcMask = $FFFC0000;

// Overtravel Limits, set = 0 to disable
Motor[4].pLimits = PowerBrick[0].Chan[3].Status.a;

// Amplifier Fault Level, Low True
Motor[4].AmpFaultLevel = 1;

// Phase Offset, For 2-phase Motor (Stepper)
Motor[4].PhaseOffset = 512;

// Phase Control
Motor[4].PhaseCtrl = 6; // Unpacked, commutation enabled, PMAC needs to do slip calculations

// Third Harmonic, disable
Motor[4].PhaseMode = 1;

// Commutation Cycle Size
Motor[4].PhasePosSf = 0; // Force this to 0 so the contents of pPhaseEnc's register has no effect
// Then firmware ignores pPhaseEnc and gets phase position directly from integrating IqCmd * SlipGain

// Absolute Power-on Phase Reference address location
Motor[4].pAbsPhasePos=PowerBrick[0].Chan[3].PhaseCapt.a;

// Power-On mode, Establish Phase Reference Automatically on Power-up
Motor[4].PowerOnMode = 2;

// Servo Error Input Magnitude Limit, Increase Limit
Motor[4].Servo.MaxPosErr = 100000;

// Slip Gain, Constant
Motor[4].DtOverRotorTc = 0; // PMAC does not compute SlipGain; we must dictate it
Motor[4].SlipGain = Sys.PhaseOverServoPeriod;
```

Power PMAC Script

Typical Direct Microstepping Setup

```
// Commutation Phase Advance Gain
// This advances your phase position internally proportional to speed for purposes of calculating the
// voltage command from the current loop
// This is calculated based on "filtered velocity" which is a moving sum of the last 16 velocities sampled
// Hence, need to divide by 16.
// Advance Gain is used per phase cycle, but filtered velocity is in units of servo cycles,
// so we convert to phase by multiplying by sys.phaseoverservoperiod
// (0.25/Sys.ServoPeriod/Sys.PhaseOverServoPeriod) is the time delay between reading ADCs and actually using them,
// which is 4 phase cycles as a result of the sigma delta method (successive approximation)
Motor[4].AdvGain = 1/16*Sys.PhaseOverServoPeriod*(0.25/Sys.ServoPeriod/Sys.PhaseOverServoPeriod)

// Position Loop Gains
Motor[4].Servo.Kp = 1
Motor[4].Servo.Kvff = 1
Motor[4].Servo.Kaff = 1
Motor[4].Servo.Kvfb = 0
Motor[4].Servo.Ki = 0
Motor[4].Servo.Kvifb = 0
Motor[4].Servo.Kviff = 0

// PWM Scale Factor
#define Mtr4DCVoltage 24 // Motor #4 DC rated voltage [VDC] –User Input
Motor[4].PwmSf = 0.95 * 16384 * Mtr4DCVoltage / DcBusInput; // DcBusInput defined earlier in presentation
```

Power PMAC Script

Typical Direct Microstepping Setup

```
// Motor #4 I2T Calculation
#define Ch4MaxAdc 33.85 // Max ADC reading [A] --User Input
#define Ch4RmsPeakCur 4 // RMS Peak Current [A] --User Input
#define Ch4RmsContCur 2 // RMS Continuous Current [A] --User Input
#define Ch4TimeAtPeak 1 // Time Allowed at peak [sec] --User Input
GLOBAL Ch4MaxOutput = 0; // Calculation Holding Register
Ch4MaxOutput = Ch4RmsPeakCur / Ch4MaxAdc * 32767 * 0.866;
Motor[4].I2TSet = Ch4RmsContCur / Ch4MaxAdc * 32767 * 0.866;
Motor[4].I2tTrip = (POW(Ch4MaxOutput,2) - POW(Motor[4].I2TSet,2)) * Ch4TimeAtPeak;

// Magnetization Current
// Increase for stiffer holding current, decrease for higher top speeds
// PMAC uses IdCmd to move the motor, basically just locking the rotor into each phase of the motor
// successively throughout its revolution
Motor[4].IdCmd = Motor[4].I2TSet / 2;

// Maximum Output Value
#define Mtr4MaxRpm 1500 // Motor Maximum Speed [RPM] --User Input
#define Mtr4StepAngle 1.8 // Motor Step Angle [degrees] --User Input
Motor[4].MaxDac = Mtr4MaxRpm / 60000 * (360 / (4 * Mtr4StepAngle)) * 2048 * Sys.ServoPeriod

// Current Loop Tuning
Motor[4].IiGain = 1;
Motor[4].IpfGain = 0;
Motor[4].IpbGain = 7;
```

Power PMAC Script

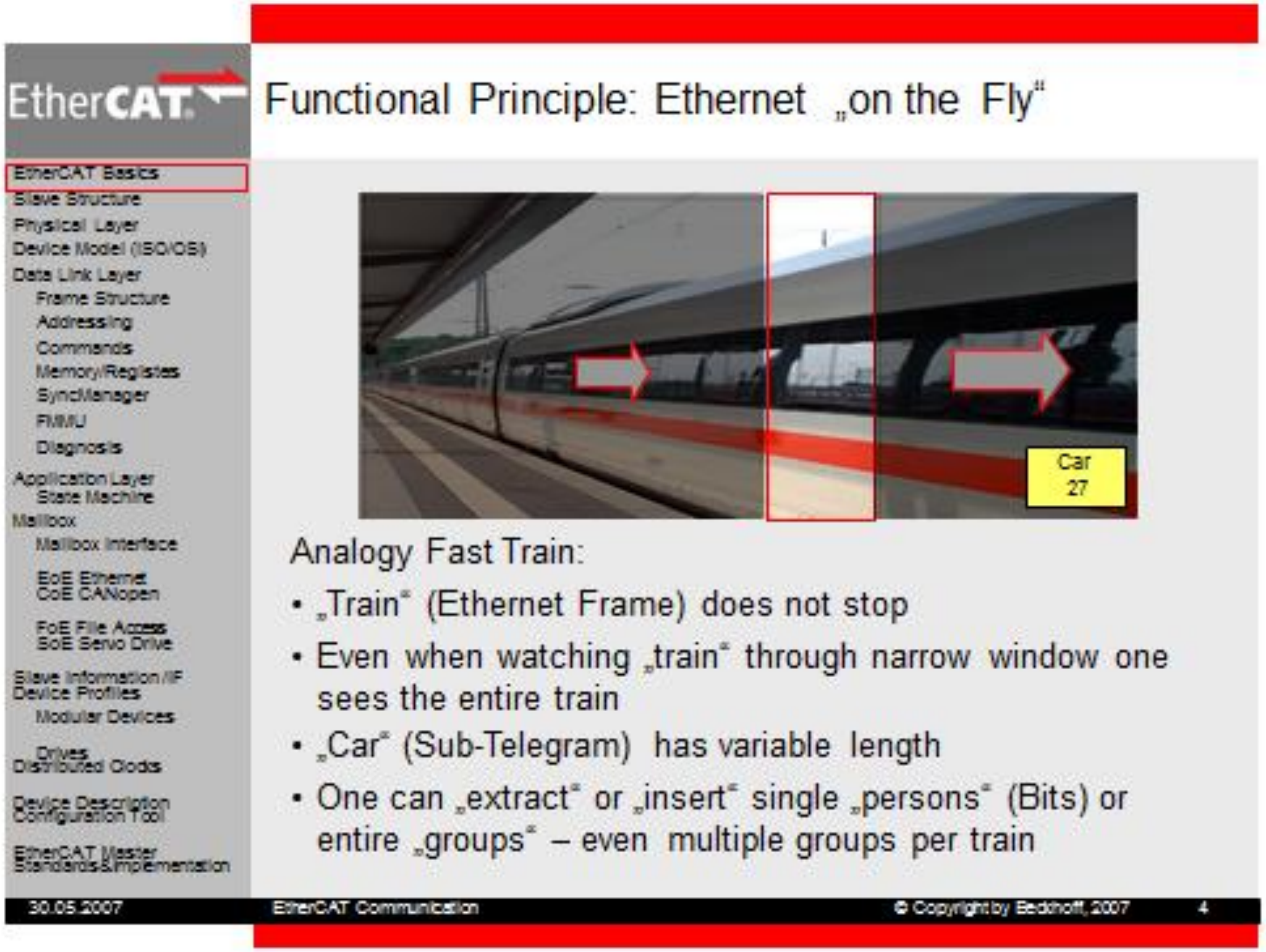
Direct Microstepping Notes

- Once **Motor[*x*].PhaseCtrl** bit 1 is set to 1, the firmware starts integrating **Motor[*x*].PhasePos += Motor[*x*].IqCmd*Motor[*x*].SlipGain** every phase cycle
- **Motor[*x*].SlipGain = Sys.PhaseOverServoPeriod** because multiple phase cycles can happen between servo cycles, so multiplying by **Sys.PhaseOverServoPeriod** will scale the sum to the correct value (e.g. if phase is 10 kHz and servo is 2 kHz, the firmware would add the same **Motor[*x*].IqCmd** five times and the integrated **IqCmd** would be 5 times too large, but multiplying by **Sys.PhaseOverServoPeriod** is the same as dividing this by 5, bringing the sum to the correct **IqCmd** value)
- Stepper motors require four current fields 90 degrees apart throughout each commutation cycle in order to commutate one electrical cycle
- Since there are 2048 entries in PPMAC's commutation sine table, the maximum number of entries the motor can traverse per phase cycle is 512 (90 degrees of electrical cycle) or 1024 per servo cycle (180 degrees) before PPMAC loses track of the motor's rotation direction
- Thus, **Motor[*x*].MaxDac** is limited according to the servo period such that the stepper motor can still achieve its nominal speed
- **Motor[*x*].AdvGain** is the "Phase Advance Gain" and is used to internally rotate the rotor's phase position a certain amount "ahead" of where it actually is proportional to the rotor speed for the purpose of properly computing torque commands to the motor; without this, there is a torque loss proportional to speed
- Phasing is "automatic" because the rotor is only ever at maximum 0.9 degrees out of phase with one of its coils as a result of the "steps" being 1.8 degrees apart (for typical stepper motors)
- **BrickLV.Chan[*j*].TwoPhaseMode** (if using Power Brick LV) is set to a value of 1, so the amplifier channel is placed in 2-phase operational mode, using the U and W output lines to drive the first phase, and the V and X output lines to drive the second phase.

Introduction to EtherCAT

EtherCAT

- **Ethernet for Control Automation Technology (EtherCAT)**



EtherCAT

Functional Principle: Ethernet „on the Fly“

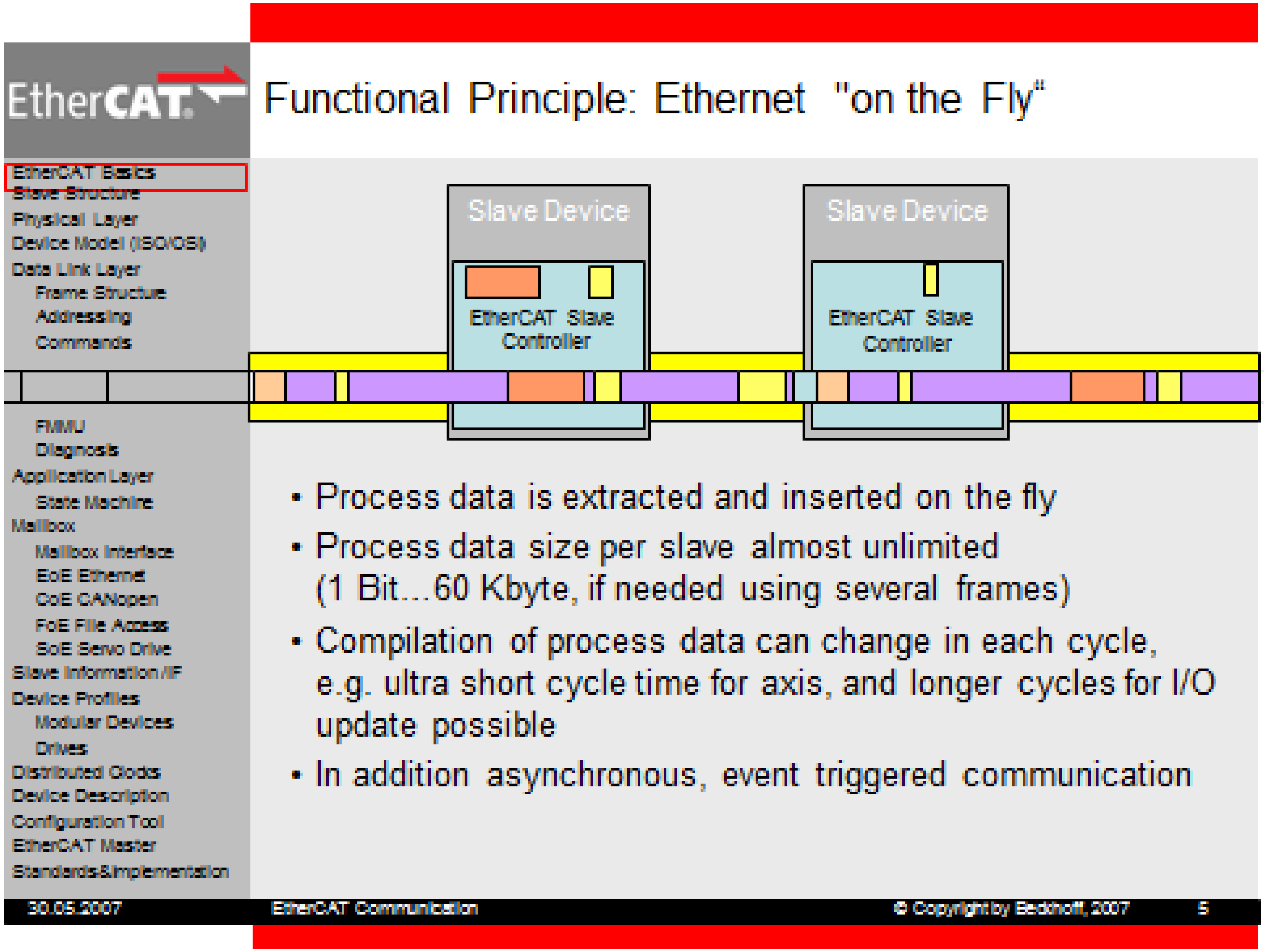
EtherCAT Basics
Slave Structure
Physical Layer
Device Model (ISO/OSI)
Data Link Layer
  Frame Structure
  Addressing
  Commands
  Memory/Registes
  SyncManager
  FMMU
  Diagnosis
Application Layer
  State Machine
Mailbox
  Mailbox Interface
  EoE Ethernet
  CoE CANopen
  FoE File Access
  SoE Servo Drive
Slave Information/IF
Device Profiles
  Modular Devices
  Drives
Distributed Clocks
Device Description
Configuration Tool
EtherCAT Master
Standards&Implementation

Analogy Fast Train:

- „Train“ (Ethernet Frame) does not stop
- Even when watching „train“ through narrow window one sees the entire train
- „Car“ (Sub-Telegram) has variable length
- One can „extract“ or „insert“ single „persons“ (Bits) or entire „groups“ – even multiple groups per train

30.05.2007 EtherCAT Communication © Copyright by Beckhoff, 2007 4

EtherCAT

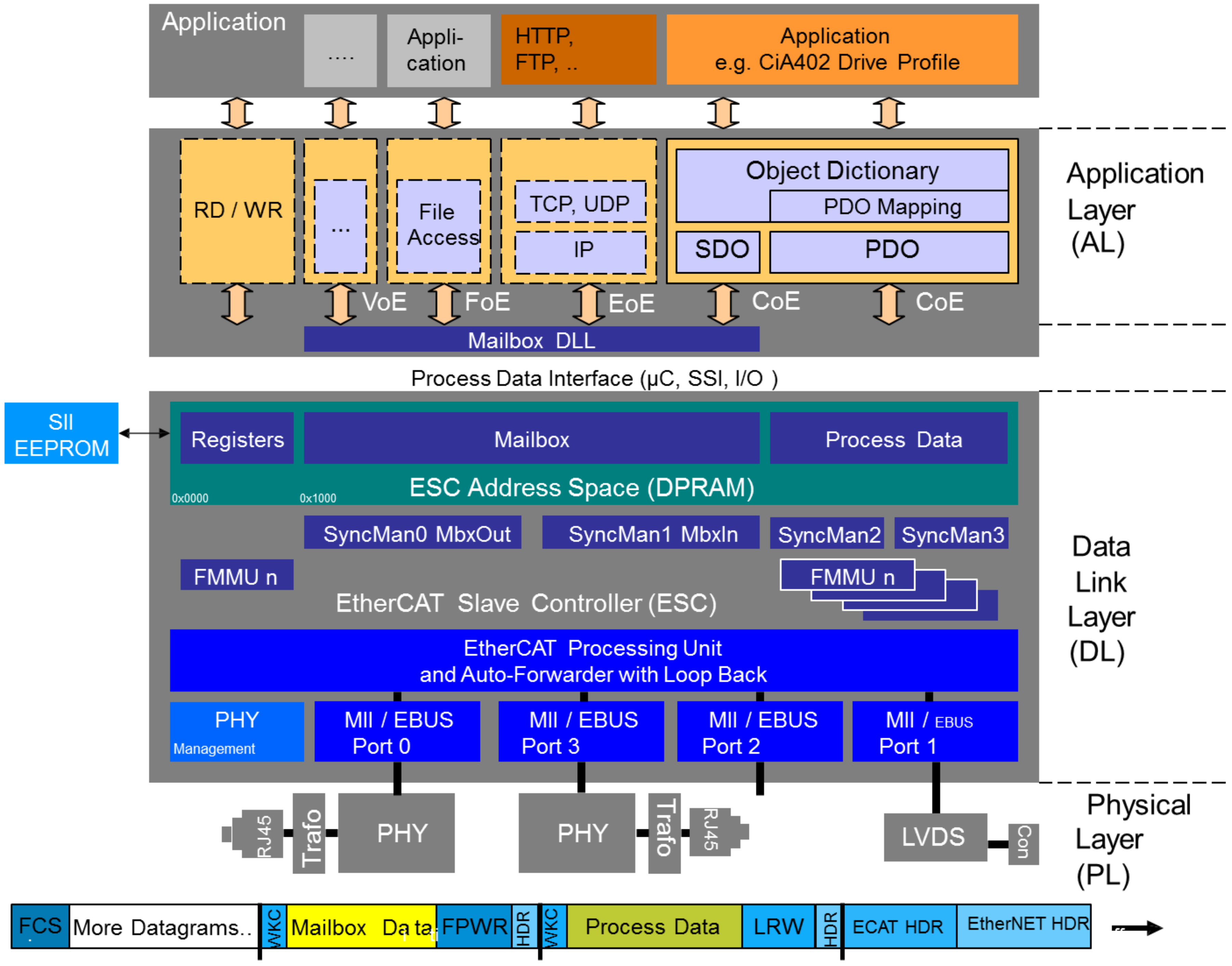


EtherCAT

Functional Principle: Ethernet "on the Fly"

- Process data is extracted and inserted on the fly
- Process data size per slave almost unlimited (1 Bit…60 Kbyte, if needed using several frames)
- Compilation of process data can change in each cycle, e.g. ultra short cycle time for axis, and longer cycles for I/O update possible
- In addition asynchronous, event triggered communication

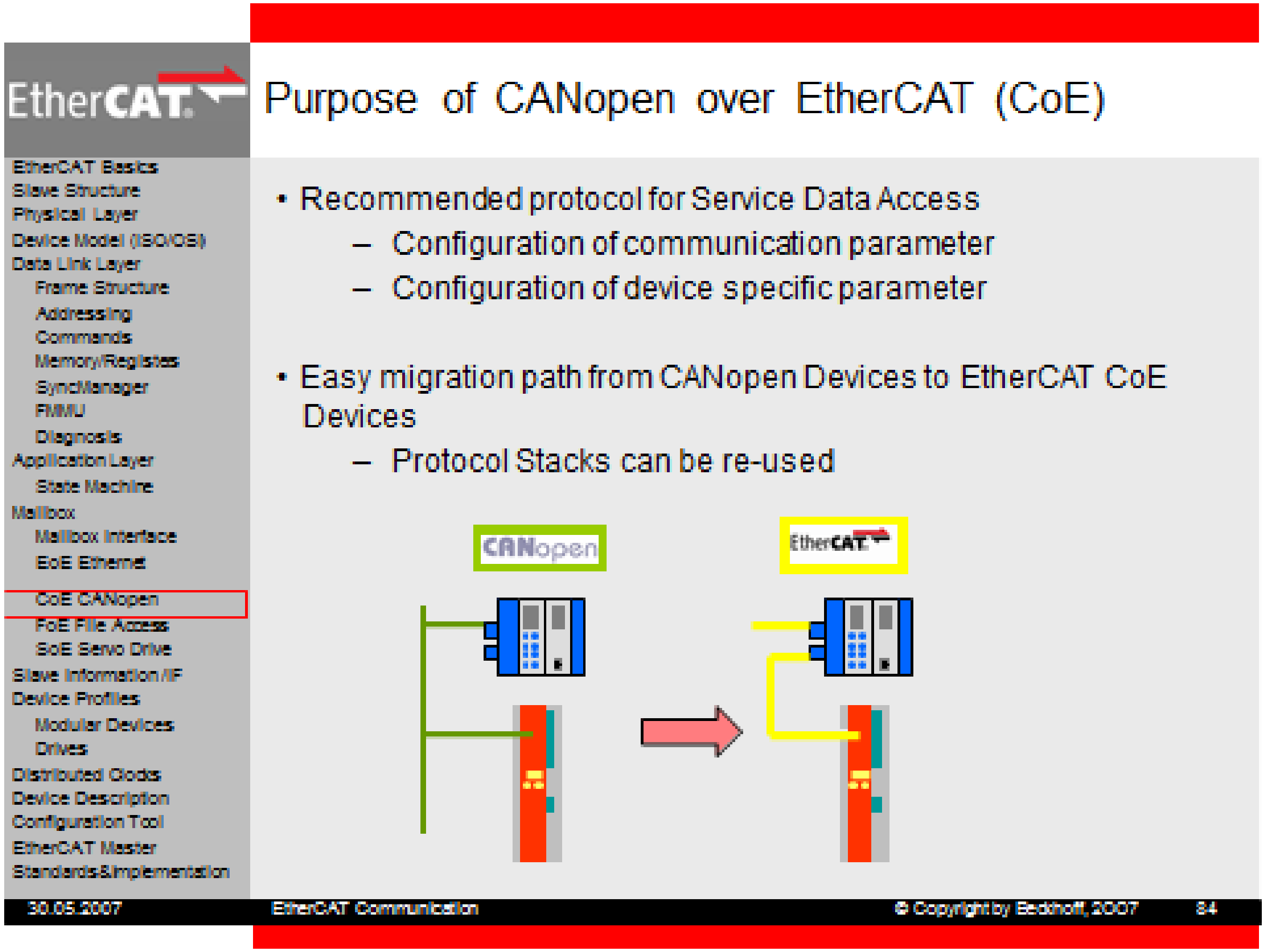
30.05.2007 EtherCAT Communication © Copyright by Beckhoff, 2007 5



Application
....
Appli-cation
HTTP, FTP, ..
Application e.g. CiA402 Drive Profile
RD / WR
...
File Access
TCP, UDP
IP
Object Dictionary
PDO Mapping
SDO
PDO
VoE
FoE
EoE
CoE
CoE
Mailbox DLL
Application Layer (AL)
Process Data Interface (µC, SSI, I/O)
SII EEPROM
Registers
Mailbox
Process Data
ESC Address Space (DPRAM)
0x0000
0x1000
SyncMan0 MbxOut
SyncMan1 MbxIn
SyncMan2
SyncMan3
FMMU n
FMMU n
EtherCAT Slave Controller (ESC)
EtherCAT Processing Unit and Auto-Forwarder with Loop Back
PHY Management
MII / EBUS Port 0
MII / EBUS Port 3
MII / EBUS Port 2
MII / EBUS Port 1
Data Link Layer (DL)
RJ45
Trafo
PHY
PHY
Trafo
RJ45
LVDS
Con
Physical Layer (PL)
FCS
More Datagrams..
WKC
Mailbox Data
FPWR
HDR
WKC
Process Data
LRW
HDR
ECAT HDR
EtherNET HDR

CAN over EtherCAT

➢ **Controller Area Network (CAN) Standards: CAN DS301 and DSP402**



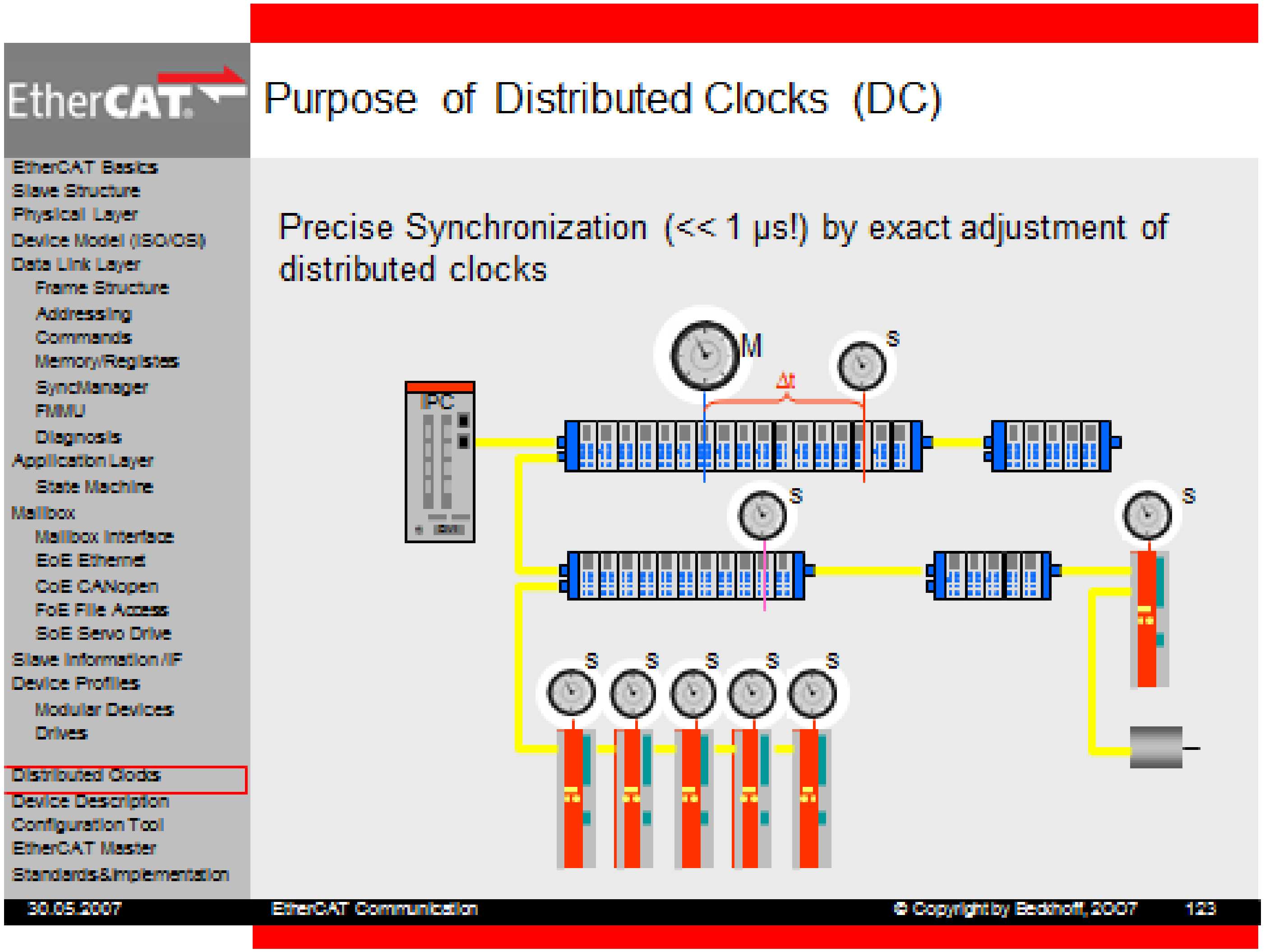
Purpose of CANopen over EtherCAT (CoE)

- Recommended protocol for Service Data Access
  - Configuration of communication parameter
  - Configuration of device specific parameter
- Easy migration path from CANopen Devices to EtherCAT CoE Devices
  - Protocol Stacks can be re-used

30.05.2007 EtherCAT Communication © Copyright by Beckhoff, 2007 84

Distributed Clock (DC)

- DC Clock Syncs to First Servo Drive Clock



Purpose of Distributed Clocks (DC)

Precise Synchronization (<< 1 µs!) by exact adjustment of distributed clocks

30.05.2007 EtherCAT Communication © Copyright by Beckhoff, 2007 123

Distributed Clock (DC)

- **DC Provides Synchronized Communication with Slaves**

  Distributed Clock = PPMAC Real Time Interrupt (RTI), (default that equals Servo Update Rate)

  Typical EtherCAT Distributed Clock settings, or servo update rates.

  - 500 Hz update rate (robot arm)
  - 1 kHz update rate (typical for most servo drives)
  - 2 kHz update rate (available on some servo drives)
  - 4 kHz update rate (available on fewer servo drives)
  - 8 kHz update rate (Delta Tau Motion Machine, future servo drives)

- **Process Data Objects are updated at the DC, or Servo Update Rate**

PDOs versus SDOs

- **Process Data Object (PDO) – DC Update Rate**
  - PDO's are cyclically updated at the Real Time Interrupt (if **Ecat[*i*].ServoExtension = 0**).
  - PDOs are mapped through System Setup.
  - PDO communication begins when **Ecat[*i*].Enable** is set to 1.
  - DC Mode = updated every servo cycle, generally used for closed loop motion control
  - Freerun Mode = asynchronous update, generally used for I/O

- **Service Data Object (SDO)**

  Read/Write access to objects asynchronously using **ecatsdo()** commands
  - Used to configure the slaves during setup
  - Used for troubleshooting
  - Query an Object (e.g. 0x6041, Status Word, to determine if logic power is applied to slaves)
  - Can be considered "startup commands" for EtherCAT devices

- **When to Use PDO/SDO Communication**

  It is important to use the correct method of data transfer for all EtherCAT communications.
  - Use PDOs for cyclic synchronous update of data, e.g. for servo drives
  - Use SDOs for one time, or occasional peeking, or poking at Objects using **ecatsdo()**
  - Do not overuse SDOs while EtherCAT is enabled; it may interfere with the synchronized nature of the Distributed Clock. If used while **Ecat[*i*].Enable=1**, all use of SDO communication is at your own risk. It is best to carry out all SDO communication while **Ecat[*i*].Enable=0**

SDO Read/Write

- **SDO Read/Write**

  Read/Write access to Objects via SDO communications

  - **L0 = ecatsdo(*rw, slave*, $*index*, *subindex*, *value*, *master*)**; **rw =1** to read, **rw =0** to write
  - **L0** holds read value.

- **Examples of SDO Read/Write**

  Example SDO read – read slave 0's status

  - **L0 = (1, 0, $6041, 0, 27, 0)**; read, slave 0, StatusWord(6041), sub-index 0, value, master 0
  - Then query **L0** to access the value that was read

  Example SDO write – set slave 1's mode of operation to cyclic synchronous position mode

  - **L0 = (0, 1, $6060, 0, 8, 0)**; write, slave 1, ModeOfOp(6060), sub-index 0, CSP(8), master 0

EtherCAT Capabilities

- **Synchronous and Asynchronous Communication**
  DC sync Mode, or Freerun Mode – Servos can be in Sync mode while I/O is in Async mode

- **Maximum Device Distance**
  Maximum 100m cable (CAT6) distance between devices (longer distance requires fiber optics)

- **Maximum Number of Devices**
  Theoretical max of 65,535 devices (PPMAC max is 256, some customer applications use 90+ devices)

- **CAN over EtherCAT (CoE)**
  Controller Area Network (CAN) over EtherCAT (CoE) - PPMAC uses for Servo Drives and I/O)

- **Ethernet over EtherCAT (EoE)**
  Not currently implemented; used for remotely setting up devices, as opposed to using USB conn.

- **File over EtherCAT (FoE)**
  PPMAC uses this protocol to update Device Firmware, using **sii_write()** command

- **Fail Safe over EtherCAT, or Field Safety over EtherCAT (FSoE)**
  Currently: Limited implementation on PPMAC, future implementation planned, ongoing dev.

EtherCAT Setup

PPMAC EtherCAT Stack (Implementation)

There are many different implementations of EtherCAT:

- **EtherLab Master Stack (2008 to Present on EtherCAT-Enabled PPMACs)**
  Open source Real Time EtherCAT kernel modules for Linux, www.etherlab.org

- **Acontis Master Stack (On CK3E and NY51[]-A)**
  Proprietary EtherCAT kernel modules, www.acontis.com

Check which Master Stack implementation you are using, enter into the terminal window:

```
Sys.EcatType
```
Power PMAC Script

- If **Sys.EcatType = 0** (EtherLab Master Stack in use)
- If **Sys.EcatType = 1** (Acontis Master Stack in use)

Note

All examples and setup information depicted herein utilize the EtherLab Master Stack implementation of EtherCAT on the PPMAC Motion Controller.

Overview – EtherLabs

This is the typical procedure for setting up the system from scratch:

- **Step 1: $$$***, save, $$$ (clear the PMAC, not necessary if continuing from existing setup)**
- **Step 2: Set the System Global Clock (generally equal to the Servo Update Rate)**
- **Step 3: Update the slave device files (ESI.xml device files)**
- **Step 4: Right click, "Configure the Master"**
- **Step 5: Set Distributed Clock (DC) to match Servo Update Rate**
- **Step 6: Set Startup (set Interpolation & Cyclic Sync Mode, e.g. Pos., Vel., or Torque)**
- **Step 7: Map the Input and Output PDOs (Process Data Objects)**
- **Step 8: Setup the Motors for each slave drive**
- **Step 9: Tune Motors (when using CS Velocity, or CS Torque modes)**
- **Step 10: Right click, "Export EtherCAT Variables"**
- **Step 11: Right click, "Generate Configuration File"**
- **Step 12: save, $$$, and then test your EtherCAT setup.**

Overview – EtherLabs from Config

Use the following steps to set up the system from an existing *.cfg file:

- Step 1: $$$***, save, $$$ (clear the PMAC)
- Step 2: Right click on the file you want to download, “Check To Download Config File”
- Step 3: Right click on the Configuration folder, “Download Config Files”
- Step 4: save
- Step 5: $$$, and then test your EtherCAT setup

Making Use of Template.tpl Files

Choose which PPMAC structures backup to pp_custom_save.cfg, copy/paste them to your .cfg:

- **Step 1: Double click on the, "pp_custom_save.tpl file (to open the file for editing)**

```
backup Sys.
backup Motor[1]..
backup EncTable[0]..
backup Ecat[0].
```

Power PMAC Script

- **Step 2: From the IDE Main Menu, left click, File, "Save All"**
- **Step 3: Right click on the Project, "Build and Download All Programs"**
- **Step 4: save**
- **Step 5: Double click the, "pp_custom_save.cfg" file to view the configuration data**
- **Step 6: Right click the Configuration folder, Add, "New Item", to add a Custom Config file**
- **Step 7: Copy and Paste the contents to your own configuration file, "MyConfig.cfg"**
- **Step 8: From the IDE Main Menu, left click, File, "Save All"**

EC Engineer Setup

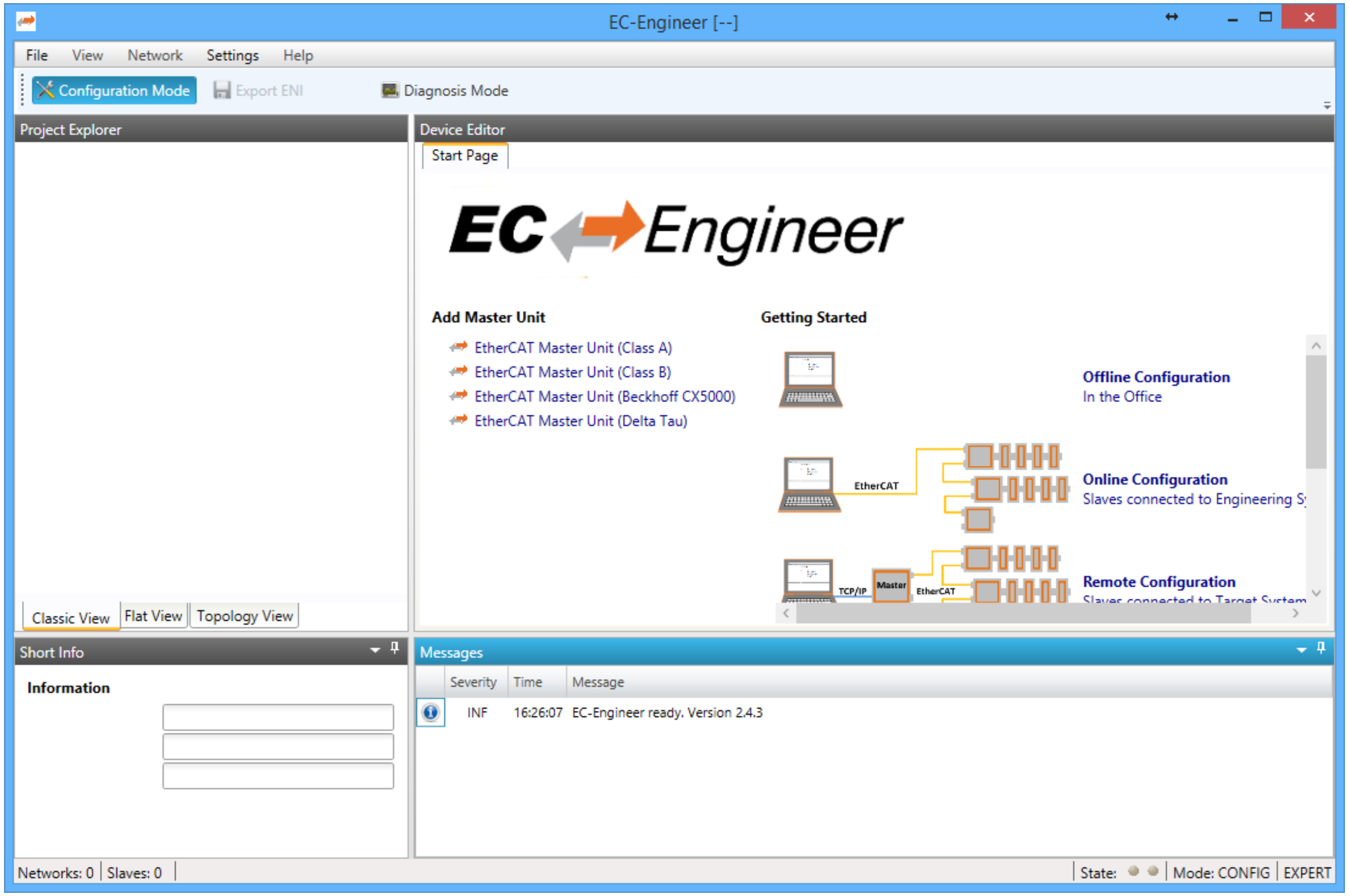
EC Engineer

- **EC Engineer uses an “ESI” file**
  EtherCAT Slave Information File
  Describes capabilities and features of a specific piece of hardware

- **Design your full EtherCAT Network in EC Engineer**

- **EC Engineer generates an “ENI” file (Required for Acontis)**
  EtherCAT Network Information File
  Describes what information will be passed over EtherCAT between various devices
  Describes expected modes of operation, based off of potential modes
  Imports into Power PMAC IDE

EC Engineer



Add Master Unit

- EtherCAT Master Unit (Class A)
- EtherCAT Master Unit (Class B)
- EtherCAT Master Unit (Beckhoff CX5000)
- EtherCAT Master Unit (Delta Tau)

Getting Started

Offline Configuration
In the Office

Online Configuration
Slaves connected to Engineering S

Remote Configuration
Slaves connected to Target System

Information

Severity	Time	Message
INF	16:26:07	EC-Engineer ready. Version 2.4.3

Networks: 0 | Slaves: 0 | State: ● ● | Mode: CONFIG | EXPERT

EC Engineer

1. **Add a Master Unit**
   Power PMAC is a Class A Master Unit
   Select “Delta Tau” from the Start Page to have some features already configured

2. **Add desired slaves**

3. **EC Engineer generates an “ENI” file**
   1. EtherCAT Network Information File
   2. Describes what information will be passed over EtherCAT between various devices
   3. Describes expected modes of operation, based off of potential modes
   4. Imports into Power PMAC IDE

EC Engineer

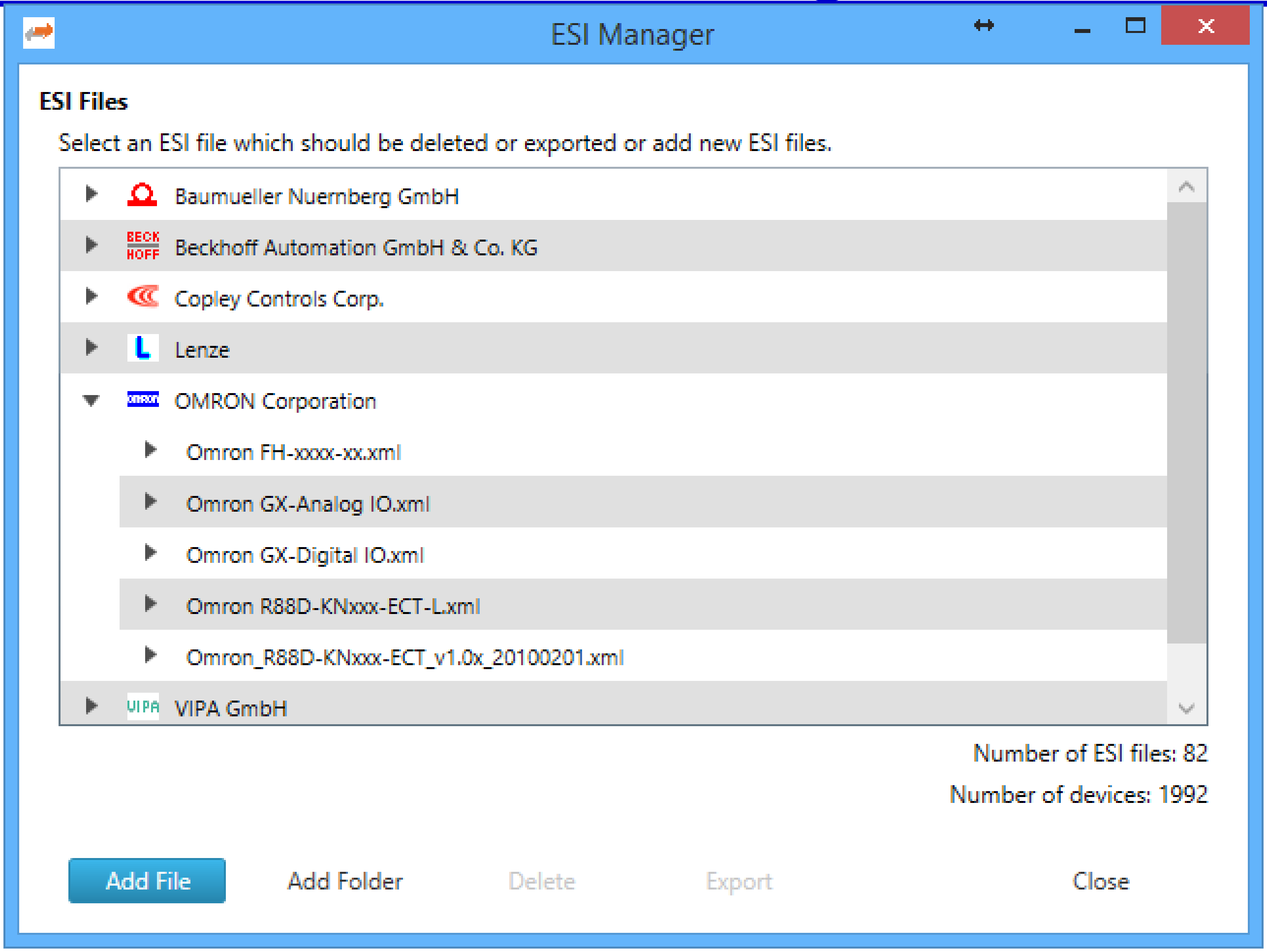
1. **Select Local or Remote System**
   We will use “Remote System” by inputting the IP Address of the units

2. **Right Click “Class-A Master” and select “Scan EtherCAT Network”**

3. **If that does not work, select “Append Slave” and select your slave from the list**

4. **If your slave is not in the list, go to the ESI Manager (File -> ESI Manager)**

ESI Manager



ESI Manager

ESI Files

Select an ESI file which should be deleted or exported or add new ESI files.

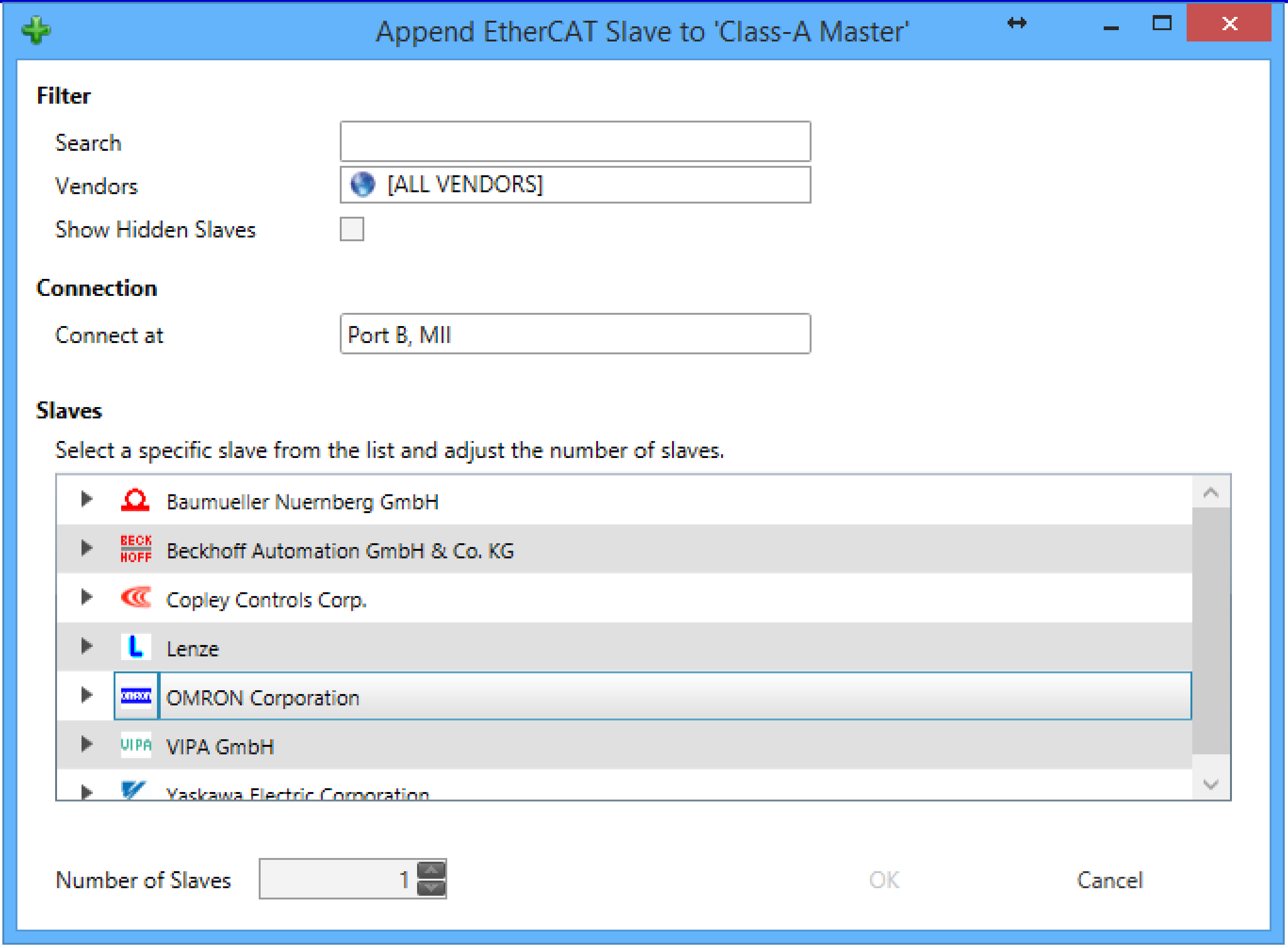
- Baumueller Nuernberg GmbH
- Beckhoff Automation GmbH & Co. KG
- Copley Controls Corp.
- Lenze
- OMRON Corporation
  - Omron FH-xxxx-xx.xml
  - Omron GX-Analog IO.xml
  - Omron GX-Digital IO.xml
  - Omron R88D-KNxxx-ECT-L.xml
  - Omron_R88D-KNxxx-ECT_v1.0x_20100201.xml
- VIPA GmbH

Number of ESI files: 82

Number of devices: 1992

Select either “Add File” (for one ESI file) or “Add Folder” (for multiple) and add the ESI Files provided by the device manufacturers

Append Slave



Append EtherCAT Slave to 'Class-A Master'

Filter

Search	
Vendors	[ALL VENDORS]
Show Hidden Slaves	☐

Connection

Connect at	Port B, MII

Slaves

Select a specific slave from the list and adjust the number of slaves.

- Baumueller Nuernberg GmbH
- Beckhoff Automation GmbH & Co. KG
- Copley Controls Corp.
- Lenze
- OMRON Corporation
- VIPA GmbH
- Yaskawa Electric Corporation

Number of Slaves: 1

After adding the ESI file, your slave should now show up under the manufacturer. Continue for other slave devices in the ring

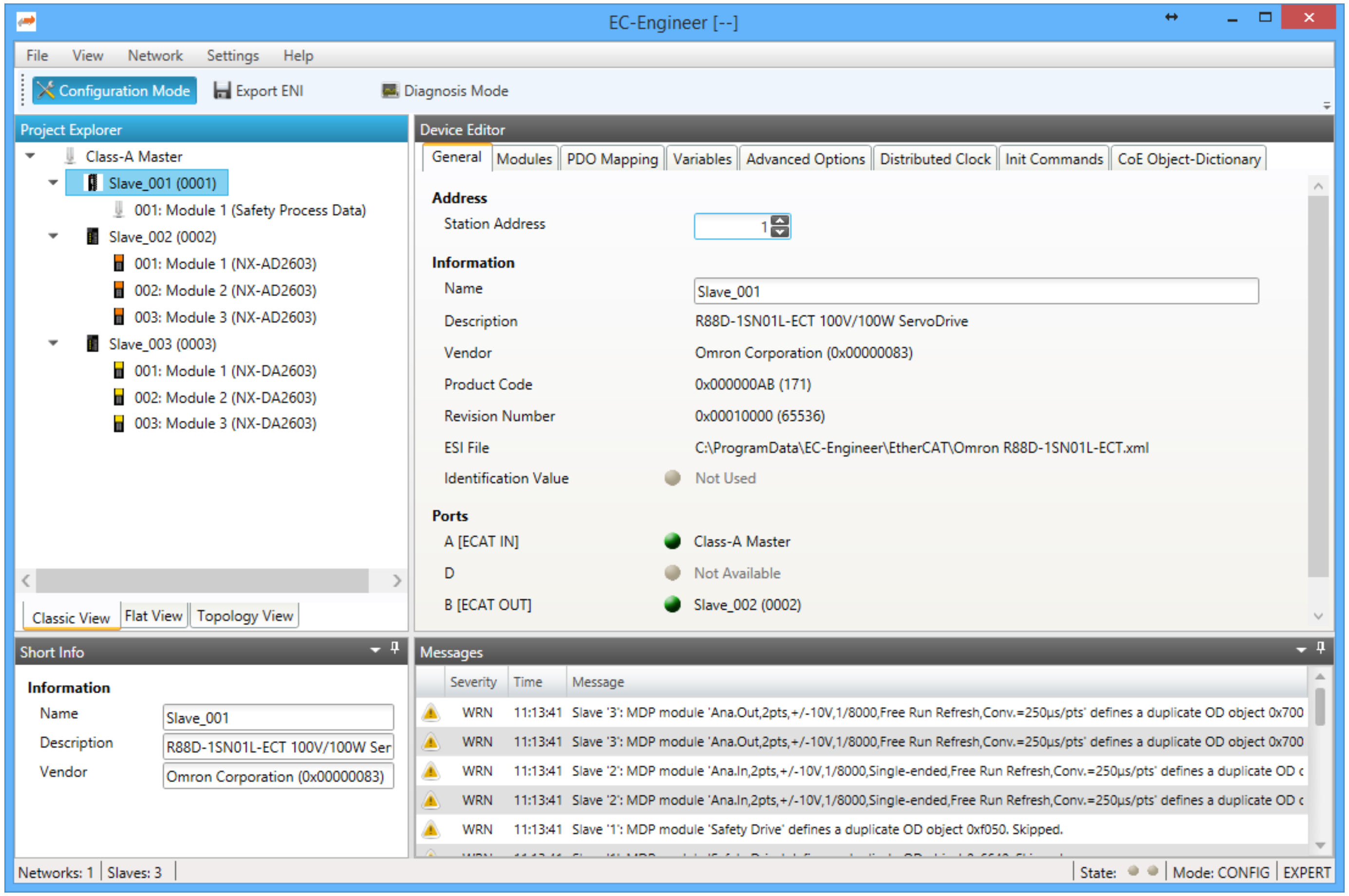
EC Engineer

- **Add new devices to Project**
  "Append Device" adds it onto the end of the ring
  "Insert Slave After" allows you to add devices into the middle of the ring

- **Copy Slave allows you to quickly add multiple of the same devices**

- **Drives vs EtherCAT Couplers**
  Drives will show up as their own ECAT Devices to be configured
  -Can be for one or more axes
  -Can also have I/O capabilities or Vision, depending on type of drive
  EtherCAT Couplers will show up to configure multiple modules (generally I/O or Safety)
  -Modules are then added to the coupler depending on what hardware is present
  -Coupler handles all the EtherCAT Data Transfer, so all setup is done to it

EC Engineer



- Class-A Master
  - Slave_001 (0001)
    - 001: Module 1 (Safety Process Data)
  - Slave_002 (0002)
    - 001: Module 1 (NX-AD2603)
    - 002: Module 2 (NX-AD2603)
    - 003: Module 3 (NX-AD2603)
  - Slave_003 (0003)
    - 001: Module 1 (NX-DA2603)
    - 002: Module 2 (NX-DA2603)
    - 003: Module 3 (NX-DA2603)

Address

Station Address	1

Information

Name	Slave_001
Description	R88D-1SN01L-ECT 100V/100W ServoDrive
Vendor	Omron Corporation (0x00000083)
Product Code	0x000000AB (171)
Revision Number	0x00010000 (65536)
ESI File	C:\ProgramData\EC-Engineer\EtherCAT\Omron R88D-1SN01L-ECT.xml
Identification Value	Not Used

Ports

A [ECAT IN]	Class-A Master
D	Not Available
B [ECAT OUT]	Slave_002 (0002)

Information

Name	Slave_001
Description	R88D-1SN01L-ECT 100V/100W Ser
Vendor	Omron Corporation (0x00000083)

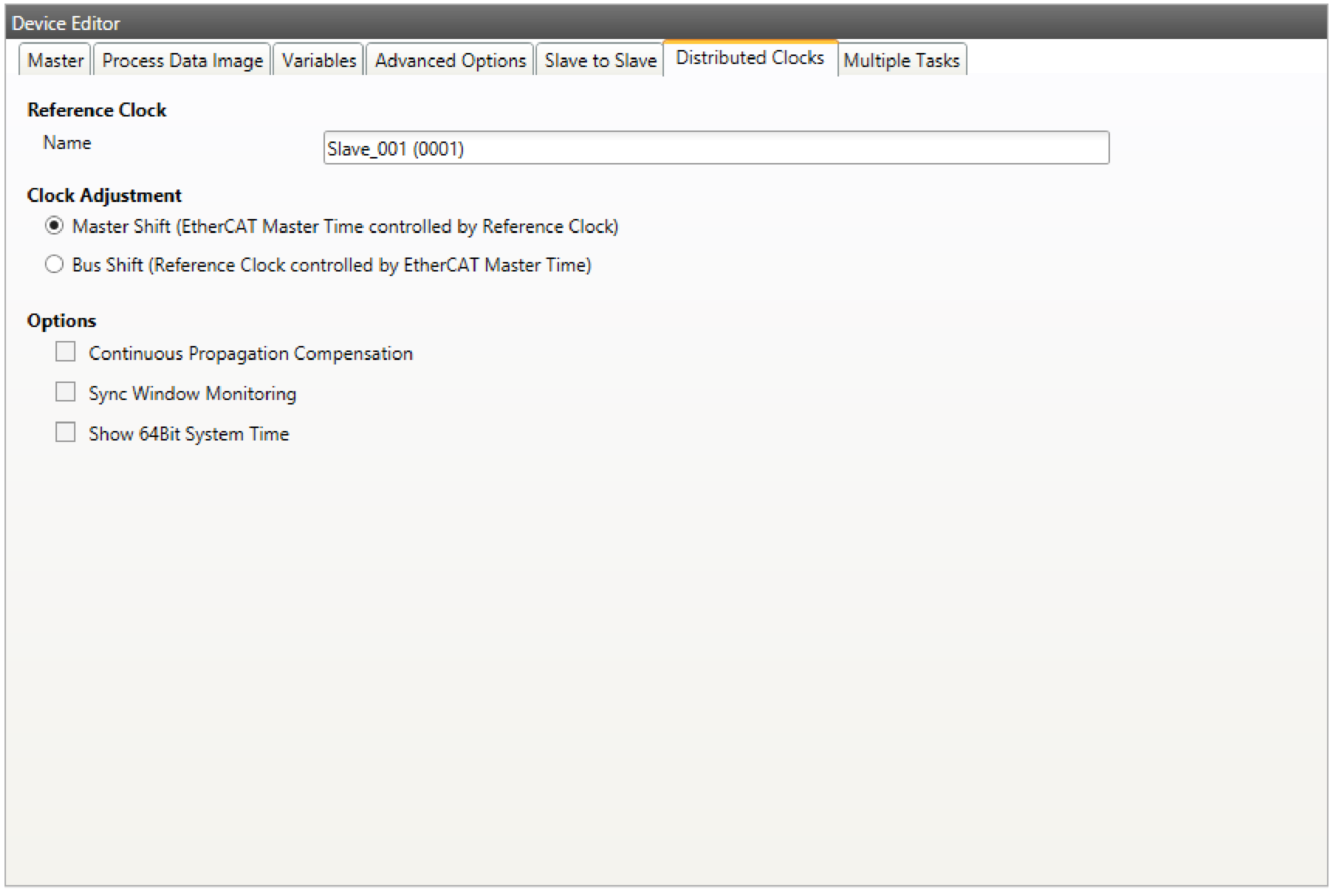
Severity	Time	Message
WRN	11:13:41	Slave '3': MDP module 'Ana.Out,2pts,+/-10V,1/8000,Free Run Refresh,Conv.=250µs/pts' defines a duplicate OD object 0x700
WRN	11:13:41	Slave '3': MDP module 'Ana.Out,2pts,+/-10V,1/8000,Free Run Refresh,Conv.=250µs/pts' defines a duplicate OD object 0x700
WRN	11:13:41	Slave '2': MDP module 'Ana.In,2pts,+/-10V,1/8000,Single-ended,Free Run Refresh,Conv.=250µs/pts' defines a duplicate OD c
WRN	11:13:41	Slave '2': MDP module 'Ana.In,2pts,+/-10V,1/8000,Single-ended,Free Run Refresh,Conv.=250µs/pts' defines a duplicate OD c
WRN	11:13:41	Slave '1': MDP module 'Safety Drive' defines a duplicate OD object 0xf050. Skipped.

Networks: 1 | Slaves: 3 | State: | Mode: CONFIG | EXPERT

Eventually all of your devices should be added and show up on the left side of the screen. Be sure that they match the order of the devices in your ring

Configuring the EtherCAT Master

- **If "Delta Tau" was selected, most of the set up should be correct**
  Make sure that Master Shift is selected, not Bus Shift, and that the Reference Clock is the first slave device



Reference Clock
Name: Slave_001 (0001)

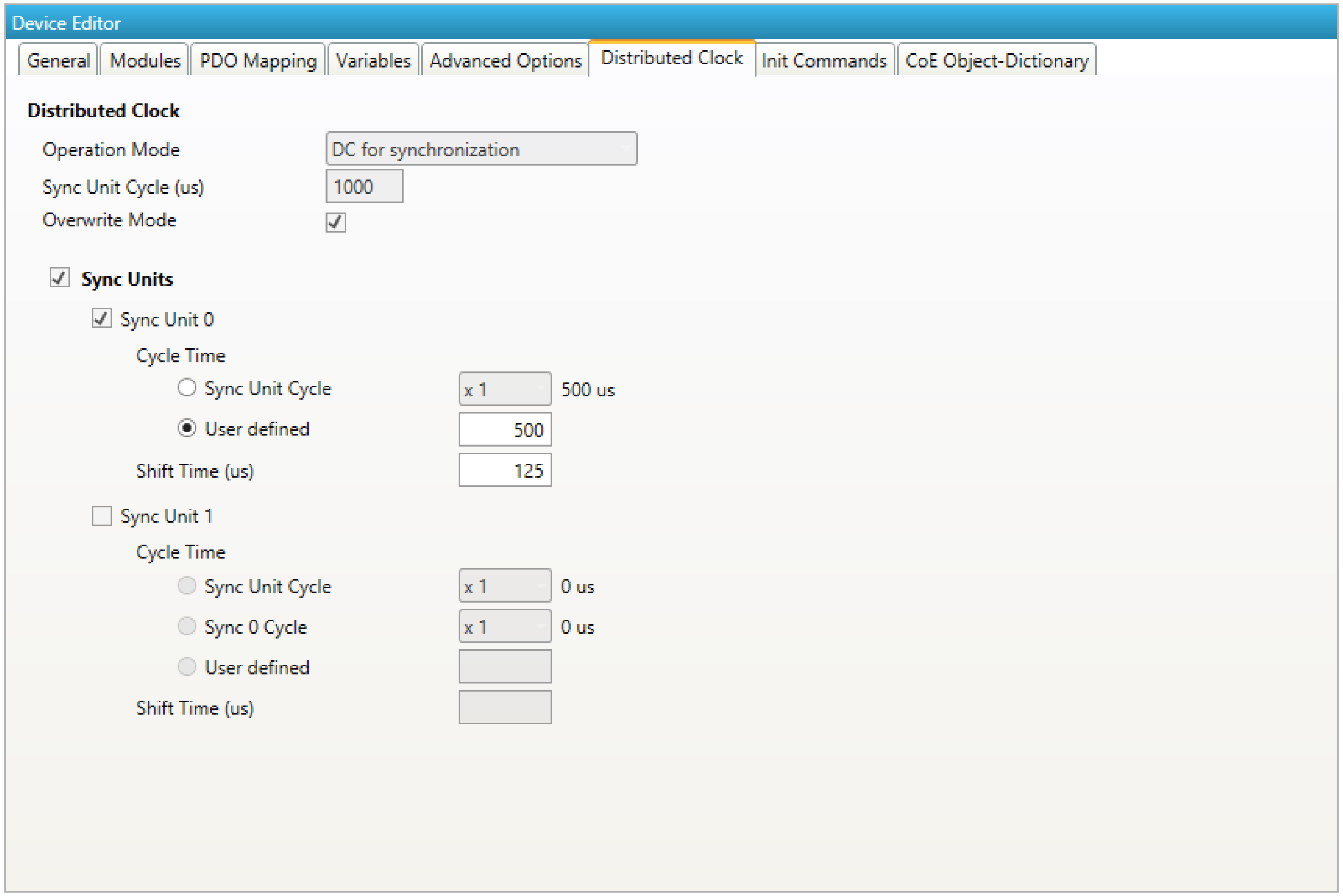
Clock Adjustment
● Master Shift (EtherCAT Master Time controlled by Reference Clock)
○ Bus Shift (Reference Clock controlled by EtherCAT Master Time)

Options
☐ Continuous Propagation Compensation
☐ Sync Window Monitoring
☐ Show 64Bit System Time

This will force the first Slave to be the reference for the Distributed Clock
The PMAC will then adjust its clocks to keep up with the drive, rather than forcing the drive to attempt to do the same

Configuring a Drive – Distributed Clock

- **The first tab to be set up for a drive is the Distributed Clock tab**
  Enable Overwrite Mode to allow for user configuration of the clock
  Set the User Defined Cycle Time to determine your EtherCAT Clock Speed
  Set the Shift Time to 125 us or half of the Cycle Time, whichever is lesser



Distributed Clock

Operation Mode	DC for synchronization
Sync Unit Cycle (us)	1000
Overwrite Mode	✓

✓ **Sync Units**

✓ Sync Unit 0

Cycle Time

Sync Unit Cycle	x 1	500 us
[✓ User defined]	500	
Shift Time (us)	125	

☐ Sync Unit 1

Cycle Time

Sync Unit Cycle	x 1	0 us
Sync 0 Cycle	x 1	0 us
User defined		
Shift Time (us)		

Configuring a Drive – PDO Mapping

- **To determine which PDOs are needed, first decide what mode will be used to control the motor**
- **Choose between Cyclic Position, Velocity, and Torque Modes**

  For all modes, you need the inputs 0x6041 (Statusword), 0x6064 (Position Actual Value) and the output 0x6040 (Controlword)

  Cyclic Position Mode needs output 0x607A (Target position)
  Cyclic Velocity Mode needs output 0x60FF (Target velocity)
  Cyclic Torque Mode needs output 0x6071 (Target torque)
- **Select the one PDO Mapping that has all parameters that you need and as few other parameters as possible**

  Depending on your system, you may need more PDOs than those 4—some common ones are touch probes, I/O, velocity or torque limits, or even a second encoder

  The fewer parameters there are for a PDO Mapping, the faster the data can be transferred and the higher the clock speed that can be selected

  If no mapping exists with everything that you need, you can create your own by adding PDOs to an existing mapping

  -Pre-made mappings are pre-optimized for the hardware
  -Using a custom mapping will take significantly longer to transfer the data and may affect performance

Configuring other Devices

- **Other devices will all function similarly to a drive, but with different PDOs**
- **Check with device manual or manufacturer to see if it needs a specific value on initialization for EtherCAT**
- **For I/O, make sure that mappings are selected for all modules connected to a given coupler**
  There may be Input PDO Mappings for an Output Coupler and Output PDO Mappings for an Input Coupler, but generally Input Devices will have Input PDOs and Output Devices will have Output PDOs

Transferring the Data to the PMAC

- **Once the full EtherCAT Network has been characterized, click on “Export ENI” to generate the ENI File for the IDE to read**
- **Set the PMAC Clock such that it is an integer multiple of the EtherCAT clock (for a pure EtherCAT system, they can be set to the same value)**

  You may need to issue a “save” and a “$$$” command to change the clock settings
- **Open the Power PMAC IDE and then go to the EtherCAT Master in System Setup**

  On the right side, Ecat[0].Error should be “No Error”, and under Slave Status the correct number of devices should show up

  -If not, try issuing “ecat reset” and then “ecat slaves” in the terminal window

  -If it still does not show up and the system claims to have an appropriate EtherCAT license for the setup you are using, check the wiring status of all slaves
- **Browse to the ENI File from EC Engineer and Download it to the PMAC**
- **If it downloads successfully and still no errors are detected, “Export EtherCAT Variables” and then “Enable EtherCAT” by right clicking on the master**

EtherCAT Motor Setup

- **With EtherCAT Enabled, an EtherCAT Amplifier should show up in Motor Setup**
  Issue “ecat slaves” to see what devices you have and that the Product ID matches
  Make sure that the Slave Number and Axis Index are correct for your given motor
- **Select the Control Type of either Cyclic Position, Velocity, or Torque with EtherCAT Feedback and Signals**
  Check that this matches the settings initialized on the drive
- **Select the correct PDOs for each of the Hardware Interface Parameters**
  Verify the Slave and Axis Numbers in the PDO if you have multiple drives or channels
  If everything else was done correctly, this should automatically populate correctly
  - -Command Signal Channel → Targetposition, Targetvelocity, or Targettorque
  - -Amplifier Enable Signal Output Channel → Controlword
  - -Amplifier Fault Signal Input Channel → Statusword
  - -Primary Feedback Channel → Positionactualval
- **Accept the page and feel free to use your motor as if it were local!**

Enabling and Disabling EtherCAT

- **ECAT should never be enabled on startup**
  If ECAT is enabled before the PMAC is ready, it may crash
  ECAT will not enable properly until all slaves are online
- **Initialization PLC should be written to enable EtherCAT when the system is ready**

Accessing EtherCAT Data

- **“Export ECAT Variables” should have created a C Program Header and a Script Program Header File with all of your mapped PDOs and given them aliases**
  You can change the aliases or assign new ones as you see fit
  Default Alias indicates which slave device correlates to the parameter, in addition to the PDO Number
- **You can now directly read/write to these parameters, depending on what a given parameter accepts**
  Doing so will populate the changes out over EtherCAT at a rate equivalent to your EtherCAT clock
  You can use these in your PLC or Motion Programs as inputs or outputs to control or be controlled by your PMAC Logic

EtherCAT Troubleshooting

Overview – Troubleshooting

The following objects, or addresses often provide useful troubleshooting information.

- **0x6041** Status Word (provides slave information, particularly for servo drives)
- **0x603F** Error Codes (16-bit error code per, IEC61800-7-301, Table 24)
- Application Layer (AL) Error Codes (IEC61158-6-12, Table 11, IEC61158-600, Table 11)
- Terminal commands:

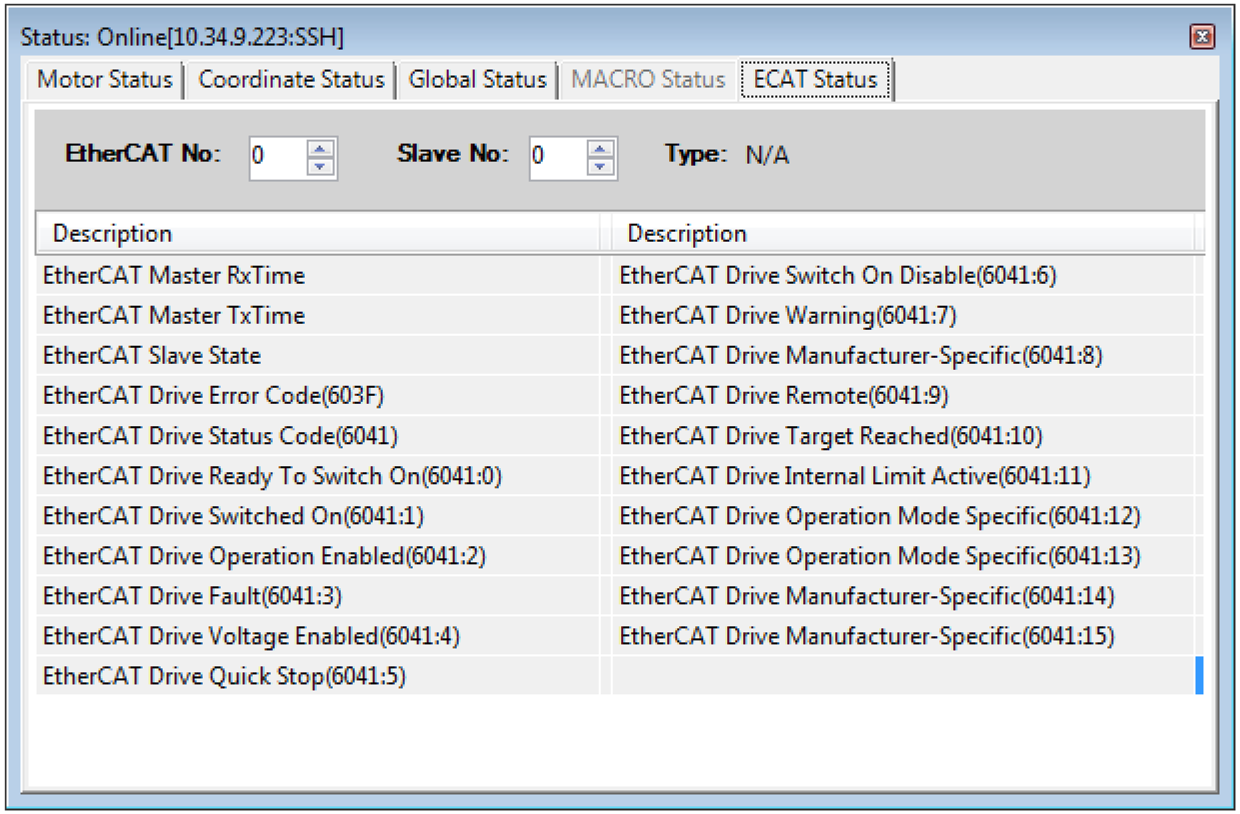
• **ecat slaves**	(check slaves and their state)
• **system ethercat pdos**	(find out what PDOs are mapped)
• **L0=ecatsdo(1, n, $6041, 0, 0, 0); L0**	(to read slave n's StatusWord from L0)
• **L0=ecatsdo(1, n, $603F, 0, 0, 0); L0**	(to read slave n's Error Code from L0)
• **system dmesg**	(Linux message dump)

Source Materials:

- www.ethercat.org (EtherCAT Technology Group)
- www.beckhoff.com

Troubleshooting Information

The Status window provides a reference for the 0x6041 StatusWord, and 0x603F Error Codes



Status: Online[10.34.9.223:SSH]

EtherCAT No: 0 **Slave No:** 0 **Type:** N/A

Description	Description
EtherCAT Master RxTime	EtherCAT Drive Switch On Disable(6041:6)
EtherCAT Master TxTime	EtherCAT Drive Warning(6041:7)
EtherCAT Slave State	EtherCAT Drive Manufacturer-Specific(6041:8)
EtherCAT Drive Error Code(603F)	EtherCAT Drive Remote(6041:9)
EtherCAT Drive Status Code(6041)	EtherCAT Drive Target Reached(6041:10)
EtherCAT Drive Ready To Switch On(6041:0)	EtherCAT Drive Internal Limit Active(6041:11)
EtherCAT Drive Switched On(6041:1)	EtherCAT Drive Operation Mode Specific(6041:12)
EtherCAT Drive Operation Enabled(6041:2)	EtherCAT Drive Operation Mode Specific(6041:13)
EtherCAT Drive Fault(6041:3)	EtherCAT Drive Manufacturer-Specific(6041:14)
EtherCAT Drive Voltage Enabled(6041:4)	EtherCAT Drive Manufacturer-Specific(6041:15)
EtherCAT Drive Quick Stop(6041:5)	

Virtual Motors

Why Use Virtual Motors?

- **Development**

When developing logic for a system, it may be beneficial to verify logic without moving actual motors
Allows for developers to test that a program will perform as expected without a full system

- **Applications**

Some systems will want to use virtual motors in their actual application
Less common, but allows for motor following
Sometimes multiple Encoder Conversion Table entries will be required for one motor, and this allows you to view intermediary steps if desired

Configuring Virtual Motors

```
//To Configure Motors 5-8 as Virtual Motors:
//Speed and Accel
Motor[5].InvAmax=1,1,1,1
Motor[5].MaxSpeed=1000,1000,1000,1000

// Motor CS Def Scale Factors
Motor[5].PosSf=1,1,1,1
Motor[5].Pos2Sf=1,1,1,1

// Set derivative gain term in servo loop to zero
// This is a Type 1 servo (single integration); does not need Kd
Motor[5].Servo.Kvfb=0,0,0,0
Motor[5].FatalFeLimit=0,0,0,0

// Lower proportional gain term from default
Motor[5].Servo.Kp=40,40,40,40

//Hardware Interfacing – Repeat for Motors 6-8 with sequential values for sys.udata[x],.a
Motor[5].pDac=sys.udata[15].a
Motor[5].pLimits=0
Motor[5].Ctrl=sys.PosCtrl
Motor[5].pAmpFault=0
Motor[5].pAmpEnable=0
```

Power PMAC Script

Configuring Virtual Motors

```
//Encoder Conversion Table Entries
EncTable[5].pEnc=sys.udata[15].a
EncTable[5].pEnc1=sys.udata[15].a

EncTable[6].pEnc=sys.udata[16].a
EncTable[6].pEnc1=sys.udata[16].a

EncTable[7].pEnc=sys.udata[17].a
EncTable[7].pEnc1=sys.udata[17].a

EncTable[8].pEnc=sys.udata[18].a
EncTable[8].pEnc1=sys.udata[18].a

EncTable[5].type=1,1,1,1
EncTable[5].index1=0,0,0,0
EncTable[5].index2=0,0,0,0
EncTable[5].index3=0,0,0,0
EncTable[5].index4=0,0,0,0
EncTable[5].index5=0,0,0,0
EncTable[5].MaxDelta=0,0,0,0
EncTable[5].ScaleFactor=1,1,1,1
EncTable[5].TanHalfPhi=0,0,0,0
EncTable[5].CoverSerror=0,0,0,0

EncTable[9].Type=0

Motor[5].ServoCtrl = 1,1,1,1
```

Power PMAC Script

Making a GUI with C#

Basic Program Structure

C#'s syntax is nearly identical to C's. The main difference is the program structure. C# has no "main" function but rather has a **namespace** containing the main **class** which executes when your program starts. A class is like a **struct** but can contain **function**s AND variables, whereas C's **struct**s could only contain variables.

The **class** has a **constructor** that runs when the class is **instantiated** (or a **new** object is made from the class). The **class** also contains **methods** which are basically like functions.

Example of Simple Program:

```
// Hello1.cs
public class Hello1
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, World!");
    }
}
```

C# Code

Note

This section will not deeply explain C# but rather will teach only what is needed to make a simple GUI that communicates with PPMAC.

Basic Program Structure

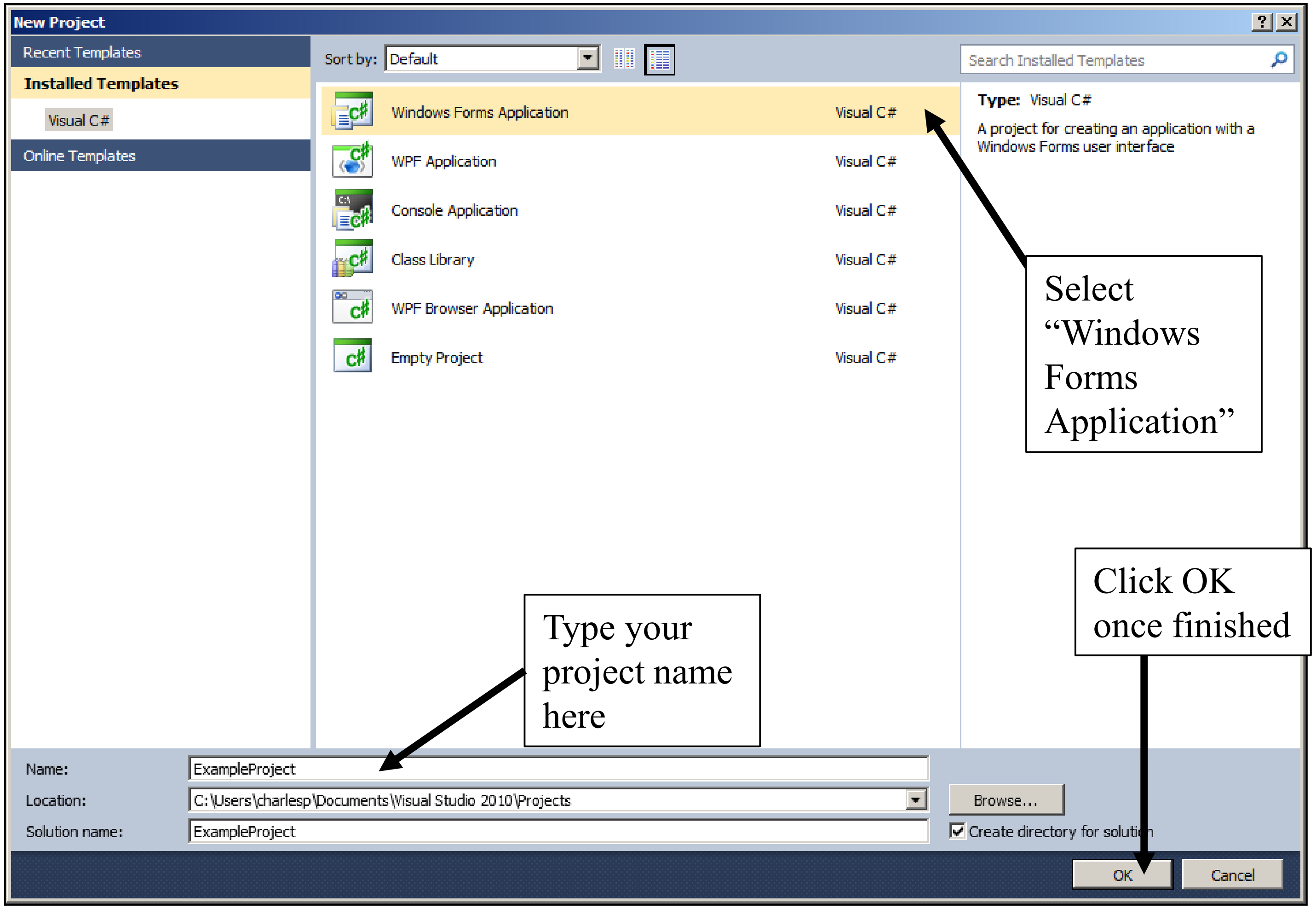
You can put more methods inside your class and just call them by typing the name of the class, then the **.** operator, then the function name, as though the method were just a field in a C structure. Here is a simple example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ExampleBasicProject{
    public class Class1  {
        public static void Main() {
        double b=8.0; // Console.WriteLine prints to screen
        Console.WriteLine("b starts with this value: " + b);
        b = Class1.MyFunc(b);
        Console.WriteLine("Then, we pass it to MyFunc and it becomes: " + b);
        Console.WriteLine("Press any key to exit...");
        Console.ReadLine(); // Reads from user input
        return;
        }
        public static double MyFunc(double input){
            return input * 2.0;}
    }
}
```

C# Code

Using GUI Tools

First we need to learn how to design our GUI in Visual Studio. Let's make a GUI we can use as a general framework for the communications examples. Within Visual Studio, click File→New→Project…, which opens the following dialog box:



New Project

Recent Templates
Installed Templates
Visual C#
Online Templates

Sort by: Default

Search Installed Templates

Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Class Library	Visual C#
WPF Browser Application	Visual C#
Empty Project	Visual C#

Type: Visual C#

A project for creating an application with a Windows Forms user interface

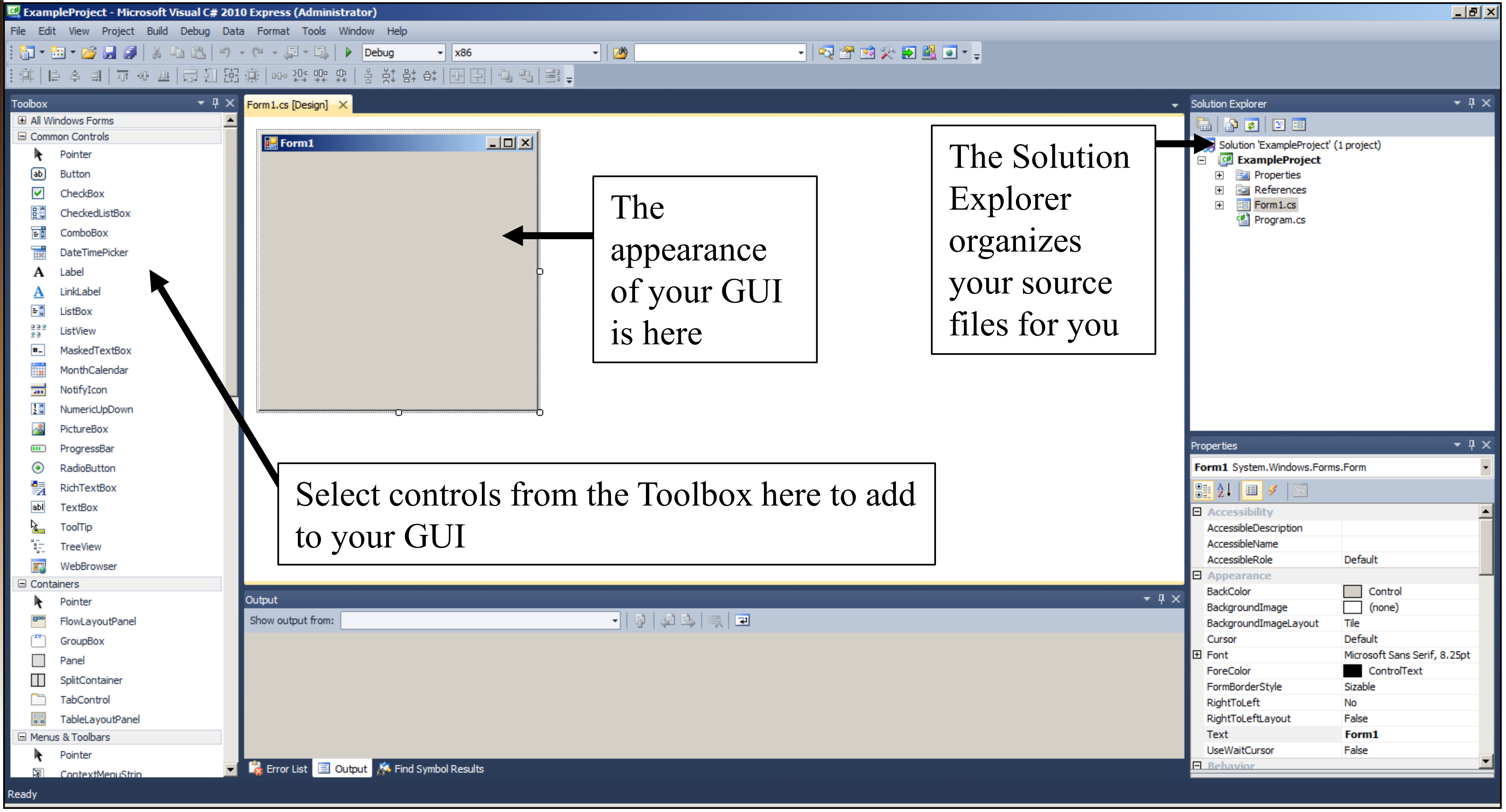
Name: ExampleProject
Location: C:\Users\charlesp\Documents\Visual Studio 2010\Projects
Solution name: ExampleProject

Browse…
☑ Create directory for solution

OK Cancel

Select "Windows Forms Application"
Type your project name here
Click OK once finished

Using GUI Tools



Form1 System.Windows.Forms.Form

Accessibility	
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
Appearance	
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayout	Tile
Cursor	Default
Font	Microsoft Sans Serif, 8.25pt
ForeColor	ControlText
FormBorderStyle	Sizable
RightToLeft	No
RightToLeftLayout	False
Text	**Form1**
UseWaitCursor	False
Behavior	

The appearance of your GUI is here

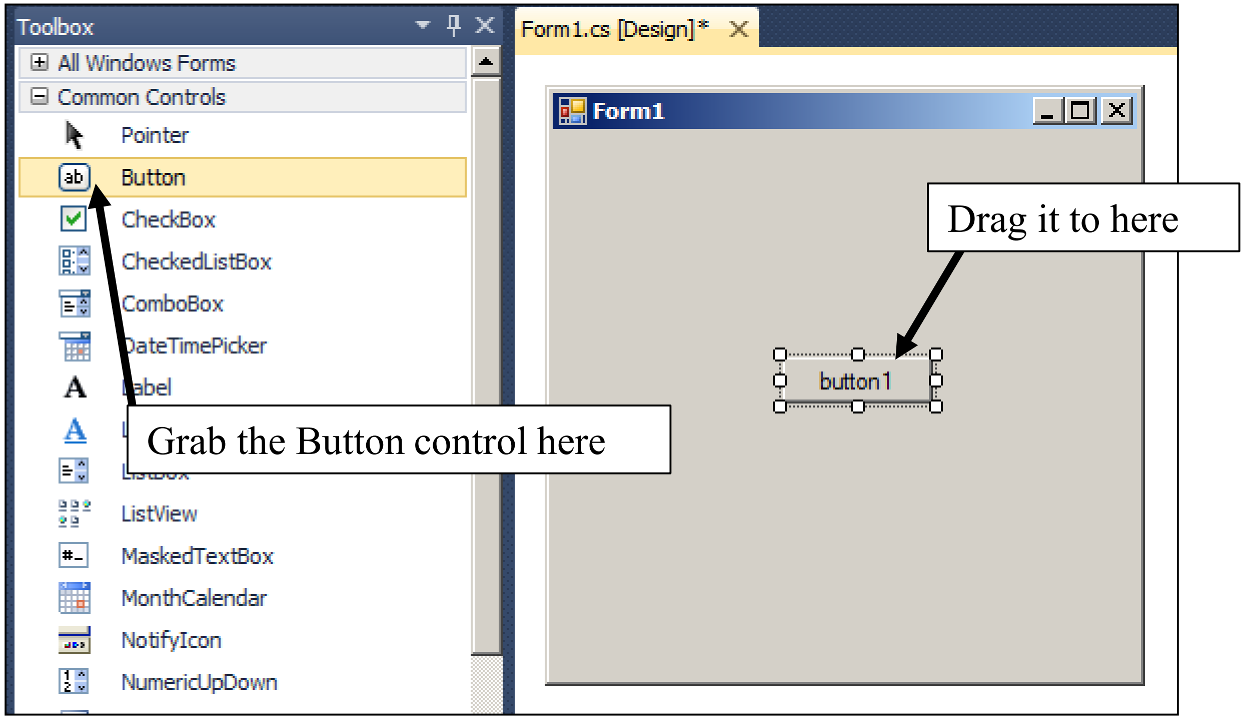
The Solution Explorer organizes your source files for you

Select controls from the Toolbox here to add to your GUI

Using GUI Tools

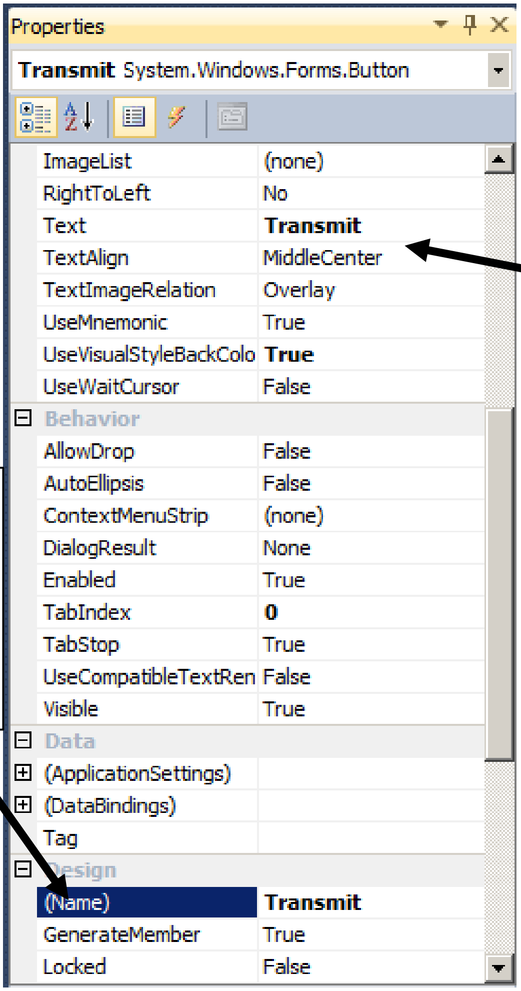
Example: Transmitting Text to PMAC and Reading the Response

Let’s select the Button control from the toolbox and drag it to the GUI frame.



Using GUI Tools

Now let's rename the button in the Properties menu to "Transmit":



Properties

Transmit System.Windows.Forms.Button

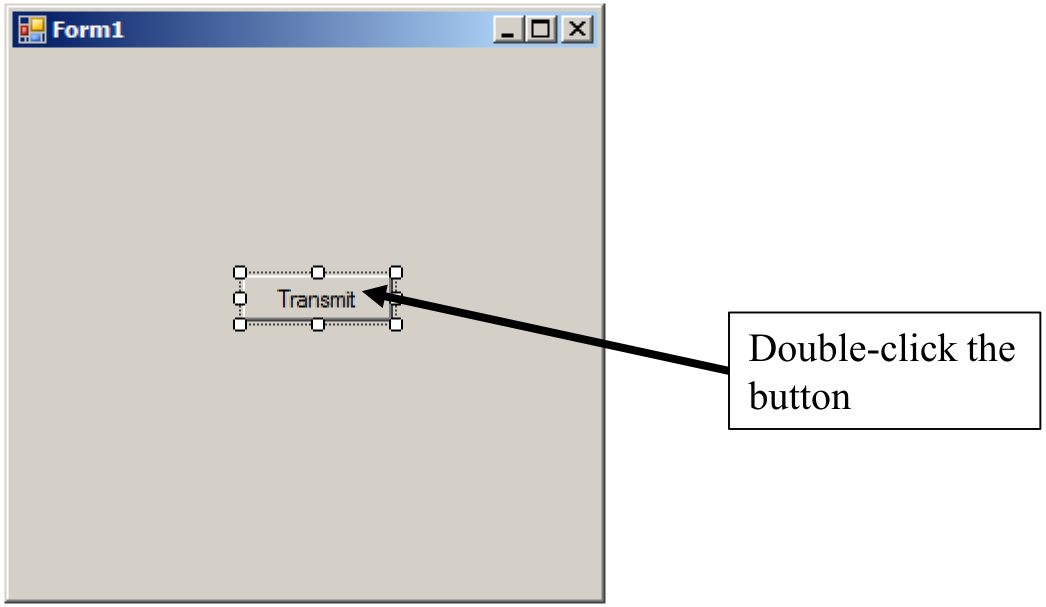
ImageList	(none)
RightToLeft	No
Text	**Transmit**
TextAlign	MiddleCenter
TextImageRelation	Overlay
UseMnemonic	True
UseVisualStyleBackColo	**True**
UseWaitCursor	False
Behavior	
AllowDrop	False
AutoEllipsis	False
ContextMenuStrip	(none)
DialogResult	None
Enabled	True
TabIndex	**0**
TabStop	True
UseCompatibleTextRen	False
Visible	True
Data	
(ApplicationSettings)	
(DataBindings)	
Tag	
Design	
(Name)	**Transmit**
GenerateMember	True
Locked	False

Change the "Text" field to change what text is inside the button

Change the (Name) field to the name of the button when it is referenced inside your source code

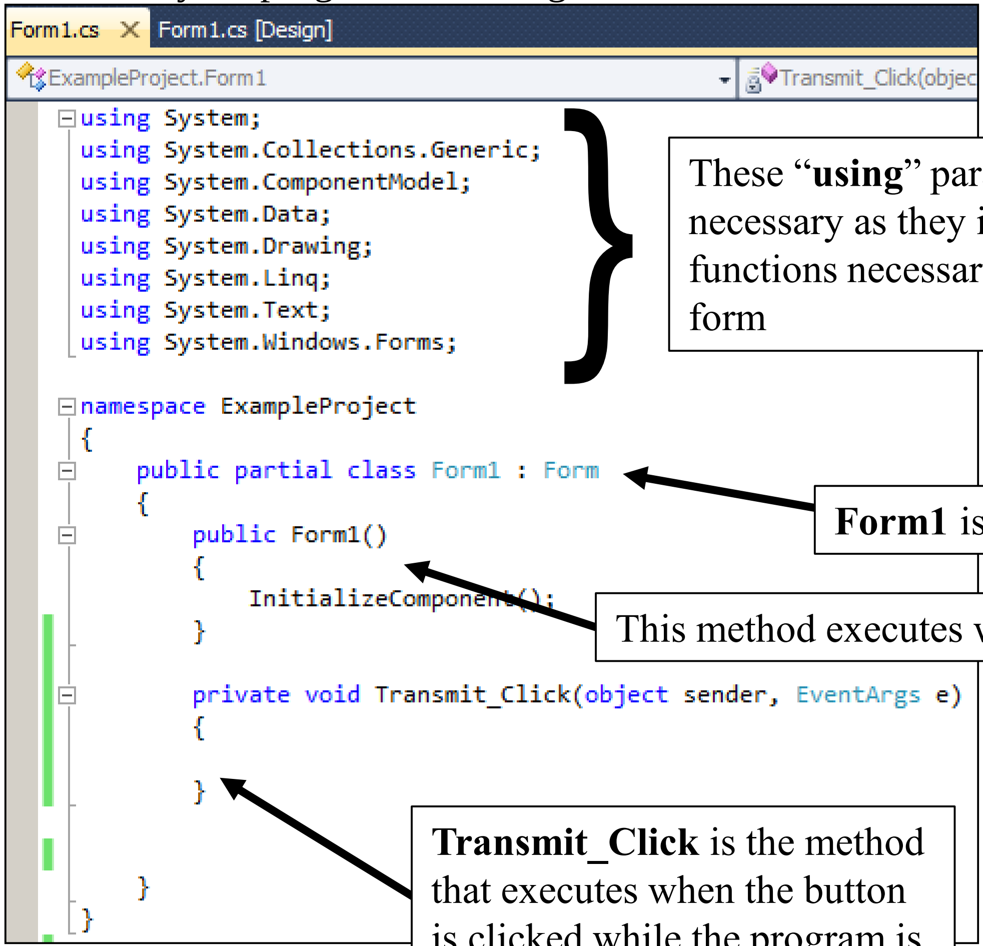
Using GUI Tools

Now double-click the button in the Design Editor:



Using GUI Tools

Double-clicking the button opens up the source code of the method that executes when the button is clicked when your program is running:



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace ExampleProject
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Transmit_Click(object sender, EventArgs e)
        {

        }


    }
}
```

These “**using**” parameters are necessary as they include functions necessary for your form

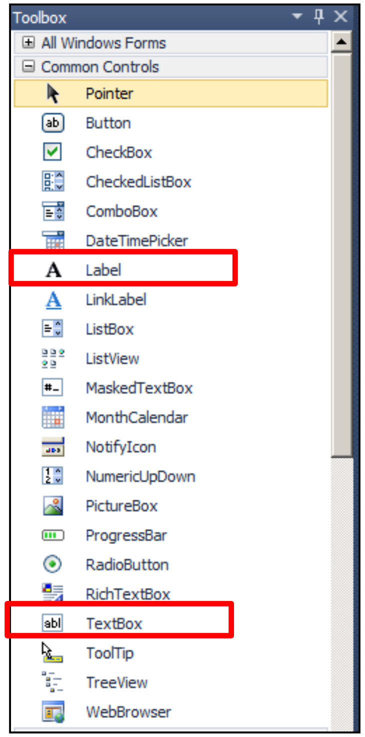
Form1 is your **Form**’s class

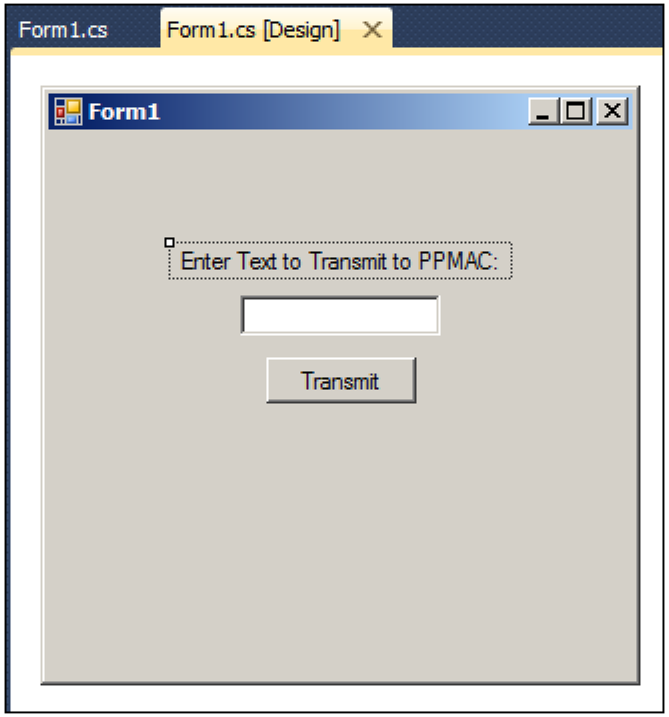
This method executes when your form loads up

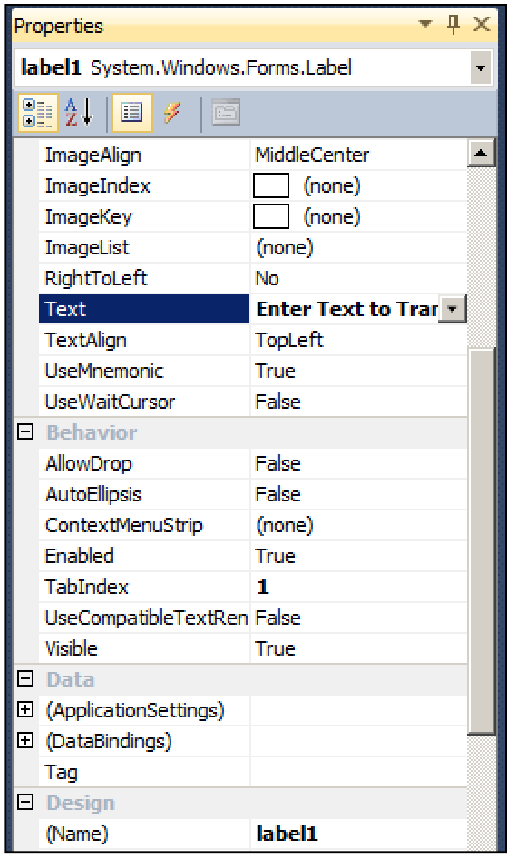
Transmit_Click is the method that executes when the button is clicked while the program is running

Using GUI Tools

Now go back to the Design tab and add a Label and a TextBox (controls highlighted on left). Change the Label to display “Enter text to transmit to PMAC:” (as shown in center below). You can change these by going to Properties and then altering the Label and (Name) fields (shown on right) below.






Properties

label1 System.Windows.Forms.Label

ImageAlign	MiddleCenter
ImageIndex	(none)
ImageKey	(none)
ImageList	(none)
RightToLeft	No
Text	Enter Text to Trar
TextAlign	TopLeft
UseMnemonic	True
UseWaitCursor	False
Behavior	
AllowDrop	False
AutoEllipsis	False
ContextMenuStrip	(none)
Enabled	True
TabIndex	1
UseCompatibleTextRen	False
Visible	True
Data	
(ApplicationSettings)	
(DataBindings)	
Tag	
Design	
(Name)	label1

Then, double-click the TextBox in the center of the Design tab.

Using GUI Tools

Double-clicking the TextBox opens up the method that executes whenever the user changes the text inside the box (it is appended to the bottom of your **Form1** class's code before the closing bracket):

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace ExampleProject{
    public partial class Form1 : Form{
        public Form1() {
            InitializeComponent();}

        private void Transmit_Click(object sender, EventArgs e) {
        }

        private void Form1_Load(object sender, EventArgs e)  { }

        private void textBox1_TextChanged(object sender, EventArgs e)
        {

        }}}
```

C# Code

Inside the textBox1_TextChanged method, we want to store the text the user entered such that we can transmit it to PMAC.

Using GUI Tools

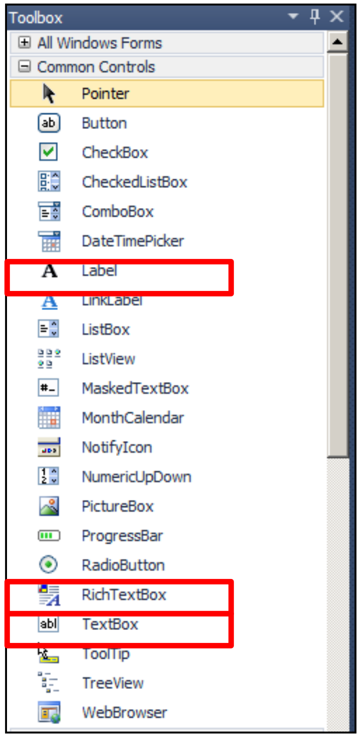
We need to add a String to be globally accessible within the Form1 class and then fill it up with the text from the TextBox.

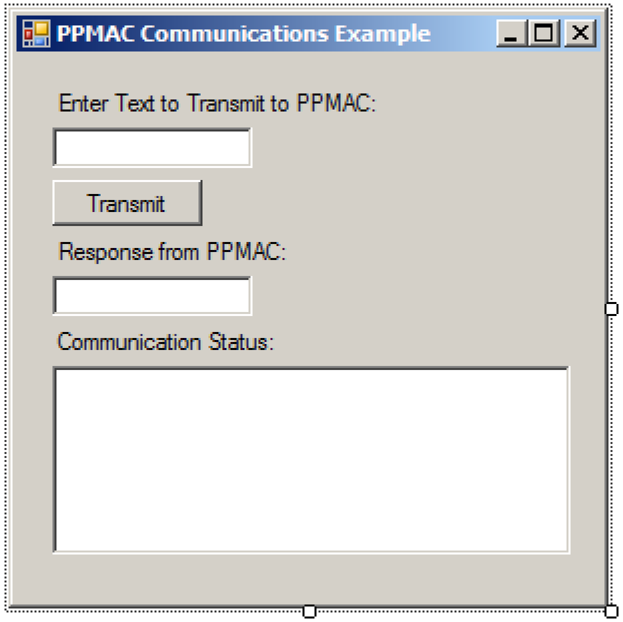
```
namespace ExampleProject
{
    public partial class Form1 : Form
    {
        String commands = String.Empty; // Store the commands in this string
        public Form1()
        {
            InitializeComponent();
        }

        private void Transmit_Click(object sender, EventArgs e)
        {

        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void textBox1_TextChanged(object sender, EventArgs e)
        {
            commands = textBox1.Text; // Fill the string with the textbox's contents
        }
    }
}
```

C# Code

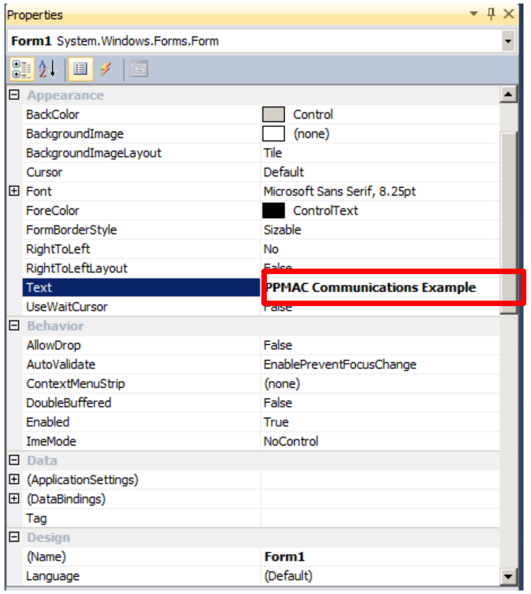
Using GUI Tools

Now let's add another TextBox to contain PMAC's response, a label to label that TextBox, and a RichTextBox to show communication status information. These objects are highlighted on the left. Then, change the form's text from "Form1" to "PMAC Communications Example" in the Form's Properties box (on right). After some superficial rearrangement of items, your GUI should look like the center image (you can make it wider if desired).





PPMAC Communications Example
Enter Text to Transmit to PPMAC:
Transmit
Response from PPMAC:
Communication Status:



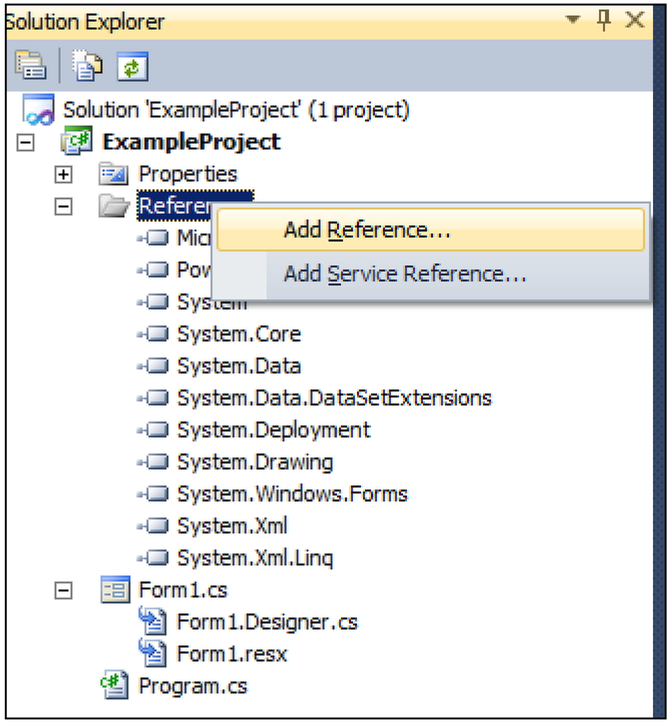
Properties

Form1 System.Windows.Forms.Form

Appearance	
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayout	Tile
Cursor	Default
Font	Microsoft Sans Serif, 8.25pt
ForeColor	ControlText
FormBorderStyle	Sizable
RightToLeft	No
RightToLeftLayout	False
Text	**PPMAC Communications Example**
UseWaitCursor	False
Behavior	
AllowDrop	False
AutoValidate	EnablePreventFocusChange
ContextMenuStrip	(none)
DoubleBuffered	False
Enabled	True
ImeMode	NoControl
Data	
(ApplicationSettings)	
(DataBindings)	
Tag	
Design	
(Name)	**Form1**
Language	(Default)

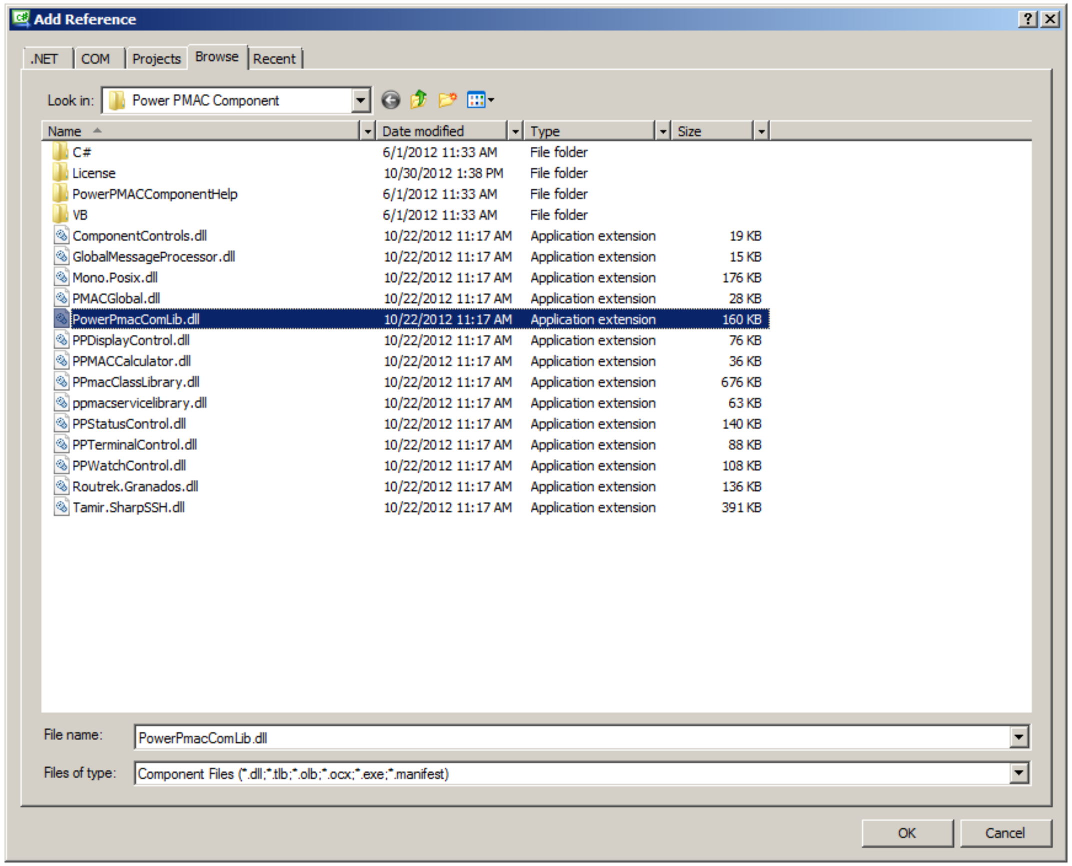
Adding the Reference

To use any of the features of the Power PMAC Component Library, you must add a reference to the DLL file. To do this, go to the Solution Explorer on the right, right-click References, and then “Add Reference”.



Adding the Reference

Now, browse to the location where you installed the Power PMAC Component Library. It is typically in C:\Program Files\Delta Tau Data Systems Inc\Power PMAC Component. Then, pick the file labeled "PowerPmacComLib.dll" and click OK.



Look in: Power PMAC Component

Name	Date modified	Type	Size
C#	6/1/2012 11:33 AM	File folder	
License	10/30/2012 1:38 PM	File folder	
PowerPMACComponentHelp	6/1/2012 11:33 AM	File folder	
VB	6/1/2012 11:33 AM	File folder	
ComponentControls.dll	10/22/2012 11:17 AM	Application extension	19 KB
GlobalMessageProcessor.dll	10/22/2012 11:17 AM	Application extension	15 KB
Mono.Posix.dll	10/22/2012 11:17 AM	Application extension	176 KB
PMACGlobal.dll	10/22/2012 11:17 AM	Application extension	28 KB
PowerPmacComLib.dll	10/22/2012 11:17 AM	Application extension	160 KB
PPDisplayControl.dll	10/22/2012 11:17 AM	Application extension	76 KB
PPMACCalculator.dll	10/22/2012 11:17 AM	Application extension	36 KB
PPmacClassLibrary.dll	10/22/2012 11:17 AM	Application extension	676 KB
ppmacservicelibrary.dll	10/22/2012 11:17 AM	Application extension	63 KB
PPStatusControl.dll	10/22/2012 11:17 AM	Application extension	140 KB
PPTerminalControl.dll	10/22/2012 11:17 AM	Application extension	88 KB
PPWatchControl.dll	10/22/2012 11:17 AM	Application extension	108 KB
Routrek.Granados.dll	10/22/2012 11:17 AM	Application extension	136 KB
Tamir.SharpSSH.dll	10/22/2012 11:17 AM	Application extension	391 KB

File name: PowerPmacComLib.dll

Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

Using GUI Tools

GUI Design Complete – Now What?

Now that the GUI's design is complete and the code framework is in place, we just need to add the standard code snippets for opening communication with Power PMAC and for sending strings to/reading strings from PPMAC. Before moving on, you may want to save (File→Save As…) a separate copy of your GUI project as a design template for a general communications GUI before any additional code is added.

You can use the same code for opening and closing communication regardless of application. However, there are two different methods of actually transmitting and receiving information to and from PMAC:

Synchronous Communication

Synchronous Communication transmits a string to PPMAC and forces the entire program to wait for PMAC to respond within a timeout period specified. This is fine for very simple programs, but is less efficient than Asynchronous Communication because the program sits there doing nothing until it receives a response from PPMAC.

Asynchronous Communication

This is the recommended method of communication. It uses **event**s, which are basically software interrupts. The program transmits a string to PPMAC and then returns to its other tasks. The **event** is triggered when PPMAC sends its response to the host computer, at which time the interrupt service routine executes to process the received data. There is no need to poll or wait for the host. This way, the program can make use of the time between sending the string to PPMAC and waiting for the response, doing other things in the intermediate time.

Synchronous Communication

To use Synchronous Communication, we just need to copy-paste some standard code into the Form we already created. First, make sure you put all of these namespaces above your project's namespace:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using PowerPmacComLib;
using System.Threading;
```
C# Code

These may overlap with that which you are already **using**, so just be careful to delete any duplicates.

Synchronous Communication

Put these variables just under the namespace declaration of your project:

```
// namespace My_Project_Name
// {
 delegate bool ComErrorInvokeDelegate();
 delegate void AppendTextDelegate(String message);
```

C# Code

Put these variables just under the **class** definition of your Form:

```
// public partial class Form1 : Form
//     { Put these variables under where the above code from earlier appears
        ISyncGpasciiCommunicationInterface communication = null;
        deviceProperties currentDeviceProp = new deviceProperties();
        deviceProperties currentDevProp = new deviceProperties();
        ManualResetEvent sync = new ManualResetEvent(false);
        int noOfCommandsSent = 0;
        Char ACK = '\x06';
        String commands = String.Empty; // If you already added this, don't add this again
        String response = String.Empty; // For storing the response from PPMAC
```

C# Code

Synchronous Communication

Now we need to add code to start the dialog box that starts communication. To do this, use this as your Form’s **constructor**:

```
 public Form1(){
            InitializeComponent();
            try {
                if (Globals.isValidLicense){ // Check for a valid PPMAC Comm Lib license
communication = Connect.CreateSyncGpascii(CommunicationGlobals.ConnectionTypes.SSH, null);
                    Globals.isValidLicense = true; }
            }
            catch (Exception ex) {
                AppendTextToOutPut(ex.Message);
                Globals.isValidLicense = false; }
            currentDevProp.IPAddress = Settings1.Default.defaultIPAddress;
            currentDevProp.Password = Settings1.Default.defaultPassword;
            currentDevProp.PortNumber = Convert.ToInt16(Settings1.Default.defaultPort);
            currentDevProp.User = Settings1.Default.defaultUser;
            currentDevProp.Protocol = CommunicationGlobals.ConnectionTypes.SSH;
            if (communication == null) {
                AppendTextToOutPut("Invalid license");
                return; }

            DevicePropertyPage devicePage = new DevicePropertyPage(currentDevProp,
communication.GpAsciiConnected);
            devicePage.OnConnect += new
DevicePropertyPage.OnConnectFunction(devicePage_OnConnect);
            devicePage.OnDisConnect += new
DevicePropertyPage.OnDisConnectFunction(devicePage_OnDisConnect);
            devicePage.ShowDialog();
        }
```

C# Code

Synchronous Communication

Now we need to add code that handles connections, disconnections, and errors. Just add the following methods to your Form's class:

```
bool devicePage_OnDisConnect()
        {
            if (communication == null)
            {
                AppendTextToOutPut("Invalid license");
                return false;
            }

            bool bSuccess = false;
            if (communication.GpAsciiConnected)
            {
                bSuccess = communication.DisconnectGpascii();
                label1.Text = "Not Connected";
            }

            return bSuccess;
        }
```

C# Code

Synchronous Communication

Continued from previous slide

```
 bool devicePage_OnConnect(deviceProperties DevProp)
        {
            if (communication == null)
            {
                AppendTextToOutPut("Invalid license");
                return false;
            }

            communication = Connect.CreateSyncGpascii(DevProp.Protocol, communication);
            bool bSuccess = communication. ConnectGpAscii(DevProp.IPAddress, DevProp.PortNumber,
DevProp.User, DevProp.Password);
            if (bSuccess)
            {
                currentDevProp.IPAddress = DevProp.IPAddress;
                currentDevProp.Password = DevProp.Password;
                currentDevProp.PortNumber = DevProp.PortNumber;
                currentDevProp.User = DevProp.User;
                AppendTextToOutPut("Connected to the device ' " + DevProp.IPAddress + " '\n");
                label1.Text = "Connected to " + DevProp.IPAddress;
                communication.ComERROR += new SocketErMessages(communication_ComERROR);
                Settings1.Default.defaultIPAddress = DevProp.IPAddress;
                Settings1.Default.defaultPassword = DevProp.Password;
                Settings1.Default.defaultPort = DevProp.PortNumber.ToString();
                Settings1.Default.defaultUser = DevProp.User;
                Settings1.Default.Save();
            }
            return bSuccess;
        }
```

C# Code

Synchronous Communication

Continued from previous slide:

```
 void communication_ComERROR(object sender, ComErArgs e)
        {
            Invoke(new ComErrorInvokeDelegate(devicePage_OnDisConnect));
            Invoke(new AppendTextDelegate(AppendTextToOutPut), "Device has been disconnected due
to a communication error.");
        }
```

C# Code

Now add a method that appends text to our RichTextBox for displaying communications status:

```
 void AppendTextToOutPut(String message)
        {
            richTextBox1.AppendText(DateTime.Now.ToLongTimeString() + " : \n");
            richTextBox1.AppendText(message + "\n");
            richTextBox1.AppendText("-------------------------------------------------------------
-------------------------------------------------------\n");
            richTextBox1.ScrollToCaret();
        }
```

C# Code

Synchronous Communication

Now, we need to fill in the method that runs when the “Transmit” button is clicked. Use this as your Transmit_Click() method:

```
private void Transmit_Click(object sender, EventArgs e)
        {
            if (communication == null)
            {
                AppendTextToOutPut("Invalid license");
                return;
            }

            if (communication.GpAsciiConnected)
            {
                // GetResponse is the actual method that sends and receives the string
                Status communicationStatus = communication.GetResponse(command, out response);
                if (communicationStatus == Status.Ok)
                {
                   textBox2.Text=response; // Populate textBox2 with the response
                }
            }
            else // Send an error to the communication status RichTextBox
                richTextBox1.AppendText("Please connect to a device first !\n");
        }
```

C# Code

Synchronous Communication

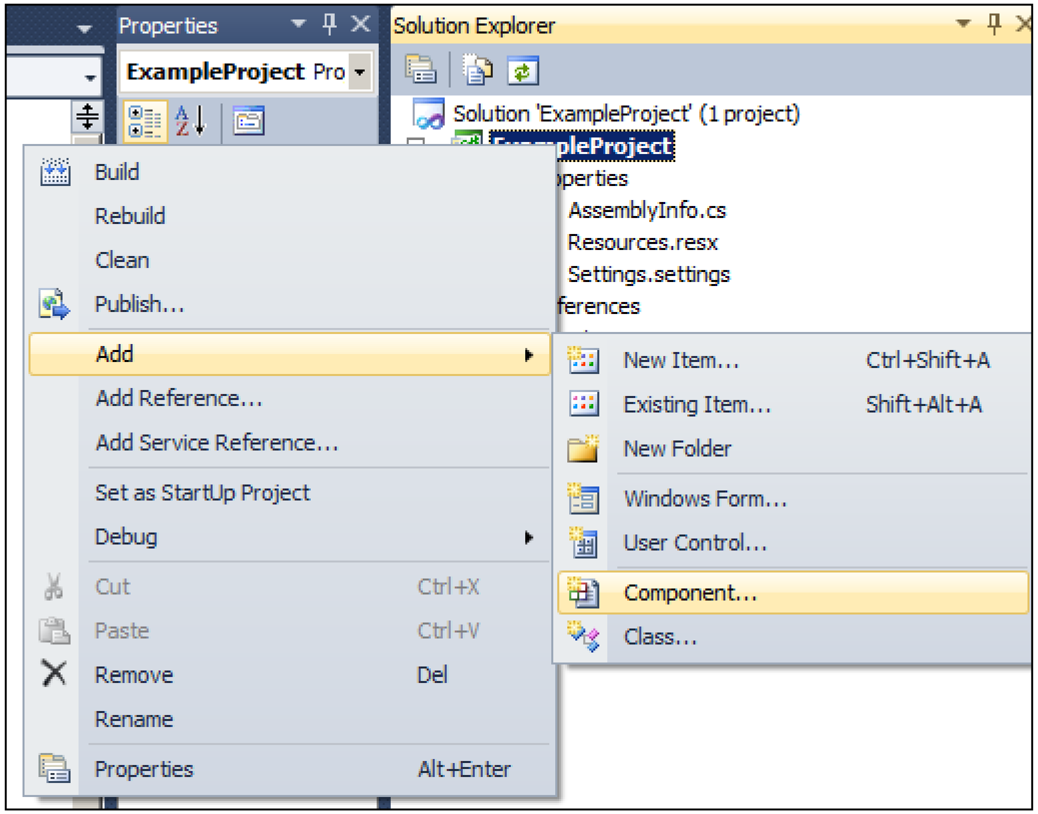
Next, we need to create the global licensing variable. Just append this as a separate class in your **namespace**:

```
public class Globals
   {
       public static bool isValidLicense = true;
   }
```

C# Code

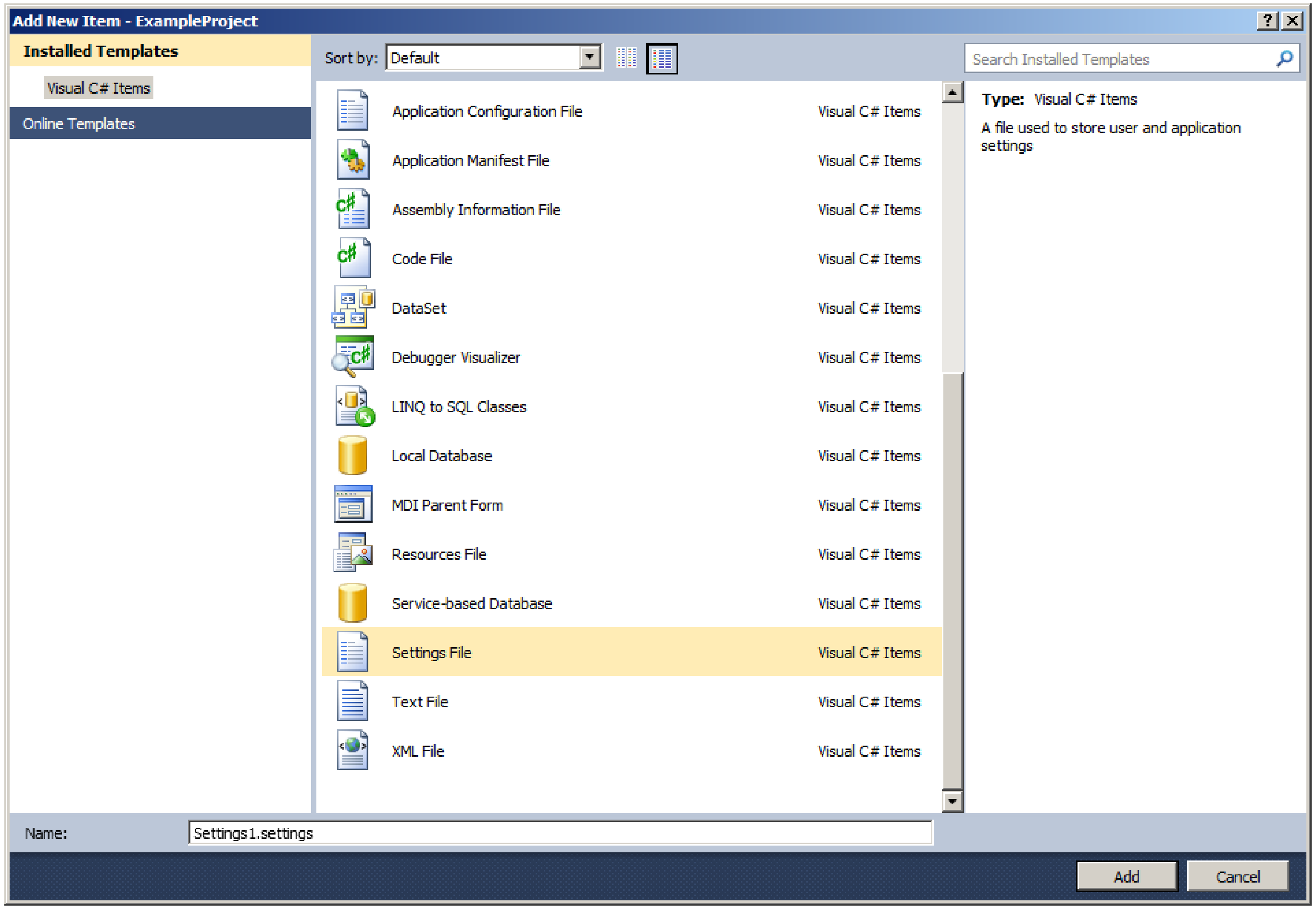
Synchronous Communication

Lastly, we need to add a settings file to retain the communications settings for talking to PPMAC. To add a settings file, in the Solution Explorer, right-click the project name, then Add→Component…



Synchronous Communication

Scroll down and select "Settings File" and name is "Settings1.settings" and then click "Add":


Add New Item - ExampleProject

Installed Templates

Visual C# Items

Online Templates

Sort by: Default

Search Installed Templates

Application Configuration File	Visual C# Items
Application Manifest File	Visual C# Items
Assembly Information File	Visual C# Items
Code File	Visual C# Items
DataSet	Visual C# Items
Debugger Visualizer	Visual C# Items
LINQ to SQL Classes	Visual C# Items
Local Database	Visual C# Items
MDI Parent Form	Visual C# Items
Resources File	Visual C# Items
Service-based Database	Visual C# Items
Settings File	Visual C# Items
Text File	Visual C# Items
XML File	Visual C# Items

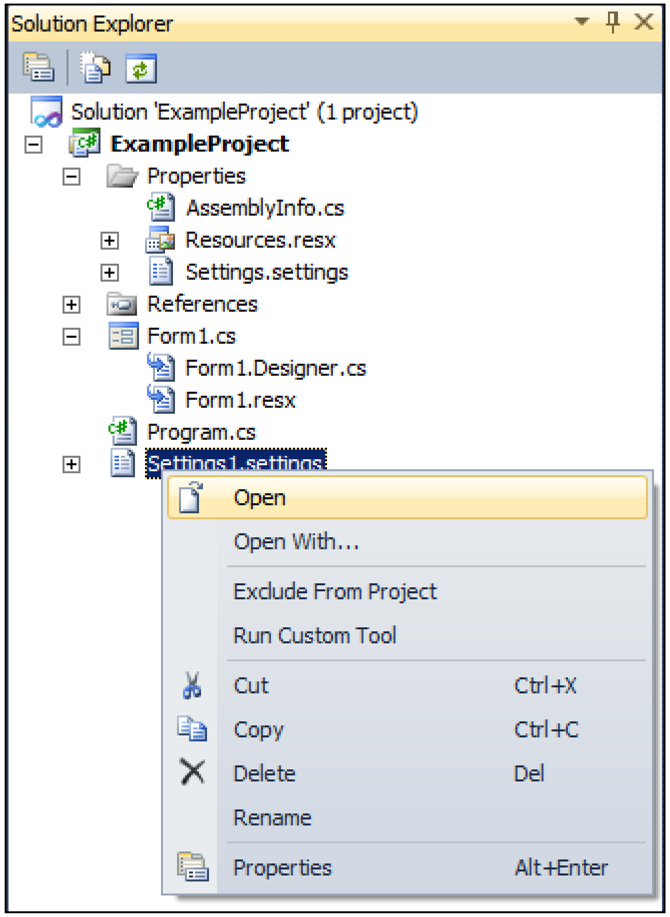
Type: Visual C# Items

A file used to store user and application settings

Name: Settings1.settings

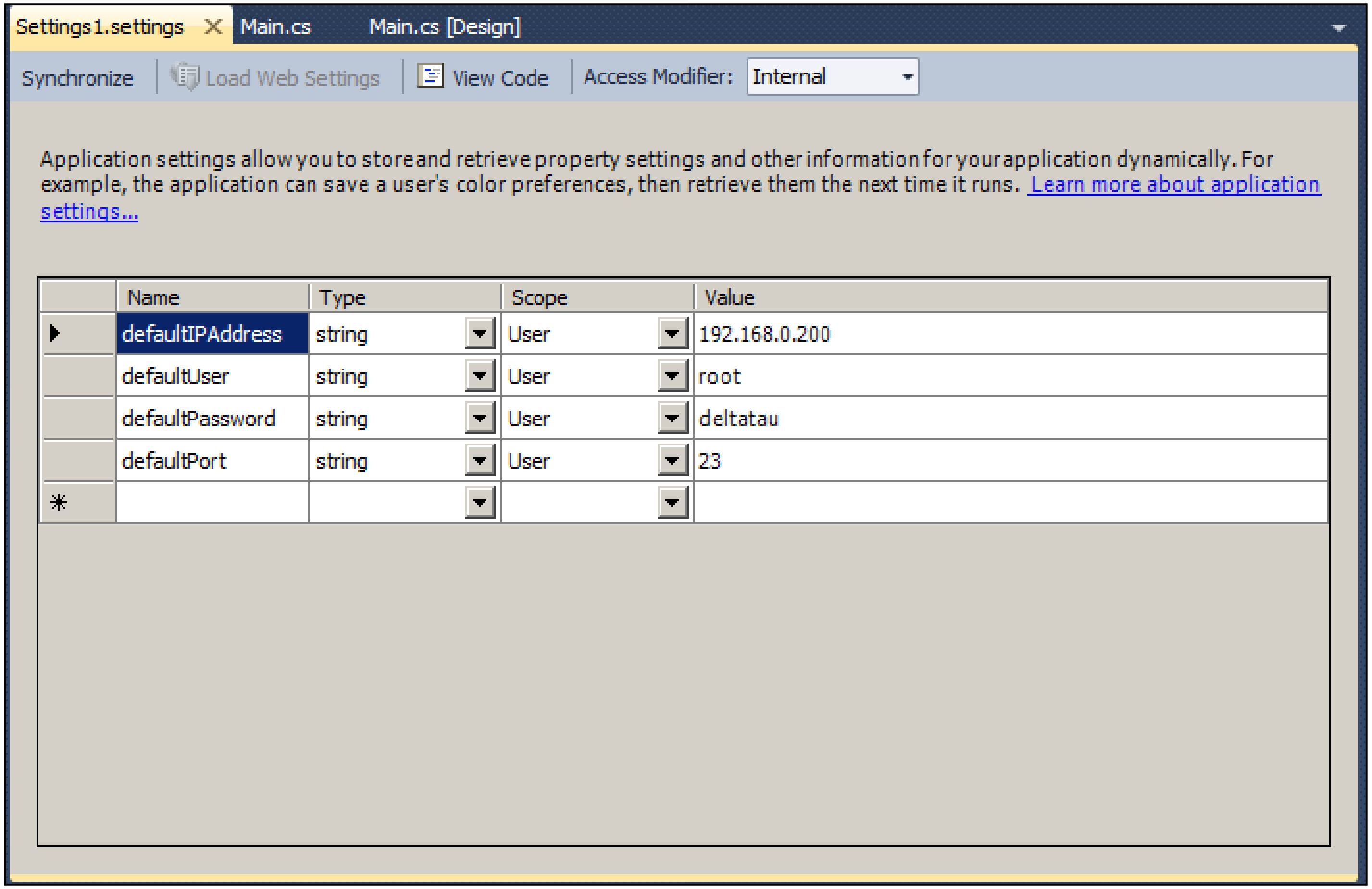
Synchronous Communication

Then, right-click the settings file in the Solution Explorer and click “Open”:



Synchronous Communication

Fill it out the screen that opens such that it looks just like the screen below:



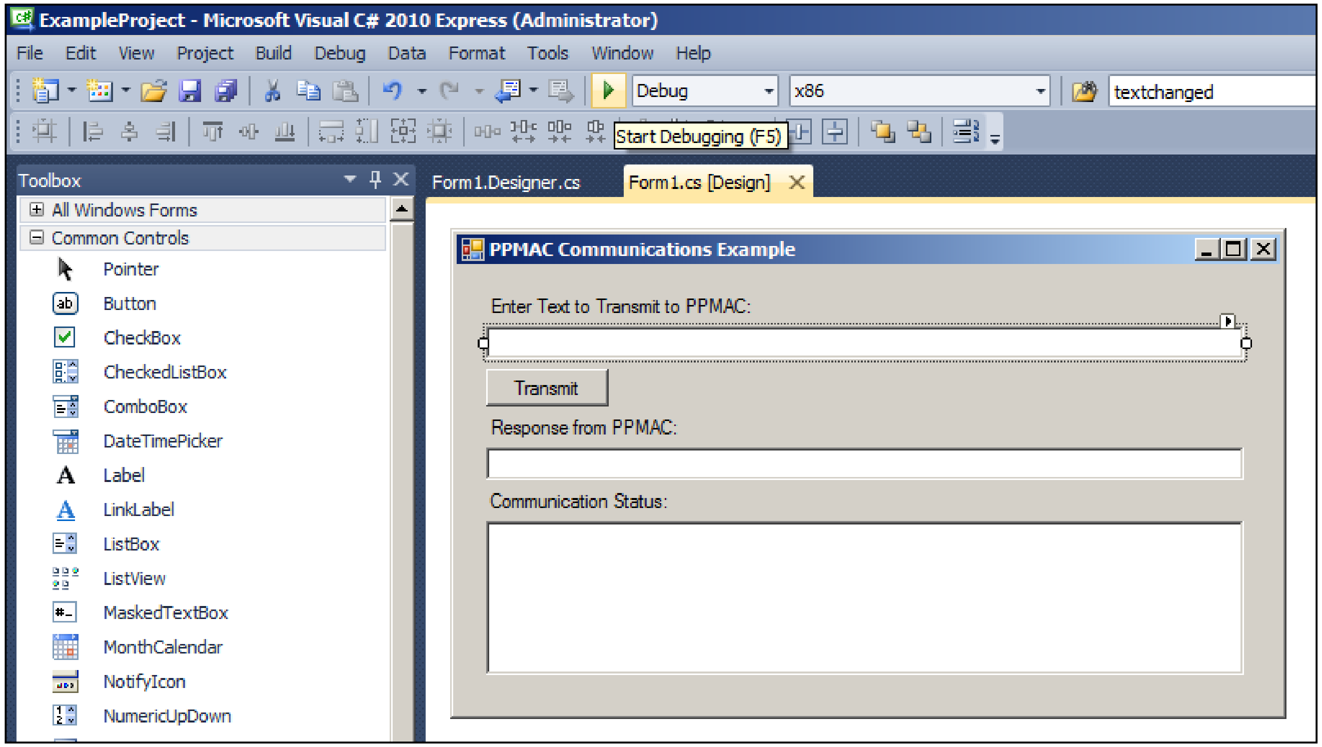
Access Modifier: Internal

Application settings allow you to store and retrieve property settings and other information for your application dynamically. For example, the application can save a user's color preferences, then retrieve them the next time it runs. Learn more about application settings...

	Name	Type	Scope	Value
▶	defaultIPAddress	string	User	192.168.0.200
	defaultUser	string	User	root
	defaultPassword	string	User	deltatau
	defaultPort	string	User	23
*				

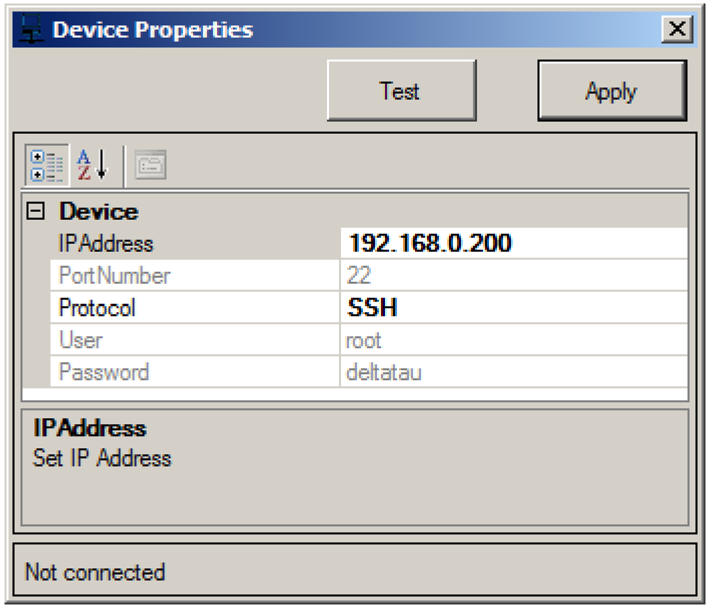
Synchronous Communication

Now, your GUI should be ready to go. Save your work and then click the Start Debugging button to run the program:



Synchronous Communication

Once the program starts, the Communications Dialog Box pops up (shown below). Enter your communications settings, then click “Apply:



Device

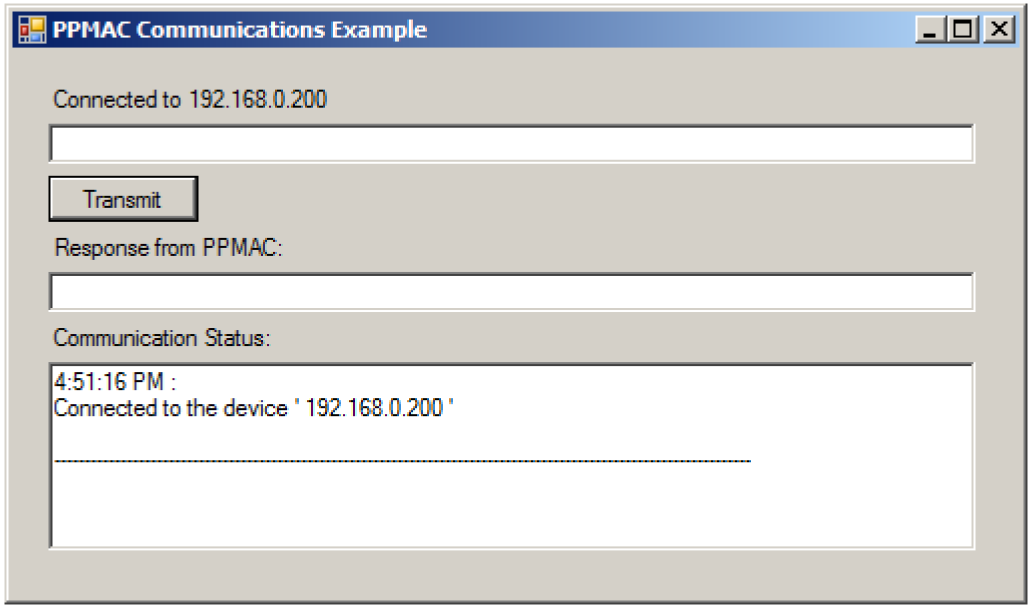
IPAddress	192.168.0.200
Port Number	22
Protocol	SSH
User	root
Password	deltatau

IPAddress
Set IP Address

Not connected

Synchronous Communication

Now your program is running! Just type your text into the first box, click “Transmit,” and then look at the response in the second box!



```
4:51:16 PM :
Connected to the device ' 192.168.0.200 '

--------------------------------------------------------------------------------------------------------------------------
```

Asynchronous Communication

For the Asynchronous Communication example, we can use the exact same GUI we designed for the Synchronous Communication example. Not much changes except now we have to instantiate a different type of class for the communication object, and we have to create a separate function to run once the host receives data back from PMAC. We will list the example in full, however, in order that these examples may be read separately.

Asynchronous Communication

To use Asynchronous Communication, we just need to copy-paste some standard code into the Form we already created. First, make sure you put all of these namespaces above your project's namespace:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using PowerPmacComLib;
using System.Threading;
```

C# Code

These may overlap with that which you are already **using**, so just be careful to delete any duplicates.

Asynchronous Communication

Put these variables just under the namespace declaration of your project:

```
// namespace My_Project_Name
// {
 delegate bool ComErrorInvokeDelegate();
 delegate void AppendTextDelegate(String message);
```

C# Code

Put these variables just under the **class** definition of your Form:

```
// public partial class Form1 : Form
//    {

// Declare and initialize variables global with this form
        deviceProperties currentDeviceProp = new deviceProperties();
        deviceProperties currentDevProp = new deviceProperties();
        IAsyncGpasciiCommunicationInterface communication = null;
        ManualResetEvent sync = new ManualResetEvent(false);
        int noOfCommandsSent = 0;
        String dataReceived = String.Empty;
        Char ACK = '\x06';
        String commands = String.Empty;
```

C# Code

Asynchronous Communication

Now we need to add code to start the dialog box that starts communication. To do this, use this as your Form's **constructor**:

```
 public Form1()
        {
            InitializeComponent();

            // Start the Asynchronous Communication
            try
            {
                if (Globals.isValidLicense)
                {
                    communication = 
Connect.CreateAsyncGpascii(CommunicationGlobals.ConnectionTypes.SSH, null);
                    Globals.isValidLicense = true;
                }
            }
            catch (Exception ex)
            {
                AppendTextToOutPut(ex.Message); // Print an error message if we can't open 
communication
                Globals.isValidLicense = false;
            }

            // Set default connection settings
            currentDevProp.IPAddress = Settings1.Default.defaultIPAddress;
            currentDevProp.Password = Settings1.Default.defaultPassword;
            currentDevProp.PortNumber = Convert.ToInt16(Settings1.Default.defaultPort);
            currentDevProp.User = Settings1.Default.defaultUser;
            currentDevProp.Protocol = CommunicationGlobals.ConnectionTypes.SSH;
```

C# Code

Asynchronous Communication

Continued from previous slide:

```
 if (communication == null)
            {
                AppendTextToOutPut("Invalid license");
                return;
            }

            // Initialize communication dialog
            DevicePropertyPage devicePage = new DevicePropertyPage(currentDevProp, 
communication.GpAsciiConnected);
            devicePage.OnConnect += new 
DevicePropertyPage.OnConnectFunction(devicePage_OnConnect);
            devicePage.OnDisConnect += new 
DevicePropertyPage.OnDisConnectFunction(devicePage_OnDisConnect);
            devicePage.ShowDialog(); // Start the communication dialog
        }
```

C# Code

Asynchronous Communication

Now we need to add code that handles connections, disconnections, and errors. Just add the following methods to your Form's class:

```
 bool devicePage_OnDisConnect() // This runs when we disconnect from the device
        {
            if (communication == null)
            {
                AppendTextToOutPut("Invalid license");
                return false;
            }

            bool bSuccess = false;
            if (communication.GpAsciiConnected)
            {
                bSuccess = communication.DisconnectGpascii();
                label1.Text = "Not Connected";
            }
            communication.AsyncDataAvailable -= new
AsyncDataReceiveEvent(communication_AsyncDataAvailable);
            return bSuccess;
        }
```

C# Code

Asynchronous Communication

Continued from previous slide:

```
 bool devicePage_OnConnect(deviceProperties DevProp) // This runs when we connect to the device
{
            if (communication == null)
            {
                AppendTextToOutPut("Invalid license");
                return false;
            }

            communication = Connect.CreateAsyncGpascii(DevProp.Protocol, communication); //
Instantiate the Async communication object
            bool bSuccess = communication.GPAsciiConnect(DevProp.IPAddress, DevProp.PortNumber,
DevProp.User, DevProp.Password); // Connect to the device
            if (bSuccess) // If successful
            {
                // Set the communication parameters with what the user entered into the
communication dialog
                currentDevProp.IPAddress = DevProp.IPAddress;
                currentDevProp.Password = DevProp.Password;
                currentDevProp.PortNumber = DevProp.PortNumber;
                currentDevProp.User = DevProp.User;
                richTextBox1.AppendText("Connected to the device ' " + DevProp.IPAddress + "
'\n");
                label1.Text = "Connected to " + DevProp.IPAddress;
                communication.AsyncDataAvailable += new
AsyncDataReceiveEvent(communication_AsyncDataAvailable);  // Create a thread for receiving the
data
                communication.ComERROR += new SocketErMessages(communication_ComERROR);
            }
            return bSuccess;        }
```

C# Code

Asynchronous Communication

Continued from previous slide:

```
 void communication_ComERROR(object sender, ComErArgs e)
        {
            Invoke(new ComErrorInvokeDelegate(devicePage_OnDisConnect));
            Invoke(new AppendTextDelegate(AppendTextToOutPut), "Device has been disconnected due
to a communication error.");
        }
```

C# Code

Now let's add a method that appends text to our RichTextBox for displaying communications status:

```
 void AppendTextToOutPut(String message) // Print the text into the output box
        {
            richTextBox1.AppendText(DateTime.Now.ToLongTimeString() + " : \n");
            richTextBox1.AppendText(message + "\n");
            richTextBox1.AppendText("-------------------------------------------------------------
-------------------------------------------------------\n");
            richTextBox1.ScrollToCaret();
        }
```

C# Code

Asynchronous Communication

Now we need to add the function that runs when the **event** (the host receiving PPMAC's response) occurs. This is C#'s version of an **interrupt service routine**:

```
 void communication_AsyncDataAvailable(object sender, AsyncDataArgs e) // This function runs when
the thread receives data from PPMAC
        {
            try
            {
                dataReceived += e.Response; // Store the data in dataReceived
                int noOfCommandsReceived = 0;
                for (int i = 0; i < dataReceived.Length; i++)
                {
                    if (dataReceived[i] == ACK)
                        noOfCommandsReceived++;
                }

                if (noOfCommandsReceived > noOfCommandsSent)
                {
                    sync.Set();
                }

            }
            catch (Exception ex){}
        }
```

C# Code

Asynchronous Communication

Next, we need to create the global licensing variable. Just append this as a separate class in your **namespace**:

```
public class Globals
   {
       public static bool isValidLicense = true;
   }
```

C# Code

Asynchronous Communication

Now, we need to fill in the method that runs when the “Transmit” button is clicked. Use this as your Transmit_Click() method:

```
 private void Transmit_Click(object sender, EventArgs e)
        {
if (communication == null)
            {
                AppendTextToOutPut("Invalid license");
                return;
            }

            if (communication.GpAsciiConnected) // If we are connected
            {
                String OutputString = String.Empty;
                sync.Reset();
                noOfCommandsSent = 0;
                dataReceived = String.Empty;

             // Send the commands stored in "commands"
                Status communicationStatus = communication.AsyncGetResponse(commands);
                if (communicationStatus == Status.Ok)
                {
                    if (sync.WaitOne(10000, false)) // Use a 10000 ms timeout period
                    {
                          // For each string received, parse and print to output box
                        for (int i = 0; i < dataReceived.Length; i++)
                        {
```

C# Code

Asynchronous Communication

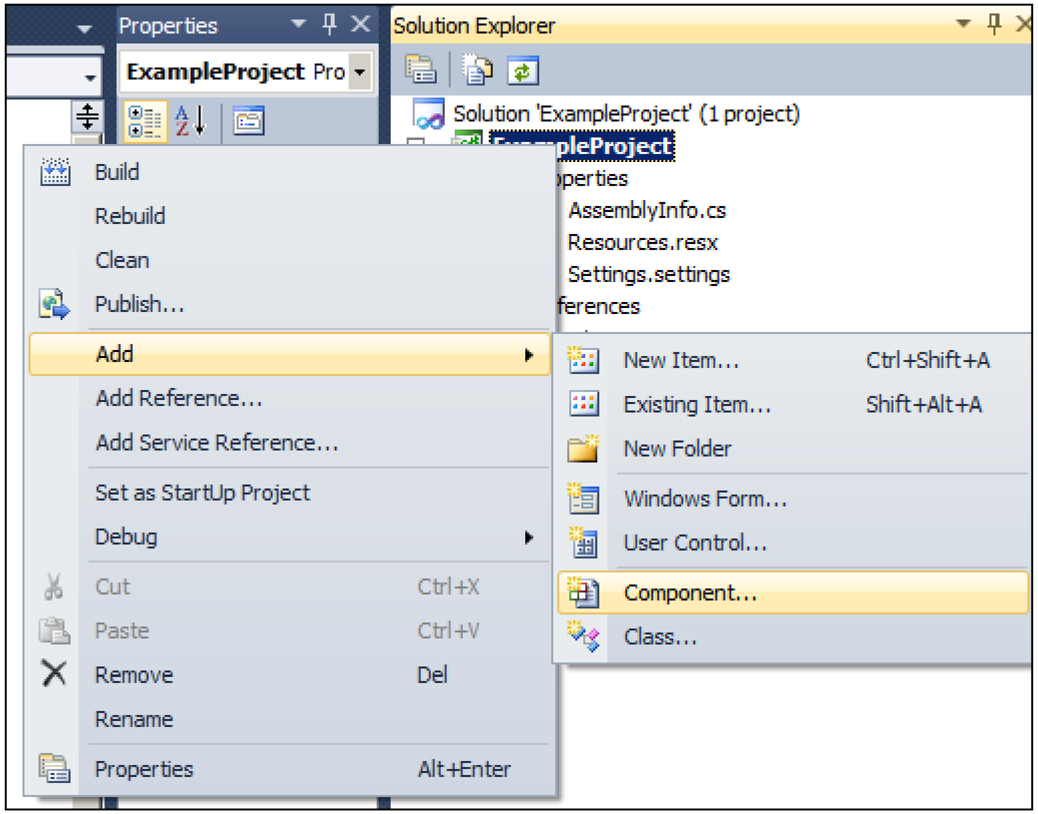
Continued from previous slide:

```
if (dataReceived[i] != ACK) // Get rid of the ACK character from what we print on screen
                                OutputString += dataReceived[i];
                            else
                                break;
                        }
                        textBox2.Text = OutputString; // Put the received text into textBox2
                        richTextBox1.AppendText("Received all data from the device!\n");
                    }
                    else
                    {
                        richTextBox1.AppendText("Cannot receive data from the device. Please check 
communication.");

                    }
                }
            }
            else
                richTextBox1.AppendText("Please connect to a device first.\n");        }
```

C# Code

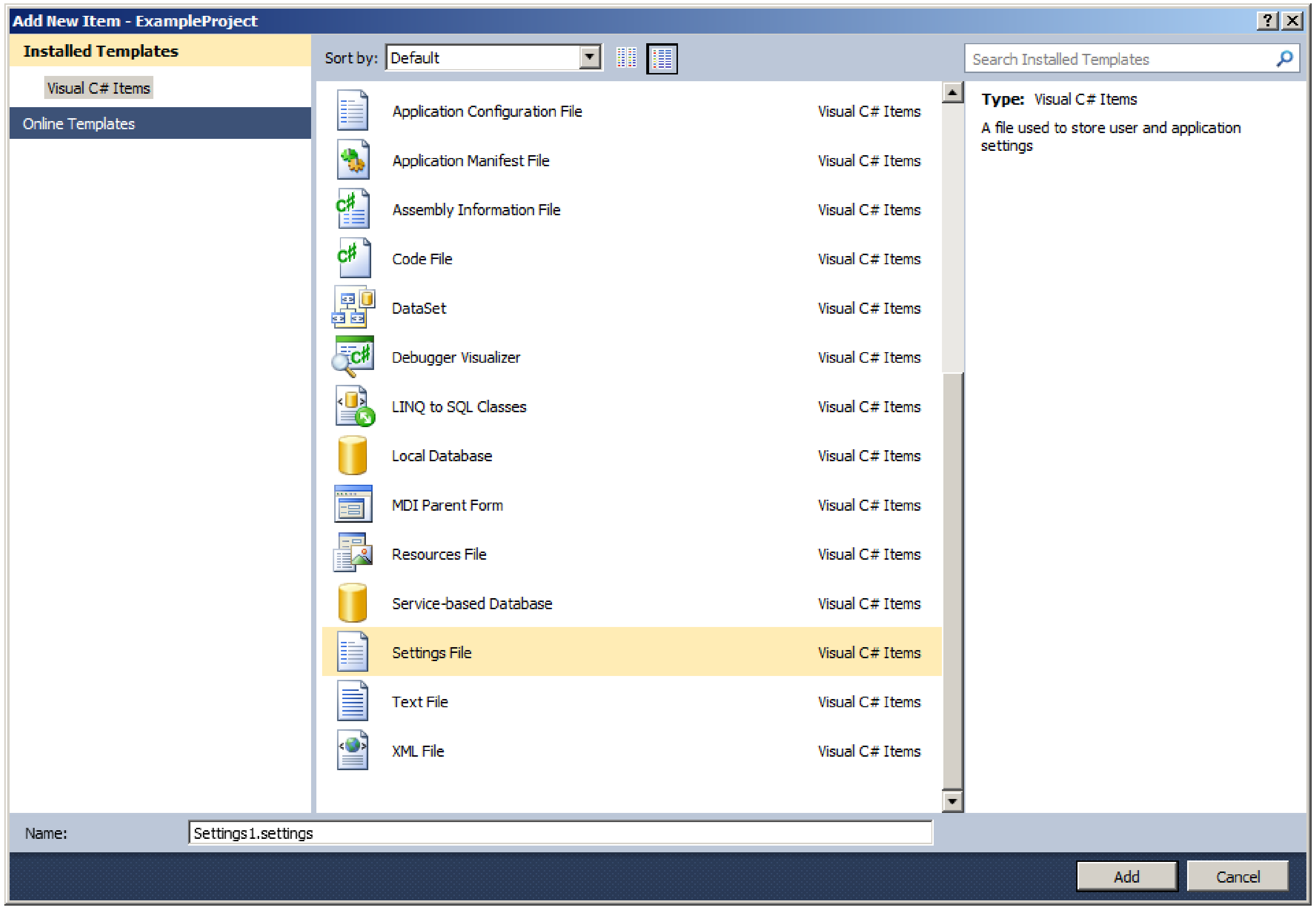
Asynchronous Communication

Lastly, we need to add a settings file to retain the communications settings for talking to PPMAC. To add a settings file, in the Solution Explorer, right-click the project name, then Add→Component…



Asynchronous Communication

Scroll down and select “Settings File” and name is “Settings1.settings” and then click “Add”:



Add New Item - ExampleProject

Installed Templates

Visual C# Items

Online Templates

Sort by: Default

Search Installed Templates

Application Configuration File	Visual C# Items
Application Manifest File	Visual C# Items
Assembly Information File	Visual C# Items
Code File	Visual C# Items
DataSet	Visual C# Items
Debugger Visualizer	Visual C# Items
LINQ to SQL Classes	Visual C# Items
Local Database	Visual C# Items
MDI Parent Form	Visual C# Items
Resources File	Visual C# Items
Service-based Database	Visual C# Items
Settings File	Visual C# Items
Text File	Visual C# Items
XML File	Visual C# Items

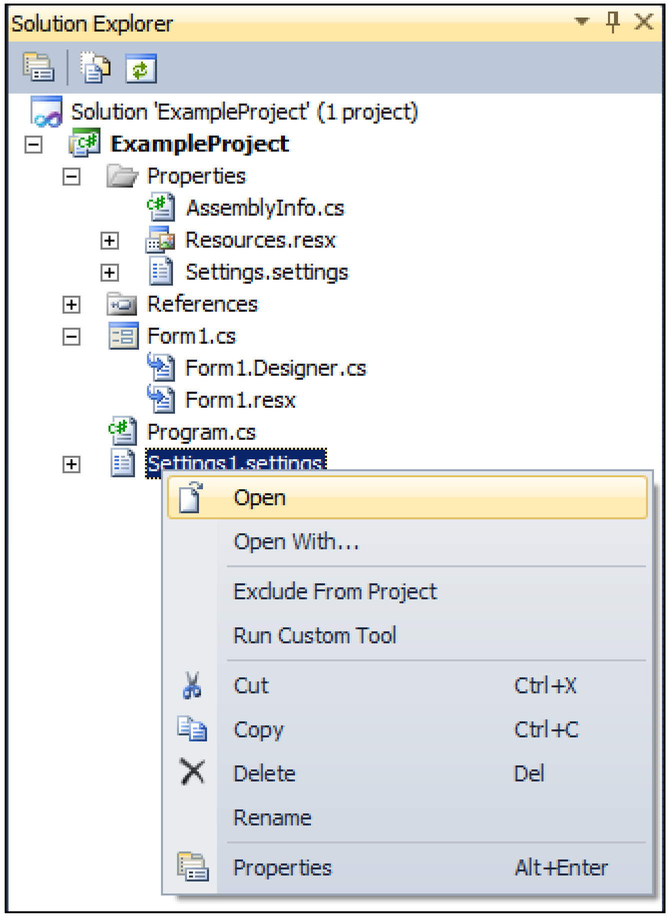
Type: Visual C# Items

A file used to store user and application settings

Name: Settings1.settings

Asynchronous Communication

Then, right-click the settings file in the Solution Explorer and click "Open":

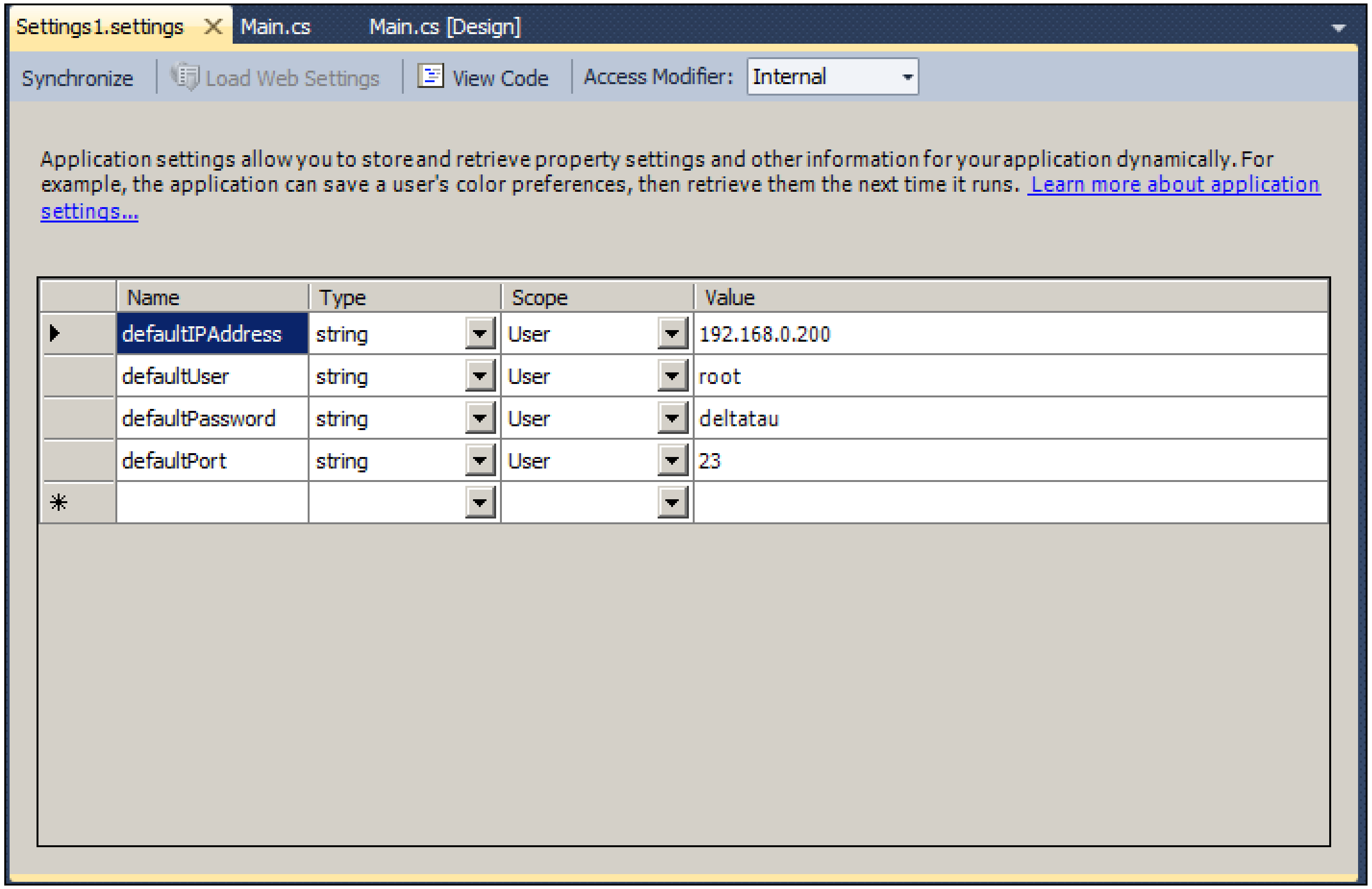


Solution Explorer

Solution 'ExampleProject' (1 project)
- ExampleProject
  - Properties
    - AssemblyInfo.cs
    - Resources.resx
    - Settings.settings
  - References
  - Form1.cs
    - Form1.Designer.cs
    - Form1.resx
  - Program.cs
  - Settings1.settings

Asynchronous Communication

Fill it out the screen that opens such that it looks just like the screen below:

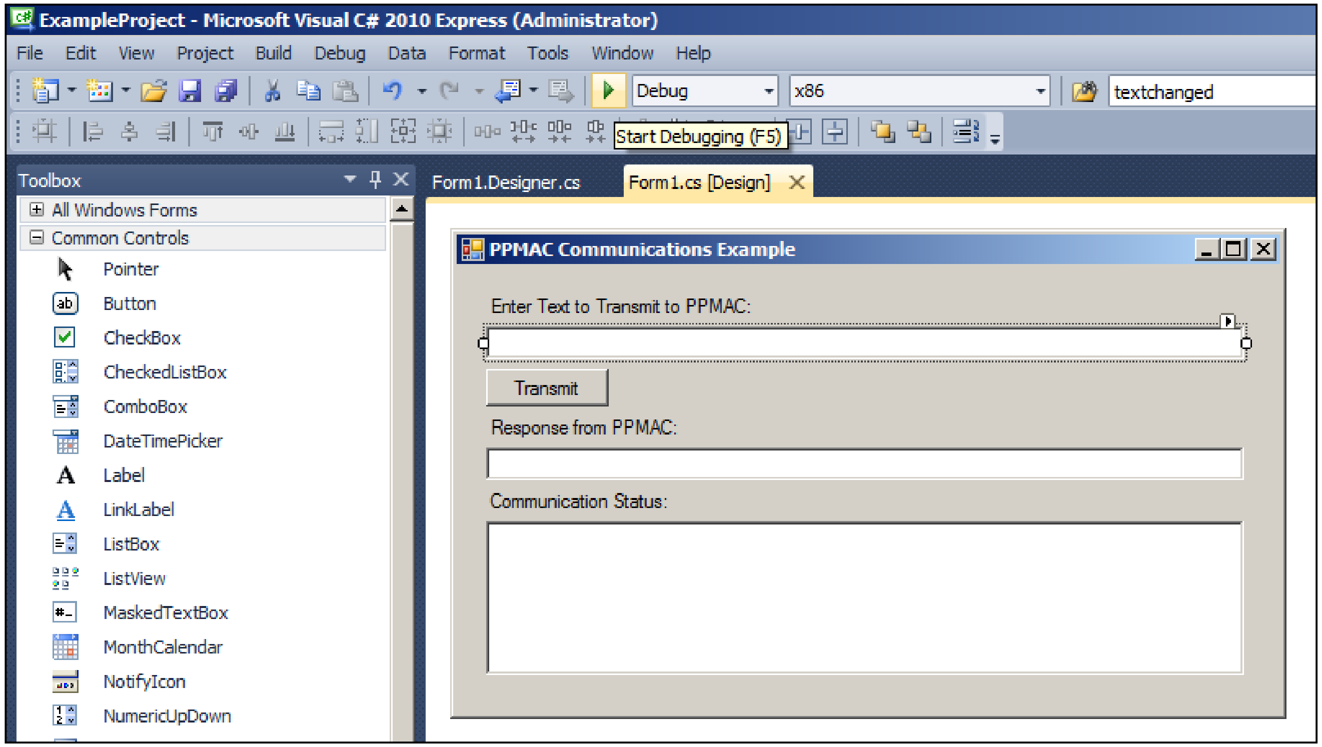


Application settings allow you to store and retrieve property settings and other information for your application dynamically. For example, the application can save a user's color preferences, then retrieve them the next time it runs. Learn more about application settings...

	Name	Type	Scope	Value
▶	defaultIPAddress	string	User	192.168.0.200
	defaultUser	string	User	root
	defaultPassword	string	User	deltatau
	defaultPort	string	User	23
*				

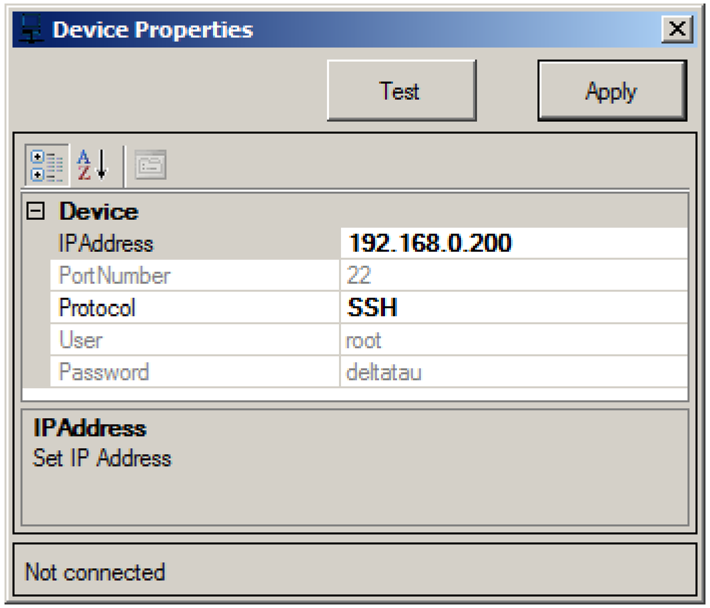
Asynchronous Communication

Now, your GUI should be ready to go. Save your work and then click the Start Debugging button to run the program:



Asynchronous Communication

Once the program starts, the Communications Dialog Box pops up (shown below). Enter your communications settings, then click “Apply:



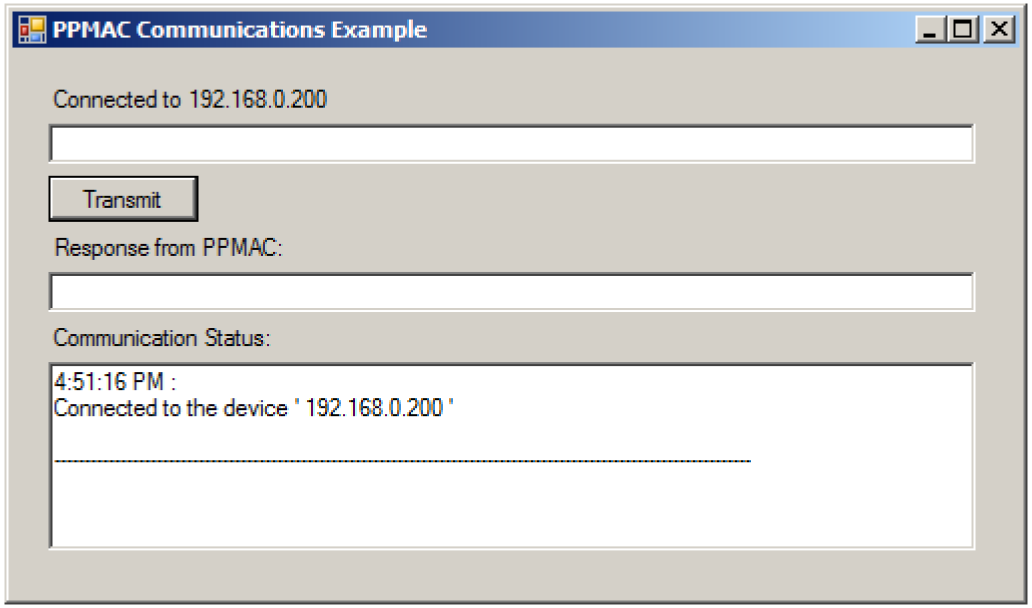
Device	
IPAddress	192.168.0.200
Port Number	22
Protocol	SSH
User	root
Password	deltatau

IPAddress
Set IP Address

Not connected

Asynchronous Communication

Now your program is running! Just type your text into the first box, click “Transmit,” and then look at the response in the second box!



Connected to 192.168.0.200

Response from PPMAC:

Communication Status:

```
4:51:16 PM :
Connected to the device ' 192.168.0.200 '

--------------------------------------------------------------------------------------------------------
```

Introduction to MACRO

MACRO Overview

- "Motion and Control Ring Optical"
- Developed by Delta Tau
- High bandwidth, non-proprietary fiber optic or wired field bus protocol
- Used for machine control networks
- Based upon 100BASEFX (FDDI) and 100BASETX (Ethernet) hardware technologies.
- 125 Mbit/s transfer rate
- Permits phase clock frequencies even beyond 40 kHz
- Noise-immune when using fiber optic cables that can be up to ~2.2 km long

Note

Visit www.macro.org for a thorough description of this protocol.

Nodes and Addressing Structure

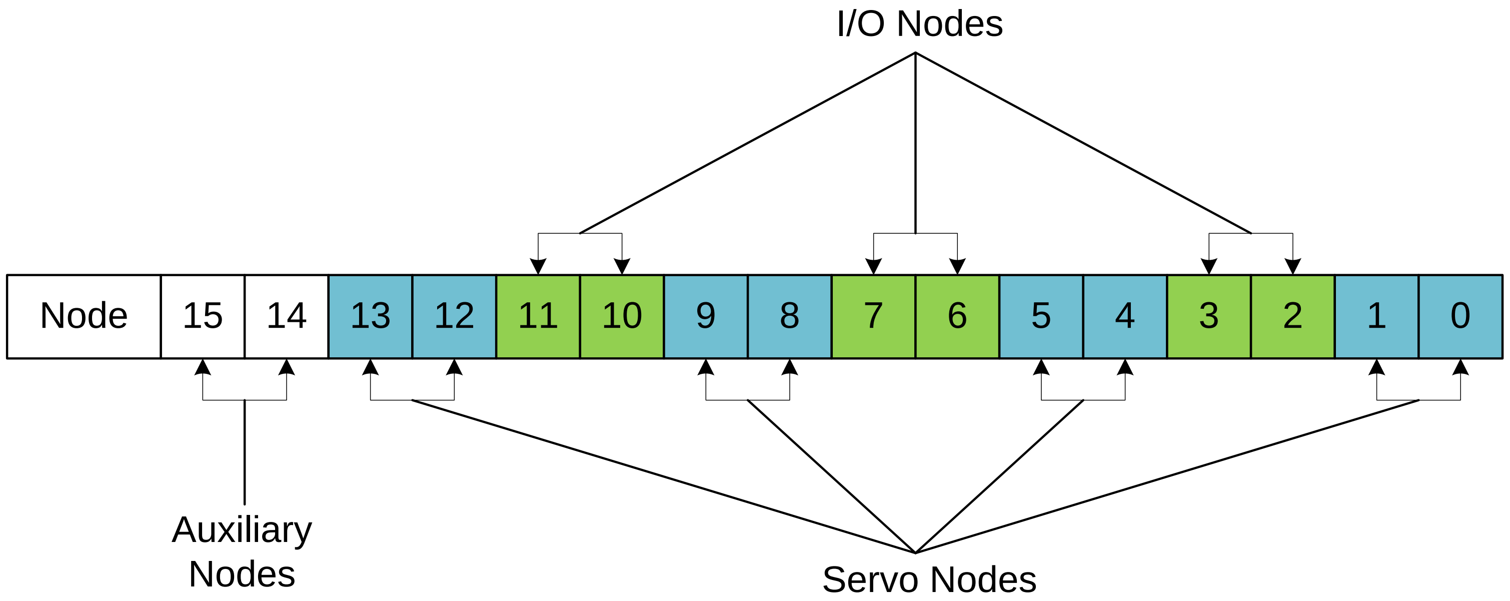
- **Gate3-based MACRO hardware has two sets of MACDRO banks: “A” and “B”**
- **Each bank consists of 16 nodes:**

  2 Auxiliary (for Communication and internal firmware use)
  8 Servo Nodes (for feedback data, flags and output commands)
  6 I/O Nodes (for digital and analog I/O data transfer, or other miscellaneous data)

- **Each motor requires one servo node, so one Gate3 chip can control up to 16 motors**
- **The number of I/O nodes used depends on what I/O devices Gate3 is controlling**

Nodes and Addressing Structure

➢ **The nodes' individual functionality is depicted in the following diagram:**



Each node consists of 8 registers: four 32-bit "Input" registers, which can be accessed by the structure **Gate3[*i*].MacroInA[*j*][*k*]** for bank A and **Gate3[*i*].MacroInB[*j*][*k*]** for bank B, and four 32-bit "Output" registers, which can be accessed by the Power PMAC structures **Gate3[*i*].MacroOutA[*j*][*k*]** for bank A and **Gate3[*i*].MacroOutB[*j*][*k*]** for bank B.

Nodes and Addressing Structure

- **When controlling non-Gate3 MACRO Stations, ACC-5E3 will have its servo node information split up differently within each node j depending on the commutation method being used.**
- **The three modes involved are as follows:**

Analog Output Mode
Motor[*x*].PhaseCtrl = 0
Motor[*x*].pAdc = 0

UV Commutation Mode (a.k.a. Sinusoidal Commutation Mode)
Motor[*x*].PhaseCtrl > 0
Motor[*x*].pAdc = 0

Direct PWM Mode
Motor[*x*].PhaseCtrl > 0
Motor[*x*].pAdc > 0 (= **Gate3[*i*].MacroInA[*j*][1]** for MACRO motors)

Note

I/O Nodes can be arranged in any way desired, regardless of motor control method, and will be described in detail in the MACRO I/O Setup section of this training.

Nodes and Addressing Structure

- **The contents of each servo node are arranged in each MACRO bank according to the control mode used (described on the previous slide) as follows:**

MACRO Bank A		
Node Structure	Bit 31	Bit 0
Gate3[i].MacroInA[j][0]	24 bits of feedback information	8 bits of 0
Gate3[i].MacroInA[j][1]	Not Used in Analog Output Mode/ Not Used in UV Commutation Mode/ 16 bits of current sensor ADCA in Direct PWM Mode	16 bits of 0
Gate3[i].MacroInA[j][2]	Not Used in Analog Output Mode/ Not Used in UV Commutation Mode/ 16 bits of current sensor ADCB in Direct PWM Mode	16 bits of 0
Gate3[i].MacroInA[j][3]	16 bits of channel status/flag information	16 bits of 0

Gate3[i].MacroOutA[j][0]	24 bits of servo output command in Analog Output Mode/ 24 bits of DACA output in UV Commutation Mode/ 24 bits of PWMA command in Direct PWM Mode	8 bits of 0
Gate3[i].MacroOutA[j][1]	Not Used in Analog Output Mode/ 16 bits of DACB command in UV Commutation Mode/ 16 bits of PWMB command in Direct PWM Mode	16 bits of 0
Gate3[i].MacroOutA[j][2]	Not Used in Analog Output Mode/ Not Used in UV Commutation Mode/ 16 bits of PWMC command in Direct PWM Mode	16 bits of 0
Gate3[i].MacroOutA[j][3]	16 bits of channel control commands/flag commands	16 bits of 0

Nodes and Addressing Structure

- **MACRO Bank B's contents are arranged identically those of A:**

MACRO Bank B		
Node Structure	Bit 31	Bit 0
Gate3[i].MacroInB[j][0]	24 bits of feedback information	8 bits of 0
Gate3[i].MacroInB[j][1]	Not Used in Analog Output Mode/ Not Used in UV Commutation Mode/ 16 bits of current sensor ADCA in Direct PWM Mode	16 bits of 0
Gate3[i].MacroInB[j][2]	Not Used in Analog Output Mode/ Not Used in UV Commutation Mode/ 16 bits of current sensor ADCB in Direct PWM Mode	16 bits of 0
Gate3[i].MacroInB[j][3]	16 bits of channel status/flag information	16 bits of 0

Gate3[i].MacroOutB[j][0]	24 bits of servo output command in Analog Output Mode/ 24 bits of DACA output in UV Commutation Mode/ 24 bits of PWMA command in Direct PWM Mode	8 bits of 0
Gate3[i].MacroOutB[j][1]	Not Used in Analog Output Mode/ 16 bits of DACB command in UV Commutation Mode/ 16 bits of PWMB command in Direct PWM Mode	16 bits of 0
Gate3[i].MacroOutB[j][2]	Not Used in Analog Output Mode/ Not Used in UV Commutation Mode/ 16 bits of PWMC command in Direct PWM Mode	16 bits of 0
Gate3[i].MacroOutB[j][3]	16 bits of channel control commands/flag commands	16 bits of 0

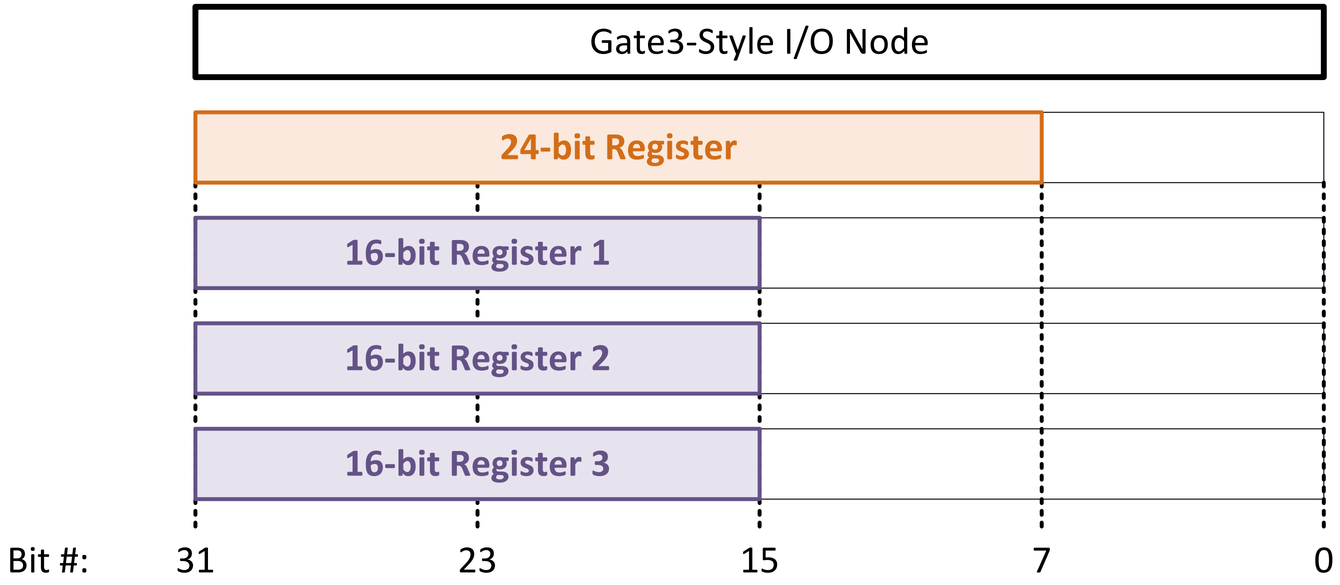
Bit Layouts

- **MACRO Status Flag Registers (Gate3[*i*].MacroInA[*j*][3]/Gate3[*i*].MacroInB[*j*][3] and Gate2[*i*].Macro[*j*][3])**
  Bit 0 – 18 (Reserved for future use)
  19 Position captured flag
  20 MACRO node reset (power-on or command)
  21 Ring break detected elsewhere
  22 Amplifier enabled at station
  23 Amplifier/node shutdown fault
  24 Home flag input value
  25 Positive limit flag value
  26 Negative limit flag value
  27 User flag value
  28 W flag value
  29 V flag value
  30 U flag value
  31 T flag value

- **MACRO Command Flag Registers (Gate3[*i*].MacroOutA[*j*][3]/Gate3[*i*].MacroOutB[*j*][3] and Gate2[*i*].Macro[*j*][3])**
  Bit 0 – 18 (Reserved for future use)
  19 Position capture enable
  20 Node position reset flag
  21 Ring break detected
  22 Amplifier enable
  23 – 31 (Reserved for future use)

Nodes and Addressing Structure

➤ **The contents of I/O nodes is arranged as follows for Gate3-style MACRO:**



Each I/O node, with Power PMAC3 MACRO style IC, possesses structure elements for inputs and outputs separately:

PMAC3 Style MACRO IC	
Inputs	**Outputs**
Gate3[*i*].MacroInA[*j*][0]	Gate3[*i*].MacroOutA[*j*][0]
Gate3[*i*].MacroInA[*j*][1]	Gate3[*i*].MacroOutA[*j*][1]
Gate3[*i*].MacroInA[*j*][2]	Gate3[*i*].MacroOutA[*j*][2]
Gate3[*i*].MacroInA[*j*][3]	Gate3[*i*].MacroOutA[*j*][3]

I/O Node Registers
Upper 24 bits
Upper 16 bits
Upper 16 bits
Upper 16 bits

Note

The above I/O node addresses represent Bank A. For Bank B addressing, replace the suffix A with B.

MACRO Motor Setup

Overview

- Step 1: Configure System Clocks for Master
- Step 2: Enable Nodes and Error Testing Parameters on Master
- Step 3: Establish Communication with Slave
- Step 4: Enable Nodes and Error Testing Parameters on Slave
- Step 5: Configure System Clocks for Slave
- Step 6: Point Motor Pointers to MACRO Registers
- Step 7: Tune Motor as Normal

Step 1: Configure Clocks

- **Gate3 Clocks (i.e. the Master EtherLite's clocks) are determined by the following structures:**

Gate3[*i*].PhaseServoDir	// =0 if this IC outputs clocks, =3 if not
Gate3[*i*].PhaseFreq	// Determines the Phase Clock Frequency
Gate3[*i*].ServoClockDiv	// Determines the Servo Clock Frequency
Sys.ServoPeriod	// Reports the Servo Period to Internal Software
Sys.PhaseOverServoPeriod	// Reports the Phase Period to Internal Software

Note

All Power PMAC EtherLite Masters have Gate3-based MACRO hardware. In contrast, no MACRO Slave product currently has Gate3-based MACRO hardware.

Note

There are other clock settings in the Power PMAC Master, but they are related to PWM, encoders, ADCs and DACs, and are not completely relevant to this training.

Step 1: Configure Clocks

- **Gate3[*i*].PhaseServoDir Settings:**
  Gate3[*i*].PhaseServoDir = 0: Internal phase clock, internal servo clock
  Gate3[*i*].PhaseServoDir = 1: External phase clock, internal servo clock
  Gate3[*i*].PhaseServoDir = 2: Internal phase clock, external servo clock
  Gate3[*i*].PhaseServoDir = 3: External phase clock, external servo clock

→Generally, this value is set correctly at $$$***, but to ensure the value is correct:
The Gate3 on the Master with the lowest-numbered index should set this structure to 0, and all other clock-generating devices in the rack should be set to 3.

Step 1: Configure Clocks

- **Gate3[*i*].PhaseFreq**
  This structure determines the phase clock. Make sure that the phase clock is set identically on the MACRO Master (the EtherLite) and all of the Slave devices on the MACRO ring.

  Set **Gate3[*i*].PhaseFreq** equal to the frequency you want for the Phase Clock [Hz].

  Example (for Gate3 at index 0):

```
Gate3[0].PhaseFreq=9035.69161891937256; // Phase Clock: 9.035 kHz
```
PPMAC Script

Step 1: Configure Clocks

- **Gate3[*i*].ServoClockDiv**
  This structure determines the servo clock on the Master. This does not need to be set the same between Master and Slave.

  The formula to use for this structure is:

$$\textbf{Gate3}[i]\textbf{.ServoClockDiv} = \frac{(\text{Phase Frequency})[\text{kHz}]}{(\text{Servo Frequency})[\text{kHz}]} - 1,$$

Example (for Gate3 at index 0):

```
Gate3[0].ServoClockDiv=Gate3[0].PhaseFreq/2.259 – 1.0 // = 3
                                                     // for 2.259 kHz Servo
```

PPMAC Script

Step 1: Configure Clocks

- **Sys.ServoPeriod**
  This structure does not **set** any clocks, but it is needed for reporting the servo period to internal programs.

  The formula to use for this structure is:

$$\textbf{Sys.ServoPeriod} = 1000 \cdot \frac{(\textbf{Gate3}[i]\textbf{.ServoClockDiv}+1)}{\textbf{Gate3}[i]\textbf{.PhaseFreq}},$$

Example (for Gate3 at index 0):

```
Sys.ServoPeriod=1000.0*(Gate3[0].ServoClockDiv+1.0)/Gate3[0].PhaseFreq
```
PPMAC Script

Step 1: Configure Clocks

- **Sys.PhaseOverServoPeriod**
  This structure does not **set** any clocks, but it is needed for reporting the phase period to internal programs.

  The formula to use for this structure is:

$$\mathbf{Sys.PhaseOverServoPeriod} = \frac{1}{\mathbf{Gate3}[\boldsymbol{i}].\mathbf{ServoClockDiv} + 1},$$

Example (for Gate3 at index 0):

```
Sys.PhaseOverServoPeriod=1/(Gate3[0].ServoClockDiv+1)
```
PPMAC Script

Step 1: Configure Clocks

- Below is an example setup file with clock settings grouped together

```
Sys.WpKey=$AAAAAAAA;
Gate3[0].PhaseServoDir=0; // This Gate3 transmits clocks; set =3 to receive clocks
Gate3[0].PhaseFreq=9035.69161891937256; // Phase Clock: 9.035 kHz
Gate3[0].PhaseClockMult=0; // Do not multiply output phase clock
Gate3[0].PhaseClockDiv=0; // Do not divide down internal phase clock
Gate3[0].ServoClockDiv=3; // Servo Clock: 2.259 kHz
Sys.ServoPeriod=1000*(Gate3[0].ServoClockDiv+1)/Gate3[0].PhaseFreq;
Sys.PhaseOverServoPeriod=1/(Gate3[0].ServoClockDiv+1);
Sys.WpKey=0;
```

PPMAC Script

- **Putting this into a "clock settings.pmh" file (or similar name) in the Global Includes folder in your IDE project is recommended.**

Note

Issuing Sys.WpKey=$AAAAAAAA is required before modifying any Gate3 structures.

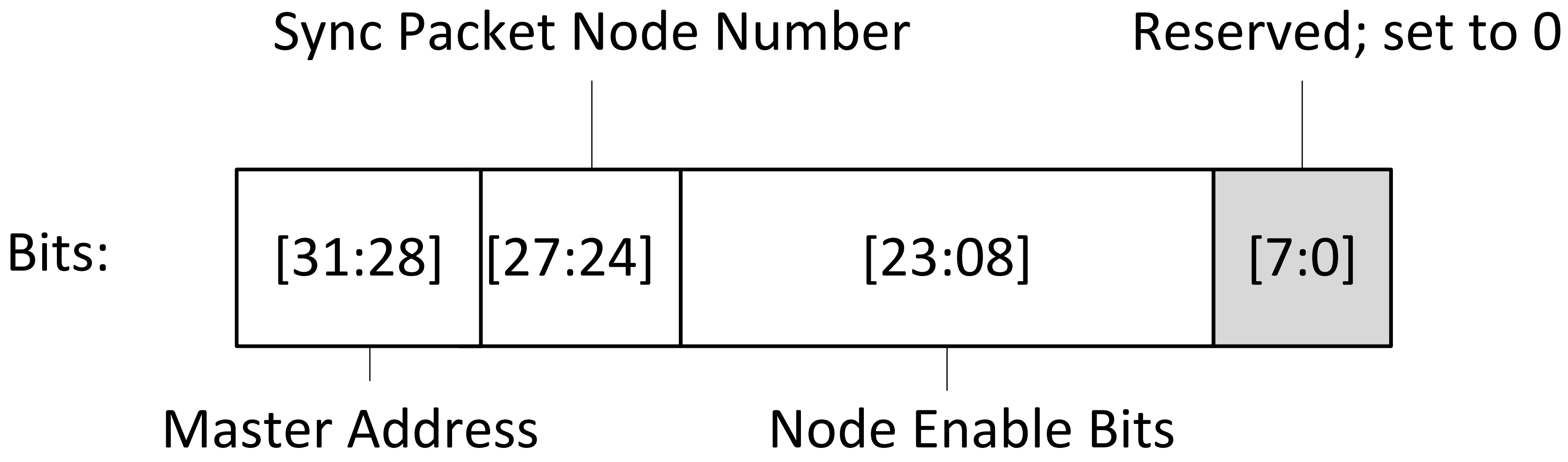
Step 2: Enable Nodes on the Master

- You must enable the same servo node on the Master that you enable on the node
- Enabling a node begins the transfer of feedback, command, and flag data
- Enabling nodes is done through the following structures:

Gate3[*i*].MacroEnableA	// Node Enable for MACRO Bank A
Gate3[*i*].MacroEnableB	// Node Enable for MACRO Bank B
Gate3[*i*].MacroModeA	// Communication mode for MACRO Bank A
Gate3[*i*].MacroModeB	// Communication mode for MACRO Bank B

Step 2: Enable Nodes on the Master

- **Gate3[*i*].MacroEnableA and Gate3[*i*].MacroEnableB are both 32-bit words comprising the following information:**



- Each of the Node Enable Bits controls a node. Bit 0 controls node 0, bit 1 node 1, etc. A value of 0 in the bit disables the node, and a value of 1 enables the node.
- Sync Packet Node Number: Set these four bits to 1 to indicate that node 15 should be used.
- Master Address: Specifies the number of the master address for this IC. For Bank A, set this to 0; for Bank B, set this to 1.

Step 2: Enable Nodes on the Master

- **Gate3[*i*].MacroModeA and Gate3[*i*].MacroModeB are both 32-bit words comprising the following information:**

Component	Bits	Hex Digit #	Functionality
(Reserved)	31 – 24	1 – 2	(Reserved for future use)
MacroMasterChkDisA	23 – 16	3 – 4	MACRO A master check disable
MacroSyncEnaA	15	5	MACRO A phase clock sync enable
MacroSyncRcvdA	14	5	MACRO A sync packet received status
MacroStationTypeA	13 – 12	5	MACRO A station type
MacroUnderrunErrA	11	6	MACRO A data underrun error status
MacroParityErrA	10	6	MACRO A parity/CRC error status
MacroCodeErrA	09	6	MACRO A byte coding error status
MacroOverrunErrA	08	6	MACRO A data overrun error status
(Reserved)	07 – 00	7 – 8	(Reserved for future use)

- **The status bits in bits [11:08] can be useful for troubleshooting**
- **Generally speaking, the only settings that need to be changed are as follows:**
  For Master devices, set bits [13:12] both to 1; slaves, set them to 0.
  For Master Devices, set bit 22 to 1 to permit node 15 to be used for broadcasting; for slaves, leave this at 0.

- This makes the usual setting for a Synchronizing Master Gate **$403000**, and $**9000** for a Non-Synchronizing Master Gate.

Step 2: Enable Nodes on the Master

- **There are three parameters for MACRO error checking which must be set :**

Macro.TestPeriod
This is the period in servo cycles at which PMAC checks for errors on the MACRO ring. The recommended value for this variable is 50.

Macro.TestMaxErrors
This is the maximum error count PMAC can receive in one test period (whose duration is specified by **Macro.TestPeriod**) before triggering a fault. The recommended value is 2, meaning that the ring would shut down on a third error in a given evaluation period.

Macro.TestReqdSynchs
This is the number of sync packets in one period (whose duration is specified by **Macro.TestPeriod**) that PMAC must receive before triggering an error. The recommended value is 2, meaning that the ring would shut down if only 0 or 1 sync packets were received per test period.

Step 2: Enable Nodes on the Master

- **Below is example setup code for configuring node 0 (one servo node) and error testing:**

```
Sys.WpKey=$AAAAAAAA;

//MACRO Communication Setup
// Activate 1 Servo Node and 0 IO Nodes of MACRO A
Gate3[0].MacroEnableA=$0FC00100;
Gate3[0].MacroModeA=$403000;  // Set MACRO A as master, sync to no other clock

// Activate 0 Servo Nodes and 0 IO Nodes of MACRO B
Gate3[0].MacroEnableB=$1FC00000;
Gate3[0].MacroModeB=$9000;  // Set MACRO B as master, synchronize to A’s clock

// MACRO Ring Check Period [servo cycles] (Related to I80 in Turbo)
Macro.TestPeriod=50;

// MACRO Maximum Ring Error Count (Related to I81 in Turbo)
Macro.TestMaxErrors=2;

// MACRO Minimum Sync Packet Count (Related to I82 in Turbo)
Macro.TestReqdSynchs=2;
Sys.WpKey=0;
```
PPMAC Script

Note

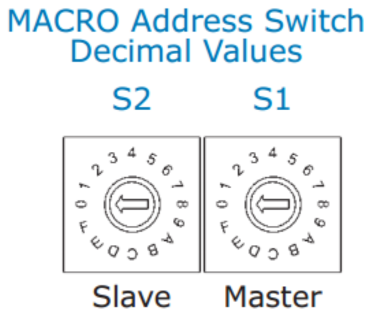
Issuing Sys.WpKey=$AAAAAAAA is required before modifying any Gate3 structures.
It is recommended to keep these settings in a “macro settings.pmh” file in the Global Includes folder of your IDE project.

Step 3: Communicate with the Slave

- The first step of communicating with the Slave is setting its Station Number and Master Number.
- Depending what the MACRO Slave Device is, the MACRO Station Number and Master Number will either be set by rotary switches (in this case, see the device's hardware reference manual) or the Ring Order Method, where the station number gets set automatically and sequentially based on its position in the ring.

Step 3: Communicate with the Slave

- Example of MACRO Station Number set with rotary switches (from Copley Accelnet MACRO Drive):

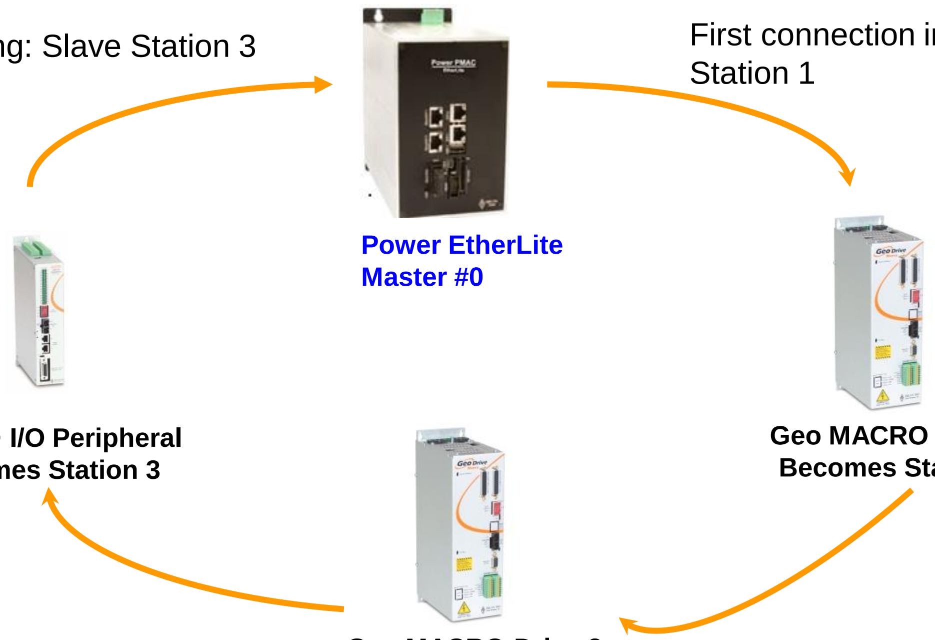


Switch	S2	S1
Address	SLAVE	MASTER
HEX	DEC	
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	I/O	8
9	I/O	9
A	I/O	10
B	I/O	11
C	I/O	12
D	I/O	13
E	RSVD	14
F	RSVD	15

- No command needs to be issued to the Slave; the Station Number is entirely set by these hardware switches.

Step 3: Communicate with the Slave

- **Example of Ring Order Method being used to set Station Numbers**



Last connection in ring: Slave Station 3 to Master

Power EtherLite Master #0

First connection in ring: Master to Slave Station 1

MACRO I/O Peripheral Becomes Station 3

Geo MACRO Drive 1 Becomes Station 1

Geo MACRO Drive 2 Becomes Station 2

Third connection in ring: Slave Station 2 to Slave Station 3

Second connection in ring: Slave Station 1 to Slave Station 2

Step 3: Communicate with the Slave

- **When setting up Station Numbers through the Ring Order Method, the Stations must first be initialized to factory default, because before initialization, the Stations will all be at Station #255. Some common examples of how to initialize are as follows:**

  - If using MACRO IC #0 (Bank A of Gate3[0]), to reinitialize the slave, type MacroSlave$$$***15, then MacroSlaveSAVE15, then MacroSlave$$$15.
  - If using MACRO IC #1 (Bank B of Gate3[0]), type MacroSlave$$$***31, then MacroSlaveSAVE31, then MacroSlave$$$31.
  - If using MACRO IC #2 (Bank A of Gate3[1]), use MacroSlave$$$***47, then MacroSlaveSAV47, then MacroSlave$$$47, and so on for other MACRO IC #s.

- Then, the Station Numbers can be set automatically with the **MacroRingOrderInit0** command
- If you want to set a Station Number manually, type **MacroStation255** to access the station whose number has not been set.
- Type **I11=n** in order to assign this device to Station #**n**.
- Type **<MacroStationClose** to exit MACRO ASCII Mode
- Type **MacroStation<n>** (without the brackets) to begin communicating with Station **n**.

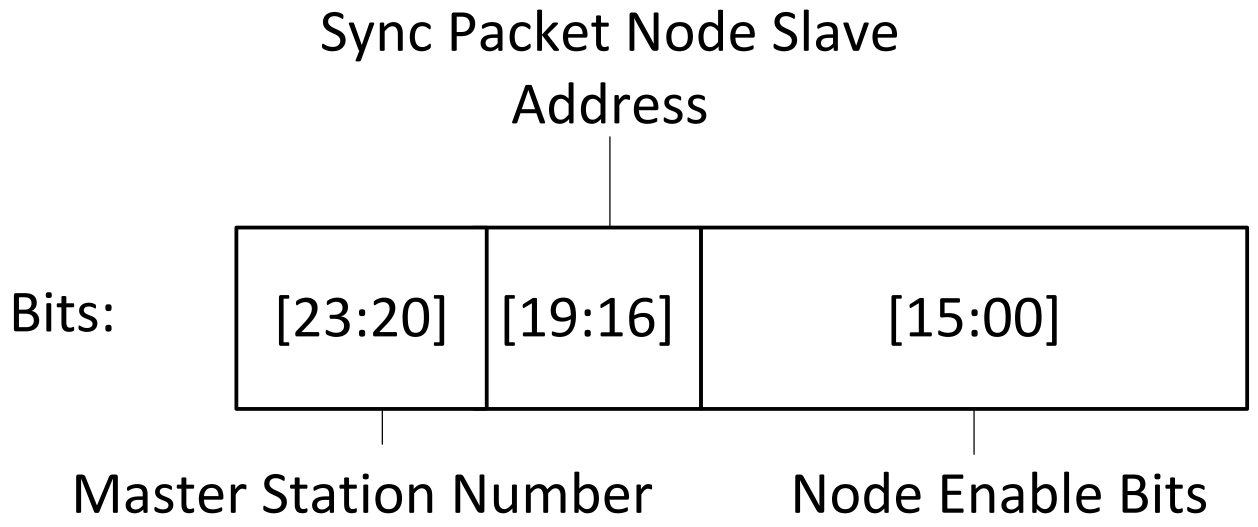
Step 4: Enable Nodes on the Slave

- **The MACRO parameters that need to be configured on the Slave are as follows:**

MI996	// Configures which nodes to enable on the Slave
MI995	// Configures which Master IC to use
MI8	// Ring Check Error Period
MI9	// Ring Error Shutdown Count
MI10	// Sync Packet Shutdown Count

Step 4: Enable Nodes on the Slave

- **MI995 is a 24-bit word comprising the following information:**



- Each of the Node Enable Bits controls a node. Bit 0 controls node 0, bit 1 node 1, etc. A value of 0 in the bit disables the node, and a value of 1 enables the node.
- Sync Packet Slave Address specifies the slave number of the packet that will generate the "sync pulse" on the Station. Always set all of these bits to 1 on the Slave.
- Master Address: Specifies the number Master IC. Can be set 0 to 15.

Step 4: Enable Nodes on the Slave

- **MI996 is a 24-bit word comprising the following information:**

Bit #	Value	Type	Function
0	1($1)	Status	Data Overrun Error (cleared when read)
1	2($2)	Status	Byte Violation Error (cleared when read)
2	4($4)	Status	Packet Parity Error (cleared when read)
3	8($8)	Status	Packet Underrun Error (cleared when read)
4	16($10)	Config	Master Station Enable
5	32($20)	Config	Synchronizing Master Station Enable
6	64($40)	Status	Sync Node Packet Received (cleared when read)
7	128($80)	Config	Sync Node Phase Lock Enable
8	256($100)	Config	Node 8 Master Address Check Disable
9	512($200)	Config	Node 9 Master Address Check Disable
10	1024($400)	Config	Node 10 Master Address Check Disable
11	2048($800)	Config	Node 11 Master Address Check Disable
12	4096($1000)	Config	Node 12 Master Address Check Disable
13	8192($2000)	Config	Node 13 Master Address Check Disable
14	16384($4000)	Config	Node 14 Master Address Check Disable
15	32768($8000)	Config	Node 15 Master Address Check Disable

- For MACRO Slaves, configuration bits 4 and 5 are set to 0.
- Slaves should synchronize themselves to the sync node, so configuration bit 7 should be set to 1.
- In most applications, slaves will accept only packets from their own master so bits 8 to 15 are all set to 0. All other bits are status bits that are normally 0.
- This makes the usual setting of **MI995** equal to **$0080**.

Step 4: Enable Nodes on the Slave

- **There are three parameters for MACRO error checking which must be set:**

MI8
This is the period in units of phase cycles at which PMAC checks for errors on the MACRO ring. It should be set according to the following formula:

$$MI8 = 50\frac{MACRO\ Ring\ Rate\ (Phase\ Rate\ of\ All\ Devices)}{Ring\ Controller\ Servo\ Rate}$$

For example, if the phase clock of all devices on the ring is twice that of the servo clock of the ring controller (e.g. Power PMAC), MI8 would be set to 100.

MI9
This is the maximum error count PMAC can receive in one test period (whose duration is specified by **MI8**) before triggering a fault. The recommended value is 2, meaning that the ring would shut down on a third error in a given evaluation period.

MI10
This is the number of sync packets in one period (whose duration is specified by **MI8**) that PMAC must receive before triggering an error. The recommended value is 2, meaning that the ring would shut down if only 0 or 1 sync packets were received.

Step 4: Set the Slave Nodes

- Assuming the MACRO Station numbers have been successfully set already, type MacroStation<n> (without the brackets) to begin communicating with Station n.
- Below is an example of setting up one servo node on the Slave. This example can be typed into the Terminal window after establishing MACRO ASCII communication with the slave:

```
MI996=$FC001        // Enable one node, use Master IC 0
MI995=$80           // Standard MACRO Mode for a Slave
MI8=25              // Ring Check Period
MI9=3               // Max Ring Check Errors, =MI8/10, rounded up
MI10=22             // Required synch packets per check period, =MI8-MI9
```

MACRO Script

- After issuing these commands, issue <MacroStationClose and MacroStationClose to close MACRO ASCII communication.

Step 5: Configure Clocks on the Slave

- **The MACRO Slave Devices' clocks are determined by the following parameters:**

MI992	// MaxPhase Frequency Control
MI997	// Phase Clock Frequency Control
MI998	// Servo Clock Frequency Control

Note

Remember to set the Phase Clock on the Slave to the same frequency as it is on the Master!

Note

There are other clock settings in MACRO Slave devices, but they are related to PWM, encoders, ADCs and DACs, and are not completely relevant to this training.

Step 5: Configure Clocks on the Slave

- **The formulas for computing MI992, MI997, and MI998 are as follows:**

$$\text{MI992} = \left(\frac{117964.8}{2 \cdot f_{mp}}\right) - 1$$

$$\text{MI997} = \left(\frac{f_{mp}}{f_p}\right) - 1$$

$$\text{MI998} = \left(\frac{f_p}{f_s}\right) - 1$$

f_{mp} is the desired maximum phase frequency [kHz], f_p is the desired phase clock frequency [kHz], and f_s is the desired servo clock frequency [kHz].

Example: When Node 0 is being used for the Slave, setting default clocks

```
MacroSlave0,MI992=6527
MacroSlave0,MI997=0
MacroSlave0,MI998=3
```

PPMAC Script

Then, issue **MacroSlaveSAVE15** followed by **MacroSlave$$$15** to save the changes on the Station.

Step 6: Assign Motor Pointer Registers

- **Position Feedback Address**

  Motor[*x*].pEnc and **Motor[*x*].pEnc2** must be pointed to **EncTable[*n*].a** for the Encoder Conversion Table

  Entry that processes the MACRO motor's encoder, assuming a single sensor for the motor.

  Generally:
  Select **EncTable[*n*].type=4** for 32-bit single-register read over MACRO.
  Point **EncTable[*n*].pEnc** to **Gate3[*i*].MacroInA[*j*][0].a** or **Gate3[*i*].MacroInB[*j*][0].a**.
  Set **EncTable[*n*].index2=8** to eliminate the empty bottom 8 bits of this register.
  Set **EncTable[*n*].index1=8** to put the position data at the MSB of this register.
  Set **EncTable[*n*].ScaleFactor=1/exp2(EncTable[*n*].index1)**
  If you do not have a full 24 bits of data, you may need to increase index1's value.
  You may also want to consider setting a MaxChange filter on the ECT.

- **Command Output Address**

  Set **Motor[*x*].pDac = Gate3[*i*].MacroOutA[*j*].[0].**a or **Gate3[*i*].MacroOutB[*j*][0].a**

- **Input Flag Addresses**

  Set **Motor[*x*].pLimits**, **Motor[*x*].pAmpFault**, and **Motor[*x*].pCaptFlag** to **Gate3[*i*].MacroInA[*j*][3].a** or **Gate3[*i*].MacroInB[*j*][3].a**.

Step 6: Assign Motor Pointer Registers

- **Input Flag Bits**

  The user must specify which bits of the above Flag Addresses to use for which flags as follows:

  Motor[*x*].AmpFaultBit = 23
  Motor[*x*].LimitBits = 25
  Motor[*x*].CaptFlagBit = 19

  When using quadrature encoder feedback with 5 bits of 1/T sub-count extension, the following settings should be used to process whole-count captured data, as for homing:

  Motor[*x*].CaptPosShiftLeft = 13
  Motor[*x*].CaptPosShiftRight = 0
  Motor[*x*].CaptPosRound = 1

Step 6: Assign Motor Pointer Registers

- **Output Flag Addresses**
  Set **Motor[*x*].pAmpEnable=Gate3[*i*].MacroOutA[*j*][3].a** or **Gate3[*i*].MacroOutB[*j*][3].a**

- **Output Flag Bits**
  Set **Motor[*x*].AmpEnableBit=22**

- **Commutation Addresses**
  If Power PMAC is performing the phase commutation for a motor controlled over MACRO, set **Motor[*x*].PhaseCtrl=4** for “unpacked” data transfer.

  Set **Motor[*x*].pPhaseEnc=Gate3[*i*].MacroInA[*j*][0].a** or **Gate3[*i*].MacroInB[*j*][0].a**

  Use **Motor[*x*].PhaseEncLeftShift** and **PhaseEncRightShift**, if needed, to shift out garbage from the lower bits of the register, and shift back up to move the MSB to bit 31.

Step 6: Assign Motor Pointer Registers

- **Phase Position Scale Factor**

  To scale the commutation data properly, set **Motor[*x*].PhasePosSf** according to this formula:

$$\textbf{Motor}[x]\textbf{.PhasePosSf} = \frac{2048}{N}$$

  where N is the number of LSBs of the feedback.

  - For most quadrature encoders, N will be 2^{13}, because there are 5 bits of fractional count data from 1/T extension and 8 bits of "garbage" at the bottom of the position register, yielding the LSB as 13. Thus, a common value for a rotary motor with a quadrature encoder commutated over MACRO will be **Motor[*x*].PhasePosSf=2048/exp2(13)**.

Modbus Setup

Overview

- **Modbus is a communication protocol designed for Machine Control**
  - Many different implementations of it over different connection types
  - Delta Tau uses a TCP/IP implementation known as Modbus TCP through the normal communication port
  - An Ethernet Switch or Hub may be used to allow for simultaneous communication with the IDE and Modbus devices
- **Server/Client**
  - One Device will be specified to be a server
  - Multiple different Clients can connect to a server and access its memory
- **Generally, the PMAC will function as a Server**
  - It will wait for multiple clients to connect to it and access its memory
  - A user-written PLC can then monitor for changes to the variables and react accordingly
  - Clients could be either devices such as HMIs or individual devices, such as relays
  - Configuring the PMAC as a Server is easy and just requires enabling
- **The PMAC can also function as a Client**
  - This requires more configuration, as it must be able to identify a specific Server
  - Can connect to multiple Servers using different ports
  - More complex, as the client performs the actual “work” of writing data into server memory and reading from it

Modbus Server Commands

Command	Description
Sys.ModbusServerEnable	Enables a socket for Modbus Server Communication
ModbusServerQuery	Displays the last Modbus Client Query to the Server
ModbusServerResponse	Displays the last response from the Modbus Server to a Client
ModbusServerState	Displays the status of one or more Server Sockets
ModbusServerLinuxError	Displays any Linux Socket Errors for one or more Sockets

Modbus Server

- To configure a Power PMAC as a Modbus server, set Sys.ModbusServerEnable = 1 and wait for Client Communication
- Clients can then access thousands of I, M, P, and Q Variables from their Modbus addresses
- More data can be stored in other registers accessible by Sys.ModbusServerBuffer

Modbus Reference Address	Access from PPMAC
0x0000 – 0x0FFF	Sys.ModbusServerBuffer[0-8190]
0x1000 – 0x3FFF	I-Variables 0-6143
0x4000 – 0x7FFF	M-Variables 0-8191
0x8000 – 0xBFFF	P-Variables 0-8191
0xC000 – 0xEFFF	Q-Variables 0-1023 for Coordinate Systems 0-5

- You can then monitor your registers from inside of a PLC to react appropriately

Note

Mulytiply the Reference Address by 2 to access a parameter by Sys.ModbusServerBuffer.

Modbus Client Commands

Command	Description
Modbus[*x*].Config.ServerPort	Select which port to use (0 defaults to port 502)
Modbus[*x*].Config.ConnectTimeOut	Minimum of 5200 msec (zero defaults to OS Timeout)
Modbus[*x*].Config.SendRecvTimeOut	Minimum is 3100 msec (zero defaults to 6300 msec)
Modbus[*x*].Config.ServerMode	Determines whether MSW or LSW is transmitted first (0 defaults to MSW, LSW)
Modbus[*x*].Config.UnitID	ID Number sent with Modbus message to identify device
Modbus[x].Config.ServerIpAddr[]	IP Address of Modbus server, broken up into 4 bytes

Modbus Client Commands, Cont'd.

Command	Description
Modbus[*x*].Config.KeepAliveEnable	Enables socket KEEPALIVE feature
Modbus[*x*].Config.KeepAliveIdle	Determines how long before the first KEEPALIVE request are sent to the server over TCP when no Modbus queries are being sent, in seconds
Modbus[*x*].Config.KeepAliveInterval	Determines how frequently to send KEEPALIVE requests after the first one is sent, in seconds
Modbus[*x*].Config.KeepAliveCnt	Determines the total number of KEEPALIVE messages that will be sent

Modbus Client On-Line Commands

Command	Description
ModbusPing #	Pings for a Modbus device at the address specified by Modbus[x].Config.ServerIPAddr[]
ModbusConnect #	Attempt to connect to the specified server
ModbusAscii #	Issue ASCII strings to the specified server
ModbusState #	Determines the status of the server
ModbusClose #	Close communication with the specified server
ModbusCloseAll	Close communication with all servers
ModbusRegister	Read/Write a specified address as a register
ModbusFloat	Read/Write a specified address as a float
ModbusDiscrete	Read a specified number of discretes (input bits)
ModbusCoil	Read/Write a specified number of coils (output bits)
ModbusQuery #	Displays the last Modbus client query to the Modbus server
ModbusResponse #	Displays the last Modbus server response to the client query

Modbus Client

- To configure a Power PMAC as a Modbus server, set Sys.ModbusServerEnable = 1 and wait for Client Communication
- Clients can then access thousands of I, M, P, and Q Variables from their Modbus addresses
- More data can be stored in other registers accessible by Sys.ModbusServerBuffer

Modbus Reference Address	Access from PPMAC
0x0000 – 0x0FFF	Sys.ModbusServerBuffer[0-8190]
0x1000 – 0x3FFF	I-Variables 0-6143
0x4000 – 0x7FFF	M-Variables 0-8191
0x8000 – 0xBFFF	P-Variables 0-8191
0xC000 – 0xEFFF	Q-Variables 0-1023 for Coordinate Systems 0-5

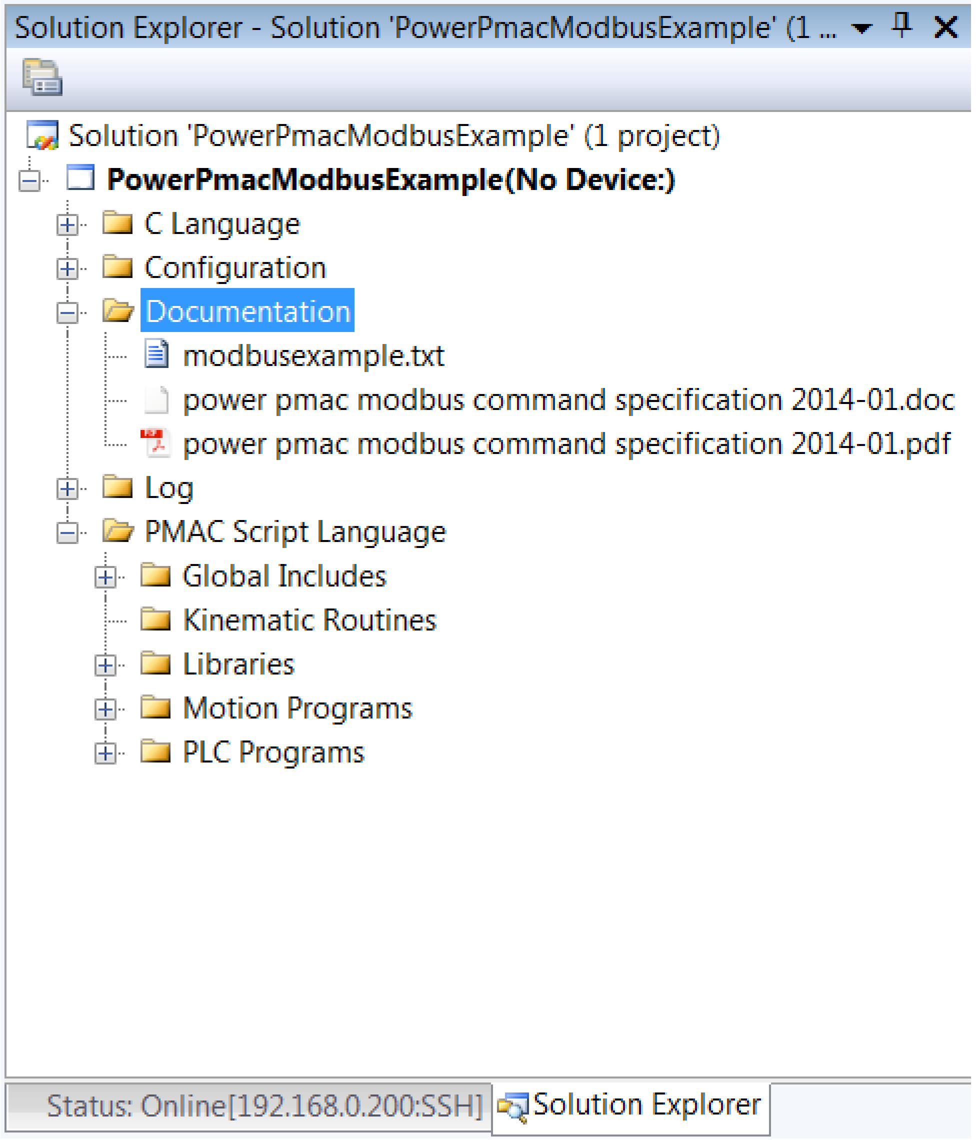
- You can then monitor your registers from inside of a PLC to react appropriately

Note

Mulytiply the Reference Address by 2 to access a parameter by Sys.ModbusServerBuffer.

More Information

- Sample project in the IDE installation
- Generally in “C:\DeltaTau\Power PMAC IDE\PowerPMACProjectExamples\PowerPmacModbusExample”
- Sample project opens both a socket and a client connection to the socket
- Documentation folder contains list of full parameters for Modbus in addition to explanations of each of the commands


Solution Explorer - Solution 'PowerPmacModbusExample' (1 ...

```
Solution 'PowerPmacModbusExample' (1 project)
  PowerPmacModbusExample(No Device:)
    C Language
    Configuration
    Documentation
      modbusexample.txt
      power pmac modbus command specification 2014-01.doc
      power pmac modbus command specification 2014-01.pdf
    Log
    PMAC Script Language
      Global Includes
      Kinematic Routines
      Libraries
      Motion Programs
      PLC Programs
```

Status: Online[192.168.0.200:SSH] Solution Explorer

Connecting to PowerPmacNC 16

Overview

- **PowerPmacNC 16 comes in two versions, with and without the PDK**
  - With the PDK, the software is fully customizable
  - We will focus on connecting to the default configuration of it
- **Set up your Power PMAC in the IDE**
  - Configure all axes, I/O, and system logic
  - We will use 3 Axes for our system
  - This requires us to set up at least one virtual motor on our system
  - A full configuration with 8 virtual motors is in the sample project included with the NC Software
  - Make sure that the USB Hardware Key is inserted into the system
- **Find the PowerPmacNC.ini file in \PowerPmacNC\bin\Debug**
  - Look for the lines that specify the Axis Labels and Line Numbers
  - Verify that each line only has 3 terms in it
- **Run the NC Software**
  - The specific file is “\PowerPmacNC\bin\Debug\PowerPmacNc.exe”
  - Once it loads, press “Log In” to go to the main screen

Machine View

- **Select the Machine View Tab in the top left corner**
  The Machine View tab details the general status of the system
  We will focus on connecting to the default configuration of it

- **Controller**
  We will go here and verify that the IP Address is correct
  Displays the status of the Power PMAC
- **Axes**
  Displays the positions, torques, and feedrates for the system
- **Tool**
  Displays the known parameters about the spindle tooltip
- **NC File**
  Displays statistics for the current NC file
- **G-Codes**
  Displays the G-Codes currently loaded into the system
- **M-Codes**
  Displays the M-Codes currently loaded into the system
- **Tool Offsets**
  Displays the Tool Offset Table, if one is loaded
- **Work Offsets**
  Displays the Work Offsets Table, if any are loaded

- **After verifying the device IP Address, Press "Go Online" and then "Run"**

Passing Parameters to PowerPmacNC 16

- **PowerPmacNC 16 automatically queries predefined variables to get its parameters**
  In the “PPCNC_ProjectSource” project, they are detailed in “ppnc_ncinterfacedefinitions.pmh”
  Some key parameters are outlined below
- **P10X: Axis Position**
  P100 for the X Axis, P101 for Y Axis, P102 for Z Axis, etc.
- **P11X: Distance to Go**
- **P12X: Torque**
- **P14X: Following Error**

- **The parameters must be calculated in a PLC to force them to continually update**
- **A Sample PLC to perform this is detailed in “ppnc_positionreport.plc” in the sample project**