PSoC Programmer

Component Object Model (COM) Interface Guide

Document Number. 001-45209 Rev. *T

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com

Copyrights

Contents

1. Overview

The *Component Object Model (COM) Interface Guide* describes the COM interface that you can use as an alternative to the command line interface (CLI) for PSoC Programmer. The command line interface is described in the *Command Line Interface Guide* (*PSoC Programmer CLI User Guide.pdf* in the Documents directory). The Cypress engineering programmers are not recommended for mass production programming environments due to their mechanical design, plastic enclosures, and plastic headers. The Miniprog3 provides over current protection and over voltage protections, but due to the manufacturing environments the engineering programmers might be subjected to damaging conditions. It is recommended that you use mass production programmers for manufacturing. You can find the complete list of recommended vendors here.

1.1 Using the COM Object - Early Binding

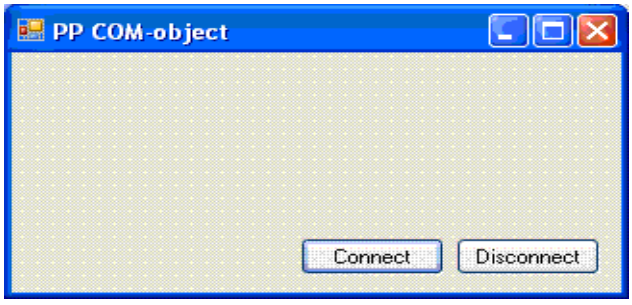
You can use the COM object to call PSoC Programmer from your application. The installer registers the PSoC Programmer COM object when it is installed. If, for any reason, the object is not registered, you can register it manually:

1. Select **Start** > **Run**.
2. Type the following command:
   ```
   PSoCProgrammerCOM.exe /regserver
   ```
   To unregister the COM object, use the following command:
   ```
   PSoCProgrammerCOM.exe /unregserver
   ```

The following example shows how to call the COM object from a Visual Studio project:

1. Start Microsoft Visual Studio and create a new C# project of Windows Application (Form) type.
2. Place two buttons on the form and give the corresponding objects the names *buttonConnect* and *buttonDisconnect.*

Figure 1-1. COM Object Application



3. Select **Project** > **Add Reference**.
4. In the **Add Reference** window, select the **COM** tab.
5. Find and select the **PSoCProgrammerCOM** component.
6. Click **OK**.

7. Add the PSOCPROGRAMMERCOMLib namespace to the source file. Switch into the Code View of the form. Using directives, add two more to the end of the list:
   ```
   using System.Diagnostics;
   using PSOCPROGRAMMERCOMLib;
   ```
8. Declare the **pp** variable.
9. In the class **Form1**, add the following code:
   ```
   PSoCProgrammerCOM_Object pp;
   ```
10. Double-click on each button to generate two event handlers for the **Connect** and **Disconnect** buttons. The two event handlers generated in the Form1 class are:
    - buttonConnect_Click
    - buttonDisconnectClick
11. Add two private methods as event handlers for the COM object:
    ```
    private void OnProgrammerConnected(string progID)
    private void OnProgrammerDisconnected(string progID)
    ```

The following code example shows how to use the event handlers added in the previous example. Using this example, you can connect programmers to or disconnect them from the PC, and then see the resulting messages. You can also use this sample code as a starting point to develop your own applications that use the PSoC Programmer COM object.

- In the *buttonConnect_Click* method, the code checks if the object exists or not. If it does, then it exits. It makes no sense for one application to have multiple PSoC Programmers running.
- The *buttonConnect_Click* method creates a new instance of the wrapper class. Once this instance is initiated, later events are connected from this point.
- The following three lines are optional, but recommended.
  ```
  int serverProcessID;
  int clientProcessID = Process.GetCurrentProcess().Id;
  serverProcessID = pp._StartSelfTerminator(clientProcessID);
  ```
  The *StartSelfTerminator* function starts a monitor on the COM object side that monitors the client. If the client terminates abnormally, then the Programmer object exits as well. It also returns the ProcessID of the COM object so that the client can kill the COM object if necessary.
- The last line of *button_ConnectClick* method is an example code, which gets the version of the COM object and shows it in the form's caption.

In the *buttonDisconnect_Click* handler, the COM object is detached from the application and frees the Windows resources.

```
public partial class Form1 : Form
    {
        PSoCProgrammerCOM_ObjectClass pp;

        public Form1()
        {
            InitializeComponent();
        }

        private void buttonConnect_Click(object sender, EventArgs e)
        {
            if (pp != null) return; //Programmer already started
            pp = new PSoCProgrammerCOM_ObjectClass();
            //Init Events here
            pp.Connected += OnProgrammerConnected;
            pp.Disconnected += OnProgrammerDisconnected;
            //Start Client monitor code - optional code, can be passed
```

```
            int serverProcessID;
            int clientProcessID = Process.GetCurrentProcess().Id;
            serverProcessID = pp._StartSelfTerminator(clientProcessID);

            //Get version of COM-object and print it in the form's caption
            this.Text = "PP COM-object "+pp.Version();
        }

        private void buttonDisconnect_Click(object sender, EventArgs e)
        {
            //Disconnect from COM-object and unload it
            pp = null;
            GC.GetTotalMemory(true);

            //Clear Form's caption
            this.Text = "PP COM-object";
        }

        private void OnProgrammerConnected(string progID)
        {
            MessageBox.Show("Connected: " + progID);
        }

        private void OnProgrammerDisconnected(string progID)
        {
            MessageBox.Show("Disconnected: " + progID);
        }
    }
```

1.2 Using the COM Object - Late Binding

The disadvantage of the early binding access is its dependency on the version of the COM object. After the client application is compiled with an early-bind wrapper, you must use only the version of the progranmer that existed on your computer during compilation. If you upgrade later to a newer version of the programmer, the client application does not work. This is not important for script languages, which use the version-neutral “ProgID” to access the COM object. However, precompiled clients must be updated to the most recent early binding wrapper and rebuilt. All client applications that require version-neutral access to the COM object should use late-binding methodology (wrapper).

A client developed for the current version of the programmer can work with the future releases without being rebuilt. This approach allows leveraging the latest and greatest updates of hardware access in custom applications.

Note that late-binding wrappers were not generated automatically by most of IDEs and thus they have to be created manually by developers.

PSoC Programmer project provides fully functional late-binding wrappers for .NET clients. The file PP_ComLib_Wrapper.dll is located in the installation folder.

The example of late-binding access in C++ can be found in the “Examples” folder. The scripting languages, such as Perl or Python, can access the COM object using a late-binding approach (depends on the interpreter). When instantiating the COM object, it is recommended that you use version-neutral ProgID in your clients: “PSoCProgrammer.PSoCProgrammerCOM_Object” (the version-dependent ProgID is “PSoCProgrammer.PSoCProgrammerCOM_Object.9”).

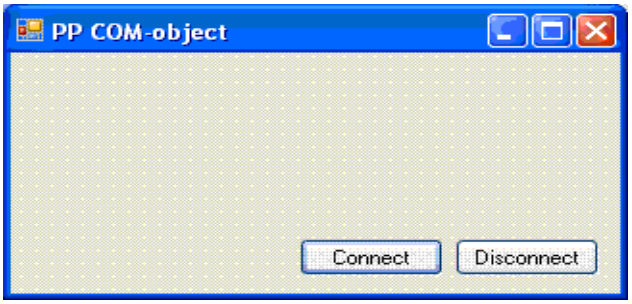
Because it is currently a popular development tool (Visual Studio 2005/2008/2010), we focus our attention on C# (.NET) examples.

Note, that PP_ComLib_Wrapper contains all of the APIs from the COM interface described in this guide. It also consists of APIs that can create or destroy this class, and can connect to or disconnect from the Programmer.

The quick start example is as follows. It is similar to the example given for early-binding access previously in this guide.

1. Create a C# project.
2. Start Microsoft Visual Studio and create a new C# project of Windows Application (Form) type.

Figure 1-2. PP_COMLib_Wrapper Application



3. Select **Project** > **Add Reference**.
4. In the **Add Reference** window, select the **Browse** tab.
5. Go to the PP install directory; find and select the *PP_ComLib_Wrapper.dll*. Click **OK**.
6. Add the **PP_ComLib_Wrapper** namespace to the source file. Switch into the Code View of the form. Add to the end of the list, using directives:
   ```
   using PP_ComLib_Wrapper;
   ```
7. Declare the **pp** variable. In the class Form1 add following code:
   ```
   PP_ComLib_WrapperClass pp;
   ```
8. Double-click on each button to generate two event handlers for the Connect and Disconnect buttons. The two events handlers generated in the Form1 class are:
   - `buttonConnect_Click;`
   - `buttonDisconnect_Click;`
9. The following code example shows how to connect or disconnect to the specified COM object via PP_COMLib_Wrapper library.
   - In the buttonConnect_Click method, the code checks whether PSoC Programmer is installed on the PC and if there is an application connected to the COM object. If it is not connected to the COM object, it performs the **Connect** action.
   - In the buttonDisconnect_Click method, the code checks whether any application is connected to the COM object. If it is connected, it performs the **Disconnect** action.

```
public partial class Form1 : Form
{
//Programmer Instance
PP_ComLib_WrapperClass pp;
public Form1()
{
InitializeComponent();
```

```
    }
    //GUI event handlers
    private void Form1_Load(object sender, EventArgs e)
    {
    pp = new PP_ComLib_WrapperClass();
    }
private void buttonConnect_Click_Click(object sender, EventArgs e)
{
 if (!pp.w_IsConnected())
{
 if (pp.w_ConnectToLatest() == 0)
  this.Text = "PP COM-object " + pp.Version();
  else
  this.Text = "No any PP is installed on your PC";
  }
}
private void buttonDiconnect_Click_Click(object sender, EventArgs e)
   {
   if (pp.w_IsConnected())
{
   pp.w_Disconnect();
   this.Text = "PP COM-object " + pp.Version();
   }
 }
}
```

1.3 Parameters

The description of the commands in this document are provided in pseudocode. The description is accompanied by a syntax example in C#. INs and OUTs are hex numbers unless otherwise noted.

- IN means it is an input
- OUT means it is returned
- *string* means that the argument is a string
- *svector* is a vector of strings (separated by new lines)
- *nvector* means that this argument is a vector of hex numbers (separated by spaces).

Consider the interface of the CheckSum(…) API in pseudo-language:

int CheckSum(int blockID, ushort OUT checksum, string OUT strError)

Most PSoc Programmer COM functions return a status that shows the function name and a string representation of the status or error. Other functions return a string or void value.

It is recommended that you analyze the return values before processing the returned parameters or moving to the next function. If a function fails, some parameters may remain in an uninitialized state. Processing these parameters may cause an exception or run-time error.

To analyze the status, you can use the following function:

```
private bool SUCCEEDED(long hr)
   {
      return (hr >= 0);
   }
```

COM functions return either a negative or positive integer to indicate success or failure:

- 0 or greater = Success
- Negative integer = Failed

IFor example:

```
string strError;
ushort checksum;
r = pp.checksum(0, out checksum, out strError);
      if (SUCCEEDED(r))
         AddToLogNL("--->Flash Checksum: " + checksum.ToString("X4"));
      else
         AddToLogNL("ERROR! Checksum operation FAILED "+strError);
```

1.4 Command Overview

Table 1-1. PSoC Programmer APIs

APIs	Chip APIs						Port APIs	Programmer APIs	Protocols	Misc
	Common	PSoC 1	PSoC 3/5	PSoC 4	FM0	PSoC 6				
_StartSelfTerminator(...)										X
Acquire(...)		X								
AcquireChip(...)		X								
AcquireChipWithDelay(...)		X								
Calibrate(...)		X								
CheckSum(...)		X								
CheckSum1(...)		X								
ClosePort()							X			
DAP_Acquire()			X	X	X					
DAP_AcquireChip()			X	X	X					
DAP_GetJtagID()			X	X	X					
DAP_PollIO()			X	X	X					
DAP_PollIO1()			X	X	X					
DAP_ReadIO()			X	X	X					
DAP_ReleaseChip()			X	X	X					
DAP_WriteIO()			X	X	X					
EraseAll(...)		X								
EraseBlock(...)		X								
EraseBlock1(...)		X								
GetAcquireMode(...)								X		
GetDeviceInfo(...)	X									
GetDeviceInfo1(...)	X									
GetDeviceList(...)	X									
GetDeviceMatchedList(...)	X									
GetFamilyInfo(...)	X									
GetFamilyList(...)	X									
GetFlashCharacteris-tics(...)		X								
GetPorts(...)							X			
GetPower(...)								X		

Table 1-1. PSoC Programmer APIs

APIs	Chip APIs						Port APIs	Programmer APIs	Protocols	Misc
	Common	PSoC 1	PSoC 3/5	PSoC 4	FM0	PSoC 6				
GetPower1(...)								X		
GetPower2(...)								X		
GetPowerVoltage(...)								X		
GetProgrammerCapabilities(...)								X		
GetProgrammerCapsByName(...)								X		
GetProgrammerVersion(...)								X		
GetSiliconId(...)		X								
GetRowsPerArrayInFlash(...)				X	X					
Hex_ReadChipProtection()	X									
HEX_ReadChecksum()	X									
HEX_ReadConfig()	X									
HEX_ReadData()	X									
HEX_ReadEEPROM()	X									
HEX_ReadExtra()	X									
HEX_ReadFile()	X									
HEX_ReadImageSizes()	X									
HEX_ReadJtagID()	X									
HEX_ReadNvlCustom()	X									
HEX_ReadNvlWo()	X									
HEX_ReadProtection()	X									
HEX_WriteChipProtection()	X									
HEX_WriteData()	X									
HEX_WriteFile()	X									
HEX_WriteProtection(...)	X									
HEX_WriteEEPROM(...)	X									
HEX_GetDataInRange(...)	X									
HEX_GetDataSizeInRange(...)	X									
I2C_DataTransfer(...)									X	
I2C_GetDeviceList(...)									X	
I2C_GetSpeed(...)									X	
I2C_ReadData(...)									X	
I2C_ReadDataFromReg(...)									X	
I2C_ResetBus(...)									X	
I2C_SendData(...)									X	
I2C_SetSpeed(...)									X	
IsPortOpen(...)							X			
JTAG_EnumerateDevices(...)									X	
JTAG_SetChainConfig(...)									X	
JTAG_SetIR(...)									X	
JTAG_ShiftDR(...)									X	

Table 1-1. PSoC Programmer APIs

APIs	Chip APIs						Port APIs	Programmer APIs	Protocols	Misc
	Common	PSoC 1	PSoC 3/5	PSoC 4	FM0	PSoC 6				
JTAG_ShiftIR(...)									X	
JTAGIO(...)									X	
JTAGIOR(...)									X	
JTAGIOW(...)									X	
OpenPort(...)							X			
PowerOff(...)								X		
PowerOn(...)								X		
ProgrammerLedState(...)								X		
Protect(...)		X								
ProtectAll(...)		X								
PSoC 3_CheckSum(...)			X							
PSoC 3_DebugPortConfig(...)			X							
PSoC 3_EraseAll()			X							
PSoC 3_GetEccStatus(...)			X							
PSoC 3_ProgramRow(...)			X							
PSoC 3_ProgramRowFromHex(...)			X							
PSoC 3_ProtectAll(...)			X							
PSoC 3_ProtectArray(...)			X							
PSoC 3_ReadNvlArray(...)			X							
PSoC 3_ReadProtection(...)			X							
PSoC 3_ReadRow(...)			X							
PSoC 3_VerifyProtect(...)			X							
PSoC 3_VerifyRowFromHex(...)			X							
PSoC 3_WriteNvlArray(...)			X							
PSoC 3_WriteRow(...)			X							
PSoC3_EraseRow (…)			X							
PSoC3_EraseSector (…)			X							
PSoC3_GetEepromArrayInfo (…)			X							
PSoC3_GetFlashArrayInfo(...)			X							
PSoC3_GetSonosArrays(...)			X							
PSoC3_SetJtagChainConfig(....)			X							
PSoC4_CheckSum(...)				X						
PSoC4_EraseAll(...)				X						
PSoC4_GetFlashInfo(...)				X						
PSoC4_GetSiliconID(...)				X						
PSoC4_ProgramRow(...)				X						
PSoC4_ProgramRowFromHex(...)				X						
PSoC4_ProtectAll(...)				X						
PSoC4_ReadProtection(...)				X						

Table 1-1. PSoC Programmer APIs

APIs	Chip APIs						Port APIs	Programmer APIs	Protocols	Misc
	Common	PSoC 1	PSoC 3/5	PSoC 4	FM0	PSoC 6				
PSoC4_ReadRow(...)				X						
PSoC4_VerifyProtect(...)				X						
PSoC4_VerifyRowFrom-Hex(...)				X						
PSoC4_WriteProtec-tion(...)				X						
PSoC4_WriteRow(...)				X						
FM0_VerifyRowFromHex-OneRead (...)					X					
FM0_ProgramRow (...)					X					
FM0_EraseAll (...)					X					
FM0_CheckSum (...)					X					
FM0_ReadRow (...)					X					
FM0_GetFlashInfo (...)					X					
FM0_ProgramRowFrom-Hex (...)					X					
FM0_VerifyRowFromHex (...)					X					
FM0_GetSiliconID (...)					X					
FM0_EraseSector (...)					X					
FL_Init (...)					X					
FL_UnInit (...)					X					
FM0_FL_ProgramRow-FromHex (...)					X					
FL_ProgramPage (...)					X					
FL_ProgramSector (...)					X					
FL_LoadElf (...)						X				
FL_ExecAPI (...)						X				
FL_IsApiExists (...)						X				
FL_SetRamForAlgo-rithms (...)						X				
FL_PrepareTarget (...)						X				
FL_ExecCmsisApiInit (...)						X				
FL_ExecCmsisApiUnInit (...)						X				
FL_ExecCmsisApiErase-Sector (...)						X				
FL_ExecCmsisApiPro-gramPage (...)						X				
FL_ExecCmsisApiVerify (...)						X				
FL_ReleaseTarget (...)						X				
PSoC6_WriteRow(...)						X				
PSoC6_ProgramRow (...)						X				
PSoC6_EraseAll (...)						X				
PSoC6_CheckSum (...)						X				
PSoC6_WriteProtection (...)						X				
PSoC6_ReadRow (...)						X				
PSoC6_ReadProtection (...)						X				

Table 1-1. PSoC Programmer APIs

APIs	Chip APIs						Port APIs	Programmer APIs	Protocols	Misc
	Common	PSoC 1	PSoC 3/5	PSoC 4	FM0	PSoC 6				
PSoC6_ProtectAll(...)						X				
PSoC6_GetFlashInfo(...)						X				
PSoC6_ProgramRow-FromHex(...)						X				
PSoC6_VerifyRowFrom-Hex(...)						X				
PSoC6_GetSiliconID(...)						X				
PSoC6_WriteRowFrom-Hex(...)						X				
HEX_GetRowAddress(...)						X				
HEX_GetRowsCount(...)						X				
DAP_JTAGtoSWD(...)	X									
DAP_SWDtoJTAG(...)	X									
DAP_JTAGtoDS(...)	X									
DAP_SWDtoDS(...)	X									
DAP_DStoSWD(...)	X									
DAP_DStoJTAG(...)	X									
ReadBlock(...)		X								
ReadBlock1(...)		X								
ReadIO(....)		X								
ReadProtection(....)		X								
ReadRAM(...)		X								
ReleaseChip()		X								
SetAcquireMode(...)								X		
SetAutoReset()								X		
SetBank(...)		X								
SetChipType(...)	X									
SetPowerVoltage(...)								X		
SetProtocol(...)								X		
SetProtocolClock(...)								X		
SetProtocolConnector(...)								X		
SPI_ConfigureBus(...)									X	
SPI_DataTransfer(...)									X	
SPI_GetSupported-Freq(...)									X	
swdior(...)								X		
swdior_raw(..)								X		
swdiow(...)								X		
swdiow_raw(...)								X		
swd_LineReset()								X		
SWV_ReadData(...)									X	
SWV_Setup(...)									X	
TableRead(...)		X								
TestClock(...)		X								
ToggleReset(...)									X	
UpdateProgrammer(...)								X		
UpdateProgrammer1(...)								X		

Table 1-1. PSoC Programmer APIs

APIs	Chip APIs						Port APIs	Programmer APIs	Protocols	Misc
	Common	PSoC 1	PSoC 3/5	PSoC 4	FM0	PSoC 6				
USB2IIC_AsyncMode(...)									X	
USB2IIC_AsyncMode1(...)									X	
USB2IIC_DataTransfer(...)									X	
USB2IIC_Received-Data(...)									X	
USB2IIC_SendData(...)									X	
VerifyBlock(...)		X								
VerifyBlock1(...)		X								
VerifyBlockFromHex(...)		X								
VerifyProtect(...)		X								
Version(...)										X
WriteBlock(...)		X								
WriteBlock1(...)		X								
WriteBlockFromHex(...)		X								
WriteBlockFromHex1(...)		X								
WriteIO(...)		X								
WriteRAM(...)		X								

1.5 Obsolete and Equivalent New APIs

Table 1-2. Obsolete and New APIs

Obsolete API	New API
PSoC3_Acquire(...)	DAP_Acquire(...)
PSoC3_AcquireChip(...)	DAP_AcquireChip(...)
PSoC3_GetJtagID(...)	DAP_GetJtagID(...)
PSoC3_PollIO(...)	DAP_PollIO(...)
PSoC3_PollIO1(...)	DAP_PollIO1(...)
PSoC3_ReadHexConfig(...)	HEX_ReadConfig(...)
PSoC3_ReadHexEEPROM(..)	HEX_ReadEEPROM(...)
PSoC3_ReadHexExtra(...)	HEX_ReadExtra(...)
PSoC3_ReadHexImageSize(...)	HEX_ReadImageSizes(...)
PSoC3_ReadHexJtagID(...)	HEX_ReadJtagID(...)
PSoC3_ReadHexNvlCustom(...)	HEX_ReadNvlCustom(...)
PSoC3_ReadHexNvlWo(...)	HEX_ReadNvlWo(...)
PSoC3_ReadIO(...)	DAP_ReadIO(...)
PSoC3_ReleaseChip(...)	DAP_ReleaseChip(...)
PSoC3_SetJtagChainConfig(...)	JTAG_SetChainConfig(...)
PSoC3_WriteIO(...)	DAP_WriteIO(...)
ReadHexChecksum(...)	HEX_ReadChecksum(...)
ReadHexData(...)	HEX_ReadData(...)
ReadHexFile(...)	HEX_ReadFile(...)
ReadHexProtection(...)	HEX_ReadProtection(...)
WriteHexData(...)	HEX_WriteData(...)
WriteHexFile(...)	HEX_WriteFile(...)
WriteHexProtection(...)	HEX_WriteProtection(...)

1.6 Documentation Conventions

Table 1-3. Document Conventions for Guides

Convention	Usage
`Courier New`	Displays file locations, user entered text, and source code: `C:\ ...cd\icc\`
Italics	Displays file names and reference documentation: Read about the *sourcefile.hex* file in the *PSoC Designer User Guide*.
[Bracketed, Bold]	Displays keyboard commands in procedures: **[Enter]** or **[Ctrl]** **[C]**
File > Open	Represents menu paths: File > Open > New Project
Bold	Displays commands, menu paths, and icon names in procedures: Click the **File** icon and then click **Open**.
Times New Roman	Displays an equation: 2 + 2 = 4
Text in gray boxes	Describes Cautions or unique functionality of the product.

2. Command Descriptions

Acquire(string OUT strError)

Calls AcquireChip, then GetSiliconId.

C#:
```
int Acquire(out string strError);
   string strError;
   int hr = pp.Acquire(out strError);
   if (SUCCEEDED(hr)) { //Acquired successfully|
    MessageBox.Show("Device Acquired Successfully!");
    //...continue to work with PSoC in Test Mode
   }
   else //Acquire Failed
   {
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
         }
```

AcquireChip(string OUT strError)

Calls AcquireChipWithDelay using a delay from 1 to 20 ms until acquired.

C#:
```
int AcquireChip(out string strError);

string strError;
int hr = pp.AcquireChip(out strError);
if (SUCCEEDED(hr))
{ //Acquired successfully|
    MessageBox.Show("Device Acquired Successfully!");
    //...continue to work with PSoC in Test Mode
}
else //Acquire Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

AcquireChipWithDelay(IN delay, string OUT strError)

Acquires the chip after *delay*/2 ms after power on or reset. For example, if *delay*=2, the acquire sequence starts after 2/2 ms (1 ms).

C#:

```
int AcquireChipWithDelay(int delay, out string strError);

string strError;
int hr = pp.AcquireChipWithDelay(10, out strError);
if (SUCCEEDED(hr))
{ //Acquired successfully|
    //MessageBox.Show("Device Acquired Successfully!");
    //...continue to work with PSoC in Test Mode
}
else //Acquire Failed
{
    MessageBox.Show(strError, "xxx_Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

Calibrate(IN index, string OUT strError)

Calls the *Calibrate0* or *Calibrate1* supervisory operation (see the Technical Reference Manual) depending on the parameter used. If index = 0 then *Calibrate 0* is called; if index = 1 then *Calibrate1* is called.

C#:

```
int Calibrate(int index, out string strError);

string strError;
int hr = pp.Calibrate(0, out strError);
if (SUCCEEDED(hr))
{ //Calibrated successfully|
    MessageBox.Show("Device Calibrated Successfully!");
    //...continue to work with PSoC in Test Mode
}
else //Calibration Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

CheckSum(IN blocks, OUT result, string OUT strError)

Calls the *checksum* supervisory operation (see the Technical Reference Manual) and returns the checksum of the first blocks. If this parameter is 0, then it calculates the checksum of the whole Flash. The *block*s parameter is in the range [0..TotalBlocksCount–1]. It does not depend on the current bank; it operates on the entire Flash. To calculate the checksum correctly, call the GetSiliconId() function before you call CheckSum(). GetSiliconID() sets the correct Flash parameters of the acquired device.

C#:

```
int CheckSum(int blockID, out ushort checksum,
                  out string strError);
```

```
ushort cs;
string strError;
int hr = pp.checksum(0,out cs, out strError);
if (SUCCEEDED(hr)) {
    MessageBox.Show("Total Flash Checksum: "+cs.ToString("X4"));
}
else //Checksum Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

CheckSum1(IN blocks, IN bank, OUT result, string OUT strError)

Calculates the checksum of the first *blocks* in the given *bank*. The *blocks* parameter is in the range [1..blocksPerBank], but if blocksPerBank is 256, then this parameter must be 0 to calculate the checksum of the whole bank. If the number of blocks is other than 256, then *blocks* = 0 does not calculate the checksum of the whole bank.

C#:

```
int CheckSum1(int blockID, int bank, out ushort checksum,
                    out string strError);
ushort cs;
string strError;
int hr = pp.CheckSum1(128, 0, out cs, out strError);
if (SUCCEEDED(hr))
{
    MessageBox.Show("Checksum of Bank 1: " + cs.ToString("X4"));
}
else //Checksum Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

ClosePort(string OUT strError)

Closes the port opened with the OpenPort() function. If the programmer is physically disconnected, the port is automatically disconnected by the COM object, so there is no need to disconnect it in the disconnect event handler in the client application. After the port is closed, you cannot use Chip/Programmer APIs until the port is opened again.

C#:

```
int ClosePort(out string strError);

ushort cs;
string strError;
int hr = pp.ClosePort(out strError);
if (SUCCEEDED(hr)) ;//Port Closed Successfully
else //Operation Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

DAP_Acquire(string OUT strError)

Calls DAP_AcquireChip() and then DAP_GetJtagID(), which sets up the software to work with the acquired device. Before you call this function, you must be sure that all other modes are set correctly. The example below demonstrates a possible sequence of calls for acquiring the chip:

C#: `int DAP_Acquire(out string strError);`

```
string strError;
     pp.SetPowerVoltage("2.5", out strError);
     pp.PowerOn(out strError);
     pp.SetAcquireMode("Reset", out strError);
     pp.SetProtocol(enumInterfaces.SWD, out strError);
     pp.SetProtocolConnector(1, out strError); //10-pin
     pp.SetProtocolClock(enumFrequencies.FREQ_03_0, out strError);

int hr = pp.DAP_Acquire(out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("PSoC device acquired successfully!");
  else
     MessageBox.Show("Failed: "+strError);
```

DAP_AcquireChip(string OUT strError)

Tries to acquire the PSoC device using SWD protocol. Before you call this function, you must be sure that all other modes are set correctly. The example below demonstrates a possible sequence of calls to acquire the chip using MiniProg3:

C#: `int DAP_AcquireChip(out string strError);`

```
string strError;
     pp.SetPowerVoltage("2.5", out strError);
     pp.PowerOn(out strError);
     pp.SetAcquireMode("Reset", out strError);
     pp.SetProtocol(enumInterfaces.SWD, out strError);
     pp.SetProtocolConnector(1, out strError); //10-pin
     pp.SetProtocolClock(enumFrequencies.FREQ_03_0, out strError);

int hr = pp.DAP_AcquireChip(out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("PSoC device acquired successfully!");
    else
     MessageBox.Show("Failed: "+strError);
```

DAP_GetJtagID(nvector OUT jtagID, string OUT strError)

Reads the JtagID of the acquired PSoC device and returns a 4-byte array. This method is used only for devices that work using the debug access port (DAP). The function is used for PSoC 3, PSoC 4, and PSoC 5; however, for PSoC 4, it returns ID - 0x0BB11477, which is the ID of ARM's CM0 core.

C#: . `int DAP_GetJtagID(out object jtagID, out string strError);`

```
string strError;
```

```
object jtagID;
int hr = pp.DAP_GetJtagID(out jtagID, out strError);
if (SUCCEEDED(hr)) {
   string msg = "JTAG ID: ";
   byte[] id = jtagID as byte[];
   for (int i = 0; i < id.Length; i++) {
     msg += id[i].ToString("X2") + " ";
   }
   MessageBox.Show(msg);
}
else
   MessageBox.Show(strError);
```

DAP_PollIO(IN baseAddr, IN expectedValue, IN timeout, string OUT strError)

Polls the I/O register of the acquired device until it is assigned to *expectedData* or until the timeout period elapses.

C#:

```
int DAP_PollIO(int baseAddr, int expectedData, int timeOut,
out string strError);

string strError;
int baseAddr = 0x00000005;
int expectedData = 0x12;
int timeOut = 2000;
int hr = pp.DAP_PollIO(baseAddr, expectedData, timeOut,
out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Register "+baseAddr.ToString("X8")+
      " possesses the value "+expectedData.ToString("X2"));
 else
   MessageBox.Show(strError);
```

DAP_PollIO1(IN baseAddr, IN expectedMask, IN timeout, string OUT strError)

Polls the I/O register of the acquired device until it is masked by *expectedMask* or until the timeout period elapses.

C#:

```
int DAP_PollIO1(int baseAddr, int expectedMask, int timeOut,
out string strError);

string strError;
int baseAddr = 0x00000005;
int expectedMask = 0x80;
int timeOut = 2000;
int hr = pp.DAP_PollIO1(baseAddr, expectedMask, timeOut,
out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Register " + baseAddr.ToString("X8") +
    " has MSB set to 1 ");
```

```
else
   MessageBox.Show(strError);
```

DAP_ReadIO(IN address, OUT data, string OUT strError)

Reads data located at *address*. The *data* parameter is 4-bytes long, but actually can be 1, 2, or 4 bytes depending on the last call of PSoC3_DebugPortConfig(). By default, the chip is acquired in 1-byte transfer mode.

C#:
```
int DAP_ReadIO(int baseAddr, out int data,
out string strError);

string strError;
int address = 0x05, data;
int hr = pp.DAP_ReadIO(address, out data, out strError);
if (SUCCEEDED(hr))
   {
   string msg = "Read data "+data.ToString("X8")+
   " from address "+address.ToString("X8");
   MessageBox.Show(msg);
   }
else
   MessageBox.Show(strError);
```

DAP_ReleaseChip(string OUT strError)

Releases the acquired chip, and powers off the chip if PowerCycle was used to acquire the device. If Reset (or PowerDetect) mode is used, the chip is reset using the XRES pin (if present). The SetAutoReset() function can be used to disable reset if necessary.

C#:
```
int DAP_ReleaseChip(out string strError);
string strError;

int hr = pp.DAP_ReleaseChip(out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Chip released successfully!");
else
   MessageBox.Show("Failed to release chip: "+strError);
```

DAP_ReadRaw (unsigned char IN dap_addr, OUT data32, unsigned char OUT status, string OUT strError)

Reads 32 bit data from DAP's register. This API works in SWD and JTAG modes. Parameters in this API is the same as in DAP_WriteRaw() API

C#:
```
int DAP_ReadRaw(byte dap_addr, out int data, out byte status, out
string strError);

string strError;
byte status;
int addr = 0x20000004;
```

```
int data = 0x23242526;
int hr = pp.DAP_WriteRaw(0x05, addr, out status, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Status = " + status);
else
   MessageBox.Show(strError);
hr = pp.DAP_ReadRaw(0x07, out data, out status, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Status = " + status);
else
   MessageBox.Show(strError);
hr = pp.DAP_ReadRaw(0x07, out data, out status, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Status = " + status);
else
   MessageBox.Show(strError);
```

DAP_WriteIO(IN address, IN data, string OUT strError)

Writes *data* to the addressed cell. The *data* parameter is 4-bytes long, but actually can be 1, 2, or 4 bytes depending on the last call of PSoC3_DebugPortConfig(). By default, the chip is acquired in 1-byte transfer mode.

C#:

```
int DAP_WriteIO(int baseAddr, int data,
out string strError);

string strError;
int address = 0x00000005;
int data = 0x78;
int hr = pp.DAP_WriteIO(address, data, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Data "+data.ToString("X2")+
        " written to address "+address.ToString("X8"));
 else
   MessageBox.Show(strError);
```

DAP_WriteRaw (unsigned char IN dap_addr, IN data32, unsigned char OUT status, string OUT strError)

Writes 32 bit data in DAP's register (Debug Access Port). This API works in SWD and JTAG modes.

DAP is available in ARM's based silicon (for example: PSoC 4, PSoC 3 / PSoC 5). The "dap_addr" parameter specifies DAP register (3 LSB bits):

- bit 2 (mask 0x04) - defines APACC (1) or DPACC (0) access.
- bits 1-0 (mask 0x03) - defines register in selected access port (APACC or DPACC).

The returned parameter “status” contains 3-bit status of previous DAP transaction. This status is different for SWD and JTAG protocols. Only three LSB bits should be considered.

Table 2-1. Status for SWD Protocol

SWD	
001	ACK
010	WAIT
100	FAULT
OTHER	NACK

Table 2-2. Status for JTAG Protocol

JTAG	
010	ACK
001	WAIT
100	FAULT
OTHER	NACK

For more information about status, refer to ARM’s specification.

C#:
```
int DAP_WriteRaw(byte dap_addr, int data, out byte status, out string strError);

string strError;
byte status;
int addr = 0x20000004;
int data = 0x23242526;
int hr = pp.DAP_WriteRaw(0x05, addr, out status, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Status = " + status);
else
   MessageBox.Show(strError);
hr = pp.DAP_WriteRaw(0x07, data, out status, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Status = " + status);
else
   MessageBox.Show(strError);
```

EraseAll(string OUT strError)

Calls the *EraseAll* supervisory operation (see the Technical Reference Manual). Erases the entire chip and protection blocks.

C#:
```
int EraseAll(out string strError);

string strError;
int hr = pp.EraseAll(out strError);
if (SUCCEEDED(hr)) ;//Erased Successfully
else //Operation Failed
```

```
{
MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

EraseBlock(IN blockID, OUT sscResult, string OUT strError)

Calls the *EraseBlock* supervisory operation. Erases one block with *blockID* in the current bank. The parameter *blockID* is in the range [0..blocksPerBank-1].

C#:

```
int EraseBlock(int blockID, out int sscResult,
                    out string strError);

int sscResult;
string strError;
int hr = pp.EraseBlock(1,out sscResult, out strError);
if (SUCCEEDED(hr)) ;//Erased Successfully
else //Operation Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

EraseBlock1(IN blockID, IN bank, OUT sscResult, string OUT strError)

Calls the *EraseBlock* supervisory operation. Erases *blockID* in the given bank. The parameter *blockID* is in the range [0..blocksPerBank-1].

C#:

```
int EraseBlock1(int blockID, int bank, out int sscResult,
                     out string strError);

int sscResult;
string strError;
int hr = pp.EraseBlock1(1, 0, out sscResult, out strError);
if (SUCCEEDED(hr)) ;//Erased Successfully
else //Operation Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
  MessageBoxIcon.Error);
}
```

GetAcquireMode(string OUT mode, string OUT strError)

Returns the current value of AcquireMode. It is one of the following values:

- Power
- PowerDetect
- Reset

C#:

```
int GetAcquireMode(out string mode, out string strError);

string acquireMode;
```

```
string strError;
int hr = pp.GetAcquireMode(out acquireMode, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Current Acquire Mode is: " + acquireMode);
else //Operation Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

GetDeviceInfo(string IN devName, string OUT family, OUT familyCode, OUT pins, OUT flashSize, OUT acquireModes, svector OUT siliconIDs, string OUT strError)

Returns information about the PSoC device given by *devName*. This is a name from the list returned by the GetDeviceList() function. Device information includes:

- *family* – family name to which the PSoC device belongs. It is one of the list elements returned by the GetFamilyList() function.
- *familyCode* – A Cypress internal code for the family of devices. A family of devices is grouped together because of similarities in their arrangement of internal resources. Because new families of parts are frequently added, the following list is only an example rather than a complete list:

Table 2-3. Example Family Codes

Code	Device Families
0x00	CY8C25122, CY8C26xx3
0x01	CY7C603xx, CY7C604xx, CY7C64215, CY7C643xx, CY8C20x24, CY8C20x34, CY8C20x66, CY8C24x94, CY8C27x43, CY8C27x43-*XI, CY8C27143, CY8C27143-*XI, CY8CLED02, CY8CLED04, CY8-CLED08, CYONS2xxx
0x02	21x23, 21x34, 22x13, 23x33A, 24x23, 24x23A, 24x33A, CYRF69100, CYRF69200, CYWUSB6900
0x03	CY8C29x66, CY8CLED16
0x04	CY7C601xx, CY7C602xx, CY7C633xx, CY7C638xx, CY7C639xx

- *pins* – number of pins on the device
- *flashSize* – the device Flash size (in bytes);
- *acquireModes* – supported modes of device acquisition. This value is a union of two fields:
  - 0x01 – Reset (if XRES pin present)
  - 0x02 – Power (available for all chips)
- *siliconIDs* – an array of siliconIDs compatible with this device; for example, “00 09 21 10”. These are the hex numbers separated by spaces. The order of numbers is: sID_High, sID_Low, RevID, FamID.

C#:
```
int GetDeviceInfo(string devName, out string family,
out int familyCode, out int pins,
out int flashSize, out int acquireModes,
out object siliconIDs, out string strError)
```

```
object sIDs;
string family;
int pins, flashSize, acquireModes, familyCode;
string deviceName = "CY8C27543";
string strError;
int hr = pp.GetDeviceInfo(deviceName,out family, out familyCode,
   out pins, out flashSize,
out acquireModes, out sIDs, out strError);
if (SUCCEEDED(hr))
{
    string info = "-->"+ deviceName + " Information:";
    info += "\r\nFamily: "+family;
    info += "\r\nPins  : "+pins;
    info += "\r\nFlash : "+flashSize;
    info += "\r\nAcquire Modes: "+acquireModes;
    if ((acquireModes & 0x02) != 0) info += " POWER";
    if ((acquireModes & 0x01) != 0) info += " RESET";
    info += "\r\nSilicon IDs: ";
    string[] sIDsList = sIDs as string[];
    for (int i = sIDsList.GetLowerBound(0);
             i < sIDsList.GetLength(0); i++)
    {
        info += "\r\n\t"+sIDsList[i];
    }
    MessageBox.Show(info);
}
else
    MessageBox.Show(strError);
```

GetDeviceInfo1(string IN devName, CDeviceInfo OUT deviceInfo, OUT familyCode, OUT pins, OUT flashSize, OUT acquireModes, svector OUT siliconIDs, string OUT strError)

Returns information about the PSoC device given by *devName*. This is the name from the list returned by the GetDeviceList() function. This function was added because the structure of the database was changed during a new version of COM object development. The information returned by this function is identical to GetDeviceInfo() except for the voltage field. Device information is presented by the structure of the following view:

```
public struct CDeviceInfo
{
public string sFamilyName;
public int iFamilyCode;
      public int iPins;
public int iFlashSize;
public int iAcquireModes;
     public string sVoltage;
           public object siliconIDs;
}
```

sFamilyName – family to which the PSoC device belongs. This is one of the list elements returned by the GetFamilyList() function.

iFamilyCode – special code of the family. Values are described in the GetDeviceInfo() function.

iPins – quantity of IC's pins.

iFlashSize – amount of IC's flash (in bytes).

iAcquireModes – information about supported modes of device acquisition. This value is a union of two fields:

- **0x01** – Reset (if XRES pin present)
- **0x02** – Power (available for all chips)

sVoltage – the minimum permissible voltage necessary to program the device.

siliconIDs – array of siliconIDs compatible with this device. Example of array element: "00 09 21 10". These are the hex numbers separated by spaces. The order of numbers is: sID_High, sID_Low, RevID, FamID.

C#:
```
int GetDeviceInfo1(string devNameDisp,
out CDeviceInfo deviceInfo, out string strError);
string strError;
            CDeviceInfo deviceInfo;
            string deviceName = "CY8C3844PVI-001";
int hr = pp.GetDeviceInfo1(deviceName, out deviceInfo,out strError);
if (SUCCEEDED(hr))
      {
        string info = "-->" + deviceName + " Information:";
        info += "\r\nFamily: " + deviceInfo.sFamilyName;
        info += "\r\nPins  : " + deviceInfo.iPins;
        info += "\r\nFlash : " + deviceInfo.iFlashSize;
        info += "\r\nAcquire Modes: " + deviceInfo.iAcquireModes;
      if ((deviceInfo.iAcquireModes & 0x02) != 0) info += " POWER";
       if ((deviceInfo.iAcquireModes & 0x01) != 0) info += " RESET";
                info += "\r\nVoltage: " + deviceInfo.sVoltage;
                info += "\r\nSilicon IDs: ";
                string[] sIDsList = deviceInfo.siliconIDs as string[];
                for (int i = sIDsList.GetLowerBound(0);
                        i < sIDsList.GetLength(0); i++)
                {
                    info += "\r\n\t" + sIDsList[i];
                }
                MessageBox.Show(info);
            }
            else
                MessageBox.Show(strError);
```

GetDeviceList(string IN family, svector OUT deviceList, string OUT strError)

Returns the list of devices that belong to *family*. The result is a 2-dimensional array where the first column is the display name and the second column is the full name.

C#:
```
int GetDeviceList(string family, out string deviceList , out string
strError);
object devs;
```

```
string strError;
//int hr = pp.GetDeviceList("60100",out devs);
int hr = pp.GetDeviceList("CY8C21020", out devs, out strError);
if (SUCCEEDED(hr))
{
   Array deviceList = devs as Array;
   string s = "--> Device List\r\n";
   for (int i = deviceList.GetLowerBound(1);
     i < deviceList.GetLength(1); i++)
     {
    s += deviceList.GetValue(0,i) + "  "+deviceList.GetValue(1,i)+ "\r\n";
     }
    MessageBox.Show(s);
}
else
    MessageBox.Show(strError);
```

GetDeviceListBySiliconID(nvector IN siliconID, IN ignoreRevisionID,svector OUT deviceList, string OUT strError)

Returns the list of devices compatible with siliconID.

Parameters:

siliconID - four-byte parameter, the format is given in a GetSiliconID() function description.

ignoreRevisionID - searching mode:

- 0x00 — revision ID is taken into account during search;
- 0x01 — search ignores chip Revision ID

C#:

```
object deviceList;
byte[] siliconID = new byte[] {0x0E, 0x34, 0x13, 0x9E};
byte ignoreRevisionID = 1; // Ignore Rev ID field.
string strError;
int hr = pp.GetDeviceListBySiliconID(siliconID as object,
ignoreRevisionID, out deviceList, out strError);
if (SUCCEEDED(hr))
{
        string[] devs = deviceList as string[];
        string s = "Matched Device List:\r\n";
        for (int i = devs.GetLowerBound(0); i < devs.GetLength(0); i++)
        {
           s += devs[i] + "\r\n";
        }
        Console.WriteLine(s);
}
else
{
        Console.WriteLine("Error: " + strError);
}
```

GetDeviceMatchedList(nvector IN siliconID, svector OUT deviceList, string OUT strError)

Returns the list of devices compatible with *siliconID*. The format of the *siliconID* parameter is described in a GetSiliconID() function.

C#:
```
int GetDeviceMatchedList(object siliconID,
 out object deviceList, out string strError);

object devs;
byte[] siliconID = new byte[] {0x00, 0x0C, 0x21, 0x10};
string strError;
int hr = pp.GetDeviceMatchedList(siliconID, out devs, out strError);
if (SUCCEEDED(hr))
{
    string[] deviceList = devs as string[];
    string s = "--> Matched Device List\r\n";
    for (int i = deviceList.GetLowerBound(0);
 i < deviceList.GetLength(0); i++)
    {
        s += deviceList[i] + "\r\n";
    }
    MessageBox.Show(s);
}
else
 MessageBox.Show(strError);
```

GetFamilyInfo(string familyName, CFamilyInfo OUT familyInfo, string OUT strError)

Returns information about a given family in the structure of the following view:

```
public struct CFamilyInfo
      {
        public string sFamilyName;
public string sDisplayName;
public int iProtocol;
public int iFamilyCode;
public int iBytesPerBlock;
}
```

sFamilyName – string representation of the family returned by the GetFamilyList() function. Used internally.

sDisplayName – string name (used in the GUI) of the family returned by GetFamilyList().

iProtocol – identifies the protocol that is used to program this device.
The possible values are: 2 – ISSP, 3 – SWD/JTAG, 4 – I^2C, SWD - 5.

iFamilyCode – special family code used internally by Cypress Software. It is described in the GetDeviceInfo() function on page 29.

iBytesPerBlock – the size of one flash block or row in bytes.

C#:
```
int GetFamilyInfo(string familyName,  out CFamilyInfo familyInfo, out
string strError);
string strError;
       CFamilyInfo familyInfo;
int hr = pp.GetFamilyInfo("CY8C38xx", out familyInfo, out strError);
if (SUCCEEDED(hr))
        {
           string msg;
           msg =  "Family Name : " + familyInfo.sFamilyName+"\r\n";
           msg += "Display Name: " + familyInfo.sDisplayName+"\r\n";
           msg += "Family Code : " + familyInfo.iFamilyCode + "\r\n";
           msg += "Protocols   : " + familyInfo.iProtocol +" --> ";
           switch (familyInfo.iProtocol) {
                   case 2: msg += "ISSP";  break;
                   case 3: msg += "SWD/JTAG";  break;
                   case 4: msg += "I2C" ;break;
                   default: msg += "unknown"; break;
               }
                 msg += "\r\n";
                 msg += "BytePerBlock: " + familyInfo.iBytesPerBlock;
                 MessageBox.Show(msg);
            }
              else
                  MessageBox.Show(strError);
```

GetFamilyList(svector OUT familyList, string OUT strError)

Returns the list of families supported by PSoC Programmer.

C#:
```
int GetFamilyList(out string familyList, out string strError);

object fams;
string strError;
pp.GetFamilyList(out fams, out strError);
Array familyList = fams as Array;
string s = "--> Family List\r\n";
for (int i = familyList.GetLowerBound(1);
   i < familyList.GetLength(1); i++)
{
    s+= familyList.GetValue(0,i)+"  "+familyList.GetValue(1,i)+"\r\n";
}
    MessageBox.Show(s);
```

GetFlashCharacteristics(OUT blockSize, OUT banks, OUT blocksPerBank, string OUT strError)

Returns the Flash characteristics of the acquired device. You must identify the PSoC device connected to the programmer before calling this function. Use the GetSiliconId() function, or use the Acquire() function to set the in-socket PSoC device for this function.

C#:
```
int GetFlashCharacteristics(out int blockSize, out int banks,
                            out int blockPerBank, out string strError);
```

```
string strError;
int blockSize, banks, blocksPerBank;
int hr = pp.GetFlashCharacteristics(out blockSize, out banks,
out blocksPerBank, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Flash Characteristics:"+
                    "\r\n blockSize    = "+blockSize+
                    "\r\n banks        = "+banks+
                    "\r\n blockPerBank = "+blocksPerBank);
else //Operation Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

GetPorts(svector OUT ports, string OUT strError)

Returns a list of ports (connected programmers).

C#:

```
int GetPorts(out object ports, out string strError);

object ports;
string strError;
int hr = pp.GetPorts(out ports, out strError);
if (SUCCEEDED(hr))
{
    string[] portsList = ports as string[];
    string strPorts = "-->Ports List: ";
    for (int i = portsList.GetLowerBound(0);
           i < portsList.GetLength(0); i++)
    {
        strPorts += "\r\n" + portsList[i];
    }
    MessageBox.Show(strPorts);
}
```

GetPower(OUT power, string OUT strError)

Returns the power status as read by the programmer. If the target device is not powered, the returned value is zero. If the target device is powered (either by the programmer or an external source) the returned value is the union of following fields:

POWER_DETECTED	0x01
POWER_SUPPLIED	0x02

The 0x01 value means that the programmer detected power (it can be set only if the programmer has read power ability). 0x02 means that the programmer supplies power to the device.

C#:

```
int GetPower(out int power, out string strError);
```

```
int power;
string strError;
int hr = pp.GetPower(out power, out strError);
if (SUCCEEDED(hr)) {
    string strPowerStatus = "Power Status: ";
    if ((power & 0x01) != 0)
    strPowerStatus += "\r\n"+"POWER_DETECTED";
    if ((power & 0x02) != 0)
    strPowerStatus += "\r\n"+"POWER_SUPPLIED";
    if (power == 0) strPowerStatus += "POWER NOT DETECTED";
    MessageBox.Show(strPowerStatus);
}
else //Operation Failed
{
    MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

GetPower1(OUT power, OUT voltage, string OUT strError)

This is an extension of the GetPower() function that returns voltage (mV) measured on the target board by the programmer. The *power* parameter is described in the GetPower() function. The *voltage* parameter is expressed in mV units and is meaningful only for programmers that have capabilities set to CAN_MEASURE_POWER (see GetProgrammerCapabilities() function).

C#:
```
int GetPower1(out int power, out int voltage,
    out string strError);
```

```
int power, voltage;
string strError;
int hr = pp.GetPower1(out power, out voltage, out strError);
       if (SUCCEEDED(hr)) {
          string strPowerStatus = "Power Status: ";
          if ((power & 0x01) != 0)
             strPowerStatus += "\r\n"+"POWER_DETECTED";
          if ((power & 0x02) != 0)
             strPowerStatus += "\r\n"+"POWER_SUPPLIED";
          if (power == 0) strPowerStatus += "POWER NOT DETECTED";
              strPowerStatus += "\r\nVoltage measured: "+voltage +" mV";
              MessageBox.Show(strPowerStatus);
            }
            else //Operation Failed
            {
                MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
              MessageBoxIcon.Error);
```

GetPower2(OUT power, OUT voltage, string OUT strError)

This is an extension of the GetPower() function that returns voltages (mV) measured on the target board by the programmer. The *power* parameter is described in the GetPower() function. The *voltage1* and *voltage2* parameters are expressed in mV units and are meaningful only for

programmers that have capabilities set to CAN_MEASURE_POWER and CAN_MEASURE_POWER_2 (see GetProgrammerCapabilities() function). For example, for MiniProg3, voltage1 is meaningful. For TrueTouch, both voltages are measured on VCOM and VAUX lines.

C#:
```
int GetPower2(out int power, out int voltage1,
out int voltage2, out string strError);

int power, voltage1, voltage2;
string strError;
int hr = pp.GetPower2(out power, out voltage1, out voltage2,
out strError);
if (SUCCEEDED(hr))
{
      string strPowerStatus = "Power Status: ";
      if ((power & 0x01) != 0)
      strPowerStatus += "\r\n" + "POWER_DETECTED";
      if ((power & 0x02) != 0)
      strPowerStatus += "\r\n" + "POWER_SUPPLIED";
      if (power == 0) strPowerStatus += "POWER NOT DETECTED";
      strPowerStatus += "\r\nVCOM: " + voltage1 + " mV" + "\r\nVAUX: " +
      voltage2 + " mV";
      MessageBox.Show(strPowerStatus);
}
   else //Operation Failed
{
      MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
      MessageBoxIcon.Error);
}
```

GetPowerVoltage(string OUT voltage, string OUT strError)

Returns the currently selected internal voltage source of the programmer. (This does not mean that power is applied to the target board.) The possible string values are: 5.0, 3.3, 2.5, and 1.8.

C#:
```
int GetPowerVoltage(out string voltage, out string strError);

   string voltage, strError;
   int hr = pp.GetPowerVoltage(out voltage, out strError);
            if (SUCCEEDED(hr))
                MessageBox.Show("Currently set power source: "+voltage);
            else
               MessageBox.Show(strError);
```

GetProgrammerCapabilities(nvector OUT capabilities, string OUT strError)

Returns a vector of 6 bytes, which describes the capabilities of the programmer connected to the opened port. Each byte of array is described below:

- Capabilities[0] – identifies the set of allowed acquire modes for the current programmer. This characteristic can be a union of the following flags:

CAN_RESET_ACQUIRE	0x01
CAN_POWER_CYCLE_ACQUIRE	0x02
CAN_POWER_DETECT_ACQUIRE	0x04

- Capabilities[1] – represents the power abilities of the programmer. It is a union of following flags:

CAN_POWER_DEVICE	0x01
CAN_READ_POWER	0x02
CAN_MEASURE_POWER	0x04
CAN_MEASURE_POWER_2	0x10

- Capabilities[2] – this value is 0 if the programmer firmware cannot be updated.
- Capabilities[3] – identifies a set of PSoC families that are supported by this programmer. It is a union of the following flags:

CAN_PROGRAM_CY8C25xxx_CY8C26xxx	0x01
CAN_PROGRAM_ENCORE	0x02

- Capabilities[4] – contains a set of protocols (interfaces) supported by the programmer:

JTAG	0x01
ISSP	0x02
I2C	0x04
SWD	0x08
SPI	0x10

- Capabilities[5] – contains a set of internal voltage sources of the programmer that can be supplied to the target board:

VOLT_50V	0x01
VOLT_33V	0x02
VOLT_25V	0x04
VOLT_18V	0x08

C#:
```
int GetProgrammerCapabilities(out object caps, out string strError);
```

```
string strError;
object programmerCapabilities;
int hr = pp.GetProgrammerCapabilities(out programmerCapabilities,
```

```
                                                    out strError);
if (SUCCEEDED(hr))
{
byte[] caps = programmerCapabilities as byte[];
string strCaps = ">>>Programmer Capabilities:";
//Acquire Mode:
   strCaps += "\r\nCaps[0] Reset: ";
if ((caps[0] & (byte)enumValidAcquireModes.CAN_RESET_ACQUIRE) != 0)
   strCaps += "YES";
else
   strCaps += "NO";
   strCaps += "\r\nCaps[0] Power Cycle: ";
if((caps[0] & (byte)enumValidAcquireModes.CAN_POWER_CYCLE_ACQUIRE) != 0)
   strCaps += "YES";
else
   strCaps += "NO";
   strCaps += "\r\nCaps[0] Power Detect: ";
if((caps[0] & (byte)enumValidAcquireModes.CAN_POWER_DETECT_ACQUIRE)!=0)
   strCaps += "YES";
else
   strCaps += "NO";
//Can Power Device
   strCaps += "\r\nCaps[1] Can Power Device: ";
if ((caps[1] & (byte)enumCanPowerDevice.CAN_POWER_DEVICE) != 0)
   strCaps += "YES";
else
   strCaps += "NO";
   strCaps += "\r\nCaps[1] Can Read Power: ";
if ((caps[1] & (byte)enumCanPowerDevice.CAN_READ_POWER) != 0)
   strCaps += "YES";
else
   strCaps += "NO";
   strCaps += "\r\nCaps[1] Can Measure Power: ";
if ((caps[1] & (byte)enumCanPowerDevice.CAN_MEASURE_POWER) != 0)
   strCaps += "YES";
else
   strCaps += "NO";
   strCaps += "\r\nCaps[1] Can Measure Power 2: ";
if ((caps[1] & (byte)enumCanPowerDevice.CAN_MEASURE_POWER_2) != 0)
   strCaps += "YES";
else
   strCaps += "NO";
   //Can Update Firmware
   strCaps += "\r\nCaps[2] Firmware Update: ";
if (caps[2] != 0)
   strCaps += "YES";
else
   strCaps += "NO";
//Can Program
   strCaps += "\r\nCaps[3] Can Program CARBON: ";
if ((caps[3] & (byte)enumCanProgram.CAN_PROGRAM_CARBON) != 0)
   strCaps += "YES";
else
```

```
    strCaps += "NO";
    strCaps += "\r\nCaps[3] Can Program ENCORE: ";
if ((caps[3] & (byte)enumCanProgram.CAN_PROGRAM_ENCORE) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
//Supported Protocols
    strCaps += "\r\nCaps[4] Protocol SWD: ";
if ((caps[4] & (byte)enumInterfaces.SWD) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
    strCaps += "\r\nCaps[4] Protocol JTAG: ";
if ((caps[4] & (byte)enumInterfaces.JTAG) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
    strCaps += "\r\nCaps[4] Protocol ISSP: ";
if ((caps[4] & (byte)enumInterfaces.ISSP) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
    strCaps += "\r\nCaps[4] Protocol I2C: ";
if ((caps[4] & (byte)enumInterfaces.I2C) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
    strCaps += "\r\nCaps[4] Protocol SPI: ";
if ((caps[4] & (byte)enumInterfaces.SPI) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
//Voltages can be supplied by programmer
    strCaps += "\r\nCaps[5] Voltage 5.0V: ";
if ((caps[5] & (byte)enumVoltages.VOLT_50V) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
    strCaps += "\r\nCaps[5] Voltage 3.3V: ";
if ((caps[5] & (byte)enumVoltages.VOLT_33V) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
    strCaps += "\r\nCaps[5] Voltage 2.5V: ";
if ((caps[5] & (byte)enumVoltages.VOLT_25V) != 0)
    strCaps += "YES";
else
    strCaps += "NO";
        strCaps += "\r\nCaps[5] Voltage 1.8V: ";
    if ((caps[5] & (byte)enumVoltages.VOLT_18V) != 0)
        strCaps += "YES";
    else
        strCaps += "NO";
```

```
        MessageBox.Show(strCaps);
    }
      else
        MessageBox.Show(strError);
```

GetProgrammerCapsByName(string IN programmerName, nvector OUT caps, string OUT strError)

Returns the capabilities of the programmer by its name. This function is useful when it is necessary to get the capabilities of the programmer without opening its port. The structure of the *caps* parameter is described in the GetProgrammerCapabilities() function. The corresponding C# example demonstrates how to use this parameter. Below is simplified example that shows usage only of the function's interface.

C#:
```
int GetProgrammerCapsByName(string portName, out object caps,
    out string strError);

string strError;
    object programmerCapabilities;
int hr = pp.GetProgrammerCapsByName("MiniProg3/300000000000",
      out programmerCapabilities, out strError);
  if (SUCCEEDED(hr))
      MessageBox.Show("Capabilities read successfully");
    else
      MessageBox.Show(strError);
```

GetProgrammerVersion(string OUT versionString, string OUT strError)

Returns a string that contains the programmer version. This string has no special format; it briefly describes the programmer and its version.

C#:
```
 int GetProgrammerVersion(out string versionString, out string
strError);

string strError;
string version;
int hr = pp.GetProgrammerVersion(out version, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show(version);
else
    MessageBox.Show(strError);
```

GetSiliconId(nvector OUT siliconID, string OUT strError)

Reads the silicon ID from the PSoC device and sets up the software to work with the device. This function can be called after the device is successfully acquired by one of the Acquire functions. The returned vector is four bytes:

- siliconID[0] – high byte of PSoC silicon ID
- siliconID[1] – low byte of PSoC silicon ID
- siliconID[2] – Revision ID of the PSoC device
- siliconID[3] – Family ID of the PSoC device

C#: `int GetSiliconId(out object siliconID, out string strError);`

```
string strError;
object sId;
int hr = pp.GetSiliconId(out sId, out strError);
if (SUCCEEDED(hr))
{
    byte[] siliconID = sId as byte[];
    if (siliconID.GetLength(0) >= 4)
    {
        string s;
        s = "Silicon: " + ((siliconID[0] << 8) +
                        siliconID[1]).ToString("X4") +
            "  Revision: " + siliconID[2].ToString("X2") +
            "  Family: " + siliconID[3].ToString("X2");
        MessageBox.Show(s);
    }
}
else

        MessageBox.Show(strError);
```

GetRowsPerArrayInFlash (string OUT rowsPerArray, string strError)

Returns rows per array in flash of the acquired device.

C#: `int GetRowsPerArrayInFlash(out int rowsPerArray, out string strError;`

```
string strError;
int hr = pp. GetRowsPerArrayInFlash(out rowsPerArray, out strError);
if (SUCCEEDED(hr)){
string msg = "Info about flash:";
msg += "\r\nRows Per Array: " + rowsPerArray;
MessageBox.Show(msg);
}
else
MessageBox.Show(strError);
```

HEX_ReadChecksum(unsigned short OUT checksum, string OUT strError)

Reads the checksum from the hex file. Before you call this function, you must load the hex file using the HEX_ReadFile() function.

C#: `int HEX_ReadChecksum(out ushort checksum, out string strError);`

```
ushort checksum;
string strError;
int hr = pp.HEX_ReadChecksum(out checksum, out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("HEX checksum: "+checksum.ToString("X4"));
   else
     MessageBox.Show(strError);
```

HEX_ReadConfig(IN address, IN size, nvector OUT data,string OUT strError)

Reads data from the section with configuration data in the active hex file. This function reads *size* bytes starting at *address*.

C#:
```
int HEX_ReadConfig(int addr, int size, out object data,
out string strError);

string strError;
object data;
int hr = pp.HEX_ReadConfig(0, 5, out data, out strError);
if (SUCCEEDED(hr))
{
   string msg = "Config Data from Hex File: \r\n";
   byte[] configData = data as byte[];
   for (int i = 0; i < configData.Length; i++)
   msg += configData[i].ToString("X2") + " ";
   MessageBox.Show(msg);
}
else
   MessageBox.Show(strError);
```

HEX_ReadChipProtection(nvector OUT data, string OUT strError)

Reads the chip-level protection record.from the hex file

C#:
```
int HEX_ReadChipProtection(out object data, out string strError);
object data;
hr = pp.HEX_ReadChipProtection(out data, out strError);
if (SUCCEEDED(hr))
{
     string msg = "Chip Protection data from Hex File: \r\n";
     byte[] chipProtData = data as byte[];
     for (int i = 0; i < chipProtData.Length; i++)
     msg += chipProtData[i].ToString("X2") + " ";
     MessageBox.Show(msg);
}
else
     MessageBox.Show(strError);
```

HEX_ReadData(IN address, IN size, nvector OUT data, string OUT strError)

Reads *size* bytes of the hex file at *address*.

C#:
```
int HEX_ReadData(int addr, int size, out object data, out string
strError);

object readData;
string strError;
int hr = pp.HEX_ReadData(0,5,out readData, out strError);
```

```
if (SUCCEEDED(hr))
{
      byte[] data = readData as byte[];
      string s = "Data from HEX file: \r\n";
      for (int i = 0; i < 5; i++) s+=data[i].ToString("X2")+" ";
      MessageBox.Show(s);
}
   else
      MessageBox.Show(strError);
```

HEX_ReadEEPROM(IN address, IN size, nvector OUT data,string OUT strError)

Reads data from the section with EEPROM data in the active hex file. It reads *size* bytes starting at *address*.

C#:
```
int HEX_ReadEEPROM(int addr, int size, out object data,
out string strError);
string strError;
object data;
int hr = pp.HEX_ReadEEPROM(0, 5, out data, out strError);
if (SUCCEEDED(hr))
{
      string msg = "EEPROM Data from Hex File: \r\n";
      byte[] eepromData = data as byte[];
      for (int i = 0; i < eepromData.Length; i++)
      msg += eepromData[i].ToString("X2") + " ";
      MessageBox.Show(msg);
}
else
     MessageBox.Show(strError);
```

HEX_ReadExtra(nvector OUT data, string OUT strError)

Returns all metadata blocks from the hex file of the PSoC device.

C#:
```
int HEX_ReadExtra( out object data, out string strError);
string strError;
object data;
int hr = pp.HEX_ReadExtra(out data, out strError);
if (SUCCEEDED(hr))
{
      string msg = "Meta-Data from Hex File: \r\n";
      byte[] metaData = data as byte[];
      for (int i = 0; i < metaData.Length; i++)
      msg += metaData[i].ToString("X2") + " ";
      MessageBox.Show(msg);
      }
   else
      MessageBox.Show(strError);
```

HEX_ReadFile(string IN fileName, OUT imageSize, string OUT strError)

Reads the hex file from *fileName*.

C#:
```
int HEX_ReadFile(string fileName, out int imageSize, out string
strError);
int imageSize;
string strError;
int hr = pp.HEX_ReadFile("c:\\cy29466.hex",out imageSize,
out strError);
if (SUCCEEDED(hr))
        MessageBox.Show("Hex File loaded. Image Size = "+imageSize);
     else
        MessageBox.Show(strError);
```

HEX_ReadImageSizes(OUT flashSize, OUT configSize, OUT eepromSize, OUT nvlUserSize, OUT nvlWolSize, string OUT strError)

Returns the size of each programmable section from the active hex file. The sections are: Flash area, Config area, EEPROM block, User NVL, and WOL NVL.

C#:
```
int HEX_ReadImageSizes(out int flashSize,
out int configSize, out int eepromSize,
out int nvlUserSize, out int nvlWoSize,
out string strError);
string strError;
object data;
int flashSize, configSize, eepromSize, nvlUserSize, nvlWolSize;
int hr = pp.HEX_ReadImageSizes(out flashSize, out configSize,
out eepromSize, out nvlUserSize,
out nvlWolSize, out strError);
if (SUCCEEDED(hr))
{
        string msg = "Sizes of Hex sections:";
        msg += "\r\nFlash - "+flashSize;
        msg += "\r\nConfig - " + configSize;
        msg += "\r\nEEPROM - " + eepromSize;
        msg += "\r\nUser NVL - " + nvlUserSize;
        msg += "\r\nWOL NVL - " + nvlWolSize;
        MessageBox.Show(msg);
}
     else
        MessageBox.Show(strError);
```

HEX_ReadJtagID(nvector IN data, string OUT strError)

Read JtagID from the active hex file. This function is only applicable to PSoC hex files. This ID should be used to check compatibility of the acquired device and hex file.

C#:
```
int HEX_ReadJtagID(out object jtagID,
out string strError);
```

```
string strError;
object jtagID;
int hr = pp.HEX_ReadJtagID(out jtagID, out strError);
if (SUCCEEDED(hr))
{
      string msg = "JTAG ID: ";
      byte[] id = jtagID as byte[];
      for (int i = 0; i < id.Length; i++)
{
      msg += id[i].ToString("X2") + " ";
}
       MessageBox.Show(msg);
}
     else
       MessageBox.Show(strError);
```

HEX_ReadNvlCustom(IN address, IN size, nvector OUT data,string OUT strError)

Reads data from the section with Custom NVL data in the active hex file. It reads *size* bytes starting at *address*.

C#:
```
int HEX_ReadNvlCustom(int addr, int size,
out object data, out string strError);
string strError;
object data;
int hr = pp.HEX_ReadNvlCustom(0, 4, out data, out strError);
if (SUCCEEDED(hr))
{
      string msg = "Custom NVL Data from Hex File: \r\n";
      byte[] nvlData = data as byte[];
      for (int i = 0; i < nvlData.Length; i++)
      msg += nvlData[i].ToString("X2") + " ";
      MessageBox.Show(msg);
}
     else
      MessageBox.Show(strError);
```

HEX_ReadNvlWo(IN address, IN size, nvector OUT data, string OUT strError)

Reads data from the section with Write Once NVL data in the active hex file. It reads *size* bytes starting at *address*.

C#:
```
int HEX_ReadNvlWo(int addr, int size,
out object data, out string strError);
string strError;
object data;
int hr = pp.HEX_ReadNvlWo(0, 4, out data, out strError);
if (SUCCEEDED(hr))
{
      string msg = "Write Once NVL Data from Hex File: \r\n";
```

```
        byte[] nvlData = data as byte[];
        for (int i = 0; i < nvlData.Length; i++)
        msg += nvlData[i].ToString("X2") + " ";
        MessageBox.Show(msg);
}
     else
      MessageBox.Show(strError);
```

HEX_ReadProtection(IN address, IN size, nvector OUT data)

Reads *size* bytes starting at *address* from the protection record.

C#:
```
int HEX_ReadProtection(int addr, int size, out object data, out
string strError);
object protectionData;
string strError;
int hr = pp.HEX_ReadProtection(0,10,out protectionData,
out strError);
if (SUCCEEDED(hr))
{
        byte[] data = protectionData as byte[];
        string s = "Protection Data from HEX file: \r\n";
        for (int i = 0; i < 10; i++) s += data[i].ToString("X2") + " ";
        MessageBox.Show(s);
}
     else
        MessageBox.Show(strError);
```

HEX_WriteChipProtection(nvector IN data, string OUT strError)

Writes the chip-level protection record to the hex file.

C#:
```
int HEX_WriteChipProtection(object data, out string strError);
object data;
byte[] data = new byte[1] {0x01};
hr = pp.HEX_WriteChipProtection(data, out strError);
if (SUCCEEDED(hr))
{
        string msg = "Chip Protection data were written succeeded \r\n";
        MessageBox.Show(msg);
}
      else
        MessageBox.Show(strError);
```

HEX_WriteData(IN address, nvector IN data, string OUT strError)

Writes a block of bytes into the data area of the active hex image starting from *address*. Before calling this function, you must load a hex image with the HEX_ReadFile() function.

C#:
```
int HEX_WriteData(int address, object data, out string strError)
```

```
string strError;
byte[] data = new byte[]{1,2,3,4,5,6,7,8,9,10};
int hr = pp.Hex_WriteData(5,data, out strError);
if (SUCCEEDED(hr))
        MessageBox.Show("Hex Data written successfully!");
      else
        MessageBox.Show(strError);
```

HEX_WriteEEPROM(IN address, nvector IN data, string OUT strError)

Writes a block of bytes into the EEPROM area of the active hex starting from *address*. Before calling this function, you must load a hex with the HEX_ReadFile() function.

C#: `int HEX_WriteEEPROM(int address, object data, out string strError)`

```
string strError;
byte[] data = new byte[]{1,2,3,4,5,6,7,8,9,10};
int hr = pp.Hex_WriteEEPROM(5,data, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Hex EEPROM written successfully!");
else
   MessageBox.Show(strError);
```

HEX_WriteFile(string IN filename, string OUT strError)

Saves the active hex image into the specified file. Before calling this function, you must load a hex image with the HEX_ReadFile() function.

This function is useful when you need to modify a hex file and save the changes.

C#: `int HEX_WriteFile(string fileName, out string strError)`

```
string strError;
int hr = pp.HEX_WriteFile("c:\\test.hex", out strError);
if (SUCCEEDED(hr))
        MessageBox.Show("Hex File written successfully!");
      else
        MessageBox.Show(strError);
```

HEX_WriteProtection(IN address,nvector IN data, string OUT strError)

Writes a block of bytes into the protection area of the active hex image starting from *address*. Before calling this function, you must load a hex image with the HEX_ReadFile() function.

C#: `int HEX_WriteProtection(int address, object data, out string strError)`

```
string strError;
byte[] data = new byte[] { 1, 2, 3 };
int hr = pp.HEX_WriteProtection(125, data,out strError);
if (SUCCEEDED(hr))
        MessageBox.Show("Hex Protection Data written successfully!");
      else
        MessageBox.Show(strError);
```

HEX_GetDataInRange (IN startAddr, IN endAddr, nvector OUT data, string OUT strError)

Retrieves *data* from hex file within the given address [*startAddr*, *endAddr*] range.

C#:
```
int HEX_GetDataInRange(uint startAddr, uint endAddr, out object data,
out string strError); string strError;
object data;
uint rowSize = 0x200;
int hr = pp.HEX_GetDataInRange(0x10000000, 0x10000000 + rowSize - 1, out
data, out strError);
if (!SUCCEEDED(hr))
    MessageBox.Show(strError);
hr = pp.PSoC6_WriteRow(0, data, out strError);
if (!SUCCEEDED(hr))
MessageBox.Show(strError);
```

HEX_GetDataSizeInRange (IN startAddr, IN endAddr, OUT size, string OUT strError)

Gets the *size* of data in hex file within the given address [*startAddr*, *endAddr*] range.

C#:
```
int HEX_GetDataSizeInRange(uint startAddr, uint endAddr, out int size,
out string strError); string strError;
int size;
uint rowSize = 0x200;
int hr = pp.HEX_GetDataSizeInRange(0x10000000, 0x10000000 + rowSize - 1,
out size, out strError);
if (SUCCEEDED(hr))
MessageBox.Show(string.Format("The hex data size in range [0x{0:x5}-
0x{1:x5}] is 0x{2:x5}", 0x00, 0x10000, size));
else
MessageBox.Show(strError);
```

IsPortOpen(OUT isOpen, string OUT strError)

Returns the port status. A response of 0 means the port is closed; otherwise, the port is open.

C#:
```
int IsPortOpen(out int isOpen, out string strError);
int isOpen;
string strError;
int hr = pp.IsPortOpen(out isOpen, out strError);
if (SUCCEEDED(hr))
                MessageBox.Show("Port Open Status: " + isOpen);
            else
                MessageBox.Show(strError);
```

I2C_DataTransfer(IN deviceAddr, IN mode, IN readLen, nvector IN dataIN, nvector OUT dataOUT, string OUT strError)

This low-level protocol function is used as the sole communication function for all other I2C_xxx functions. This section explains how to use it within I²C transactions. The parameters are:

- *deviceAddr* – address of I²C slave device
- *mode* – a union of bits that define I²C bus protocol signals, which will be generated with the current packet

Table 2-4. Bits and Description

Bits	Description
0	Write/Read I²C operation
1	Make “Start” condition, send address byte
2	Make “ReStart” condition, send address byte
3	Make “Stop” condition after data transfer
4	Reinitialize I²C bus
5	Make I²C reconfiguration

- *readLen* – number of bytes to be read from the slave device. This parameter is used only if the *mode* parameter’s bit 0 is set to 1 (Read).
- *dataIN* – data to be sent to the I²C slave device. This parameter is used only if the *mode* parameter’s bit 0 is set to 0 (Write). For a read operation, this array is disregarded and its size can be set to 0.
- *dataOUT* – data returned as a result of the transaction. It has different meanings for Read and Write operations. The only common meaning is the first byte—the acknowledgement result of the address byte (1-ACK, 0 - NACK).
  - Read – dataOUT.size = 1 + readLen (bytes), where *readLen* is the bytes actually read from the device. They are meaningful if the address byte is ACKed.
  - Write – dataOUT.size = 1 + dataIN.size (bytes). The second part of the packet contains the ACK/NACK result for every sent byte. Normally, when the first NACK byte occurs, all others are NACKed as well.

This function can be used for find control of the I²C bus. For example, it can be used for communication with a slave device when one transaction is divided into several:

- mode = 0x02 (Start + Write, No Stop) – bus is busy after this transaction
- mode = 0x05 (Restart + Read, No Stop) – bus is still busy
- mode = 0x0D (Restart + Read + Stop) – bus is released after this transaction

All other high-level I²C communication functions have the Start and Stop bits set. They are all executed at one time.

C#:
```
int I2C_DataTransfer(int deviceAddr, int mode, int readLen,
      object dataIN, object dataOUT,
out string strError);

string strError;
            object dataIn = new byte[] { 0x04, 0x01, 0x00, 0x01};
```

```
            object dataOut;
int hr = pp.I2C_DataTransfer(0x00, 0x0A, 0, dataIn, out dataOut,
out strError);//Start+Write+Stop
if (SUCCEEDED(hr)) // data read successfully
            {
            byte[] data = dataOut as byte[];
             string s = "Transaction passed successfully. DataOUT = \r\n";
            for (byte i = 0; i < data.Length; i++)
               s = s + data[i].ToString("X2") + " ";
                MessageBox.Show(s);
            }
            else
                MessageBox.Show("Transaction failed!");
```

I2C_GetDeviceList(nvector OUT deviceList, string OUT strError)

Scans the I^{2}C bus for connected slave devices. This function uses the current setting of bus frequency. It returns an array with addresses that ACKed their address.

C#:
```
int I2C_GetDeviceList(out object deviceList,
  out string strError);

string strError;
            object deviceList;
int hr = pp.I2C_GetDeviceList(out deviceList, out strError);
if (!SUCCEEDED(hr))
               MessageBox.Show("Failed to enumerate I2C devices. ErrMsg="
+
   strError);
            else
            {
                string s;
                byte[] devices = deviceList as byte[];
                s = "Device Found: "+devices.Length;
                for (int i=0; i<devices.Length ;i++){
                    s += "\r\n"+devices[i];
                }
                MessageBox.Show(s);
            }
```

I2C_GetSpeed(enumI2Cspeed OUT speed, string OUT strError)

Reads the current speed of the I^{2}C bus. The possible values of speed enumeration are:

- CLK_100K 0x01
- CLK_400K 0x02
- CLK_50K 0x04
- CLK_1000K 0x05

1-MHz speed is supported only by MiniProg3 and TrueTouch Bridge devices.

C#:
```
int I2C_GetSpeed(out enumI2Cspeed speed, out string strError);
```

```
string strError;
            enumI2Cspeed speed;
int hr = pp.I2C_GetSpeed(out speed, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("I2C bus speed = "+speed.ToString());
 else
   MessageBox.Show("Failed to read bus speed.ErrorMsg = "+ strError);
```

I2C_ReadData(IN deviceAddr, IN readSize, nvector OUT data, string OUT strError)

Reads *readSize* bytes from the given slave device.

C#:
```
int I2C_ReadData(int deviceAddr, int readSize,
     out object data, out string strError);
string strError;
            object data;
int hr = pp.I2C_ReadData(0, 4, out data, out strError);
if (SUCCEEDED(hr))
{
     byte[] readData = data as byte[];
     string s = "Read Data:\r\n";
     for (int i = 0; i < readData.Length; i++)
{
     s+=readData[i].ToString("X2")+" ";
}
      MessageBox.Show(s);
}
else
  MessageBox.Show("Failed to send data!");
```

I2C_ReadDataFromReg(IN deviceAddr, nvector IN readAddr, IN readSize, nvector OUT data,string OUT strError)

Reads r*eadSize* bytes in the *data* array from the register (memory address, offset) specified by *readAddr* of the I^2C slave device addressed by *deviceAddress*. This function is used to support the Cypress I^2C Port Expander with EEPROM. It can be also used with I^2C devices that have a similar transaction sequence on the I^2C bus during read register operations. It is assumed that the I^2C device has addressable memory organization. For more information about this device, see the CY8C95xx datasheet and Application Note AN2034 - Communication - I2C Port Expander with Flash Storage at http://www.cypress.com. Note that the *readAddr* parameter is an array of bytes, because the address can usually be 1 or 2 bytes wide.

C#:
```
int I2C_ReadDataFromReg(int deviceAddr, object readAddr,
 int readSize, out object data,
 out string strError);
 string strError;
            object dataIn;
            byte[] ReadAddress = new byte[1] { 0x02 };
 int hr = pp.I2C_ReadDataFromReg(0x00, ReadAddress, 5,
   out dataIn, out strError);
```

```
            byte[] data = dataIn as byte[];
if (SUCCEEDED(hr)) // data read successfully
    {
     string s = "";
     for (byte i = 0; i < data.Length; i++)
     s = s + data[i].ToString("X2") + " ";
     MessageBox.Show("Read data from addr 0x02 of Device 0x00: "+ s);
    }
```

I2C_ResetBus(string OUT strError)

Resets a I²C bus. This is a hardware reinitialization of the I²C bus in case of bus hangup.

C#:
```
int I2C_ResetBus(out string strError);
string strError;
int hr = pp.I2C_ResetBus(out strError);
if (SUCCEEDED(hr))
      MessageBox.Show("I2C Bus reinitialized successfully!");
    else
      MessageBox.Show("Failed to reset I2C bus. ErrMsg = "+strError);
```

I2C_SendData(IN deviceAddr, nvector IN data, string OUT strError)

Sends an array of data to the given device on the bus.

C#:
```
int I2C_SendData(int deviceAddr, object data,
out string strError);

string strError;
            byte[] data = { 0x04, 0x01, 0x00, 0x01};
int hr = pp.I2C_SendData(0, data, out strError);
if (SUCCEEDED(hr))
         MessageBox.Show("Data sent to I2C device successfully!");
      else
         MessageBox.Show("Failed to send data!");
```

I2C_SetSpeed(enumI2Cspeed IN speed, string OUT strError)

Sets the speed of the I²C bus. The possible values of speed enumeration are the same as for the I2C_GetSpeed() function.

C#:
```
int I2C_SetSpeed(enumI2Cspeed speed, out string strError);

string strError;
int hr = pp.I2C_SetSpeed(enumI2Cspeed.CLK_100K, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("I2C bus speed set successfully!");
  else
    MessageBox.Show("Failed to set bus speed. Error Msg = " +
                                    strError);
```

JTAG_EnumerateDevices(OUT devices, string OUT strError)

Enumerates devices of the JTAG chain and returns jtagIDs of the detected devices in the order TDI (0s) to TDO (n–1). This function is implemented for MiniProg3 only. To put PSoC 3 or PSoC 5 devices into JTAG mode, this function first acquires all PSoC 3 and PSoC 5 chips on the chain and then enumerates them. Before calling this function, you must make sure you have set the correct acquire mode.

C#:
```
int JTAG_EnumerateDevices(out object devices, out string strError);
   string strError;
   object devices;
   int hr;
   hr = pp.SetAcquireMode("Reset", out strError);
   hr = pp.SetProtocol(enumInterfaces.JTAG, out strError);
   hr = pp.SetProtocolClock(enumFrequencies.FREQ_01_5, out strError);
   hr = pp.SetProtocolConnector(1, out strError); //10-pin connector
   hr = pp.JTAG_EnumerateDevices(out devices, out strError);
   if (SUCCEEDED(hr))
            {
                string[] jtagIDs = devices as string[];
                string s = "";
                for (int i = 0; i < jtagIDs.Length; i++)
                {
                    s += jtagIDs[i]+"\r\n";
                }
                MessageBox.Show(s);
            }
            else
                MessageBox.Show("Failed to enumerate JTAG chain!");
```

JTAG_SetChainConfig(IN deviceAddr, nvector IN Ir, string OUT strError)

Use this function to set the parameters of JTAG chain for PSoC 3 / PSoC 5/ PSoC 5LP. You must pass the parameters address of the target chip on chain "deviceAddr" and sizes of the instructions of each device on chain “Ir” to COM-object.

C#:
```
int JTAG_SetChainConfig(byte deviceAddr, object Ir, out string
strError);

string strError;
int deviceCount = 2;
UInt32[] Ir = new UInt32[deviceCount]{4,4};
byte deviceAddr = 1;

int hr = pp.JTAG_SetChainConfig(deviceAddr, Ir, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("JTAG parameters setup successfully!");
else
   MessageBox.Show(strError);
```

JTAG_SetIR (IN IR, string OUT strError)

Sets instruction register on the selected device on JTAG chain. The length of IR and address of selected device must be set by JTAG_SetChainConfig() API before using this API. This JTAG transaction ends in IDLE state and can start from any TAP's state.

C#: `int JTAG_SetIR(int IR, out string strError);`

```
string strError;
const int deviceCount = 2;
UInt32[] Ir = new UInt32[deviceCount]{4,4};
byte deviceAddr = 1;
int hr = pp.JTAG_SetChainConfig(deviceAddr, Ir, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("JTAG parameters setup successfully!");
else
   MessageBox.Show(strError);
hr = pp.DAP_AcquireChip(out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Device is aqcuired");
else
   MessageBox.Show(strError);
hr = pp.JTAG_SetIR(0x35, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Instruction register setup successfully!");
else
   MessageBox.Show(strError);
```

JTAG_ShiftDR (IN drSize, nvector IN drIn, nvector OUT drOut, string OUT strError)

Shifts in DR "drSize" bits from "drIn" array, and at the same time "drOut" array receives "size" bits shifted out from DR. Bits in "drIn" and "drOut" arrays come in the little ending format. This transaction ends in the IDLE and suppose to start also from IDLE. So it is recommended to call JTAG_SetIR() to initialize IR and move TAP to IDLE, before using JTAG_ShiftDR() API. It operates on the device selected in last call to JTAG_SetChainConfig().

C#: `int JTAG_ShiftDR(int size, object drIn, out object drOUT, out string strError);`

```
string strError;
const byte APACC = 0x0B;
const int DR_SIZE = 35;
int hr = pp.JTAG_SetIR(APACC, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Instruction register setup successfully!");
else
   MessageBox.Show(strError);
//Write Address
object drOut;
long addr = 0x20000008;
addr = (addr << 3) | 0x02; // R/W = 0b, Addr = 01b
```

```
byte[] drIn = new byte[] {(byte)(addr >> 0),
                          (byte)(addr >> 8),
                          (byte)(addr >> 16),
                          (byte)(addr >> 24),
                          (byte)(addr >> 32)};
hr = pp.JTAG_ShiftDR(DR_SIZE, drIn, out drOut, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Bits were successfully shifted");
else
   MessageBox.Show(strError);
```

JTAG_ShiftIR (IN irSize, nvector IN irIn, nvector OUT irOut, string OUT strError)

Shifts in IR “irSize” bits from “irIn” array, and at the same time “irOut” array receives “size” bits shifted out from IR. Bits in “irIn” and “irOut” arrays come in the little ending format. This transaction ends in the IDLE and suppose to start also from IDLE. Therefore, it is recommended to call JTAG_SetIR() to initialize IR and move TAP to IDLE, before using JTAG_ShiftIR() API. It operates on the device selected in last call to JTAG_SetChainConfig().

C#:
```
int JTAG_ShiftIR(int size, object irIn, out object irOUT, out string
strError);
```

```
string strError;
const byte APACC = 0x0B;
const int IR_SIZE = 0x04;
int hr = pp.JTAG_SetIR(APACC, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Instruction register setup successfully!");
else
   MessageBox.Show(strError);
byte[] outIR, inIR, inDR, outDR;
inIR = new byte[1] { 0x0E };
hr = pp.JTAG_ShiftIR(IR_SIZE, inIR, out outIR, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Bits were successfully shifted");
else
   MessageBox.Show(strError);
inDR = new byte[4] {0x00, 0x00, 0x00, 0x00};
hr = pp.JTAG_ShiftDR(0x20, inDR, out outDR, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Bits were successfully shifted");
else
   MessageBox.Show(strError);
```

JTAGIO(IN read, nvector IN tdi_tms, nvector OUT tdo)

Sends TDI+TMS bits to the JTAG chain. If *read* is not 0, then *tdo* returns read-out TDO bits.

C#:
```
int jtagio(byte read, object tdi_tms, out object tdo);
string strError;
int hr;
```

```
    hr = pp.SetAcquireMode("Reset", out strError);
    hr = pp.SetProtocol(enumInterfaces.JTAG, out strError);
    hr = pp.SetProtocolClock(enumFrequencies.FREQ_01_5, out strError);
    hr = pp.SetProtocolConnector(1, out strError); //10-pin connector
    hr = pp.DAP_Acquire(out strError);
          //Read in TDO - JTAG ID of connected PSoC 3 / PSoC 5 device
    byte read = 1;
    byte[] tdi_tms = new byte[] { 0xF0, 0xF0, 0xF0, 0x20, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x00 };
    byte[] TDO = null;
    object tdo = null;
    hr = pp.jtagio(read, tdi_tms, out tdo);
    if (!SUCCEEDED(hr))
          MessageBox.Show("Failed to communicate on JTAG-bus");
    TDO = tdo as byte[];
    string strOut = "";
    for (byte i = 0; i < TDO.Length; i++)
    strOut = strOut + TDO[i].ToString("X2") + " ";
          MessageBox.Show("TDO: " + strOut);.
```

JTAGIOR(IN Addr, OUT data)

Reads content of the register of PSoC 3 or PSoC 5 device using the JTAG-interface. Note that this function works only in the 1:1 configuration of the bus.

C#: `int jtagior(int address, out int data);`

JTAGIOW(IN Addr, IN data)

Writes *data* to the register of the PSoC 3 or PSoC 5 using the JTAG-interface. Note that this function works only in the 1:1 configuration of the bus.

C#:
```
int jtagiow(int address, int data);
string strError;
int hr;
hr = pp.SetAcquireMode("Reset", out strError);
hr = pp.SetProtocol(enumInterfaces.JTAG, out strError);
hr = pp.SetProtocolClock(enumFrequencies.FREQ_01_5, out strError);
hr = pp.SetProtocolConnector(1, out strError); //10-pin connector
hr = pp.DAP_Acquire(out strError);
int addr = 0x04;
hr = pp.jtagiow(addr, 0x75);
if (!SUCCEEDED(hr))
     MessageBox.Show("Failed to write register of PSoC3 device");

int data;
hr = pp.jtagior(addr, out data);
if (!SUCCEEDED(hr))
     MessageBox.Show("Failed to read register of PSoC3 device");
   else
     MessageBox.Show("Read data: "+data.ToString("X8"));
```

OpenPort(string IN port, string OUT strError)

Opens the port. This port is one of the strings returned by the GetPorts() function. In some cases, this function returns a FAIL result, but the port is opened. This happens when the firmware of the programmer is outdated and needs to be upgraded. Use the IsOpen() function to check whether the port was actually opened, and read the strError message to see if you need to update the firmware.

You must open the port only once to connect a programmer device (such as MiniProg1, MinipPort3, and so on). After that, you can use chip-specific or programmer-specific APIs as needed without reopening the port.

You must call **OpenPort()** again if the port is closed by the **ClosePort()** method, when you must connect to another programmer, or if the programmer is physically reconnected to the USB port.

C#: `int OpenPort(string port, out string strError);`

```
string strError;
int hr = pp.OpenPort("MINIProg1/869A8846200C", out strError);
if (SUCCEEDED(hr)){
    string version;
    pp.GetProgrammerVersion(out version, out strError);
    MessageBox.Show(version);
}
else
    MessageBox.Show(strError);
```

PowerOff(string OUT strError)

Tells the programmer to stop powering the PSoC device.

C#: `int PowerOff(out string strError);`

```
string strError;
int hr = pp.PowerOff(out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Programmer Powered OFF");
else
    MessageBox.Show(strError);
```

PowerOn(string OUT strError)

Applies power to the target PSoC device

C#: `int PowerOn(out string strError);`

```
string strError;
int hr = pp.PowerOn(out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Programmer Powered ON");
  else
    MessageBox.Show(strError);
```

ProgrammerLedState(IN ledNo, IN ledState, string OUT strError)

Sets the programmer's onboard LED state to *ledState,* which defines the mode of the LED states: 0 - Normal, 1 - Chaser, 2 - All On, 3 - All Off). Currently, this function is supported only by the MiniProg3 device. Also, if *ledNo* = -1, then it identifies the accessible feature of MiniProg3.

C#:
```
int ProgrammerLedState(int ledNo, int ledState,
out string strError);
string strError;
int ledNo = 1;
            //LED Flashing 1
int ledState = 1;
int hr = pp.ProgrammerLedState(ledNo, ledState ,out strError);
            System.Threading.Thread.Sleep(1000);
            //LED On
            ledState = 3; //on
            hr = pp.ProgrammerLedState(ledNo, ledState, out strError);
            System.Threading.Thread.Sleep(1000);
            //LED Flashing 2
            ledState = 2;
            hr = pp.ProgrammerLedState(ledNo, ledState, out strError);
            System.Threading.Thread.Sleep(1000);
            //LED Off
            ledState = 0; //off
         hr = pp.ProgrammerLedState(ledNo, ledState, out strError);;
```

Protect(IN bankID, string OUT strError)

Protects a bank of a PSoC device according to the current hex file. Flash banks must be erased before they can be protected. Use EraseAll() operation before protecting flash banks.

C#:
```
int Protect(int bankID);

string strError;
int hr = pp.Protect(0, out strError);
if (SUCCEEDED(hr))
       MessageBox.Show("Bank 0 protected!");
   else
      MessageBox.Show(strError);
```

ProtectAll(string OUT strError)

Protects a part according to the current hex file. Flash banks must be erased before they can be protected. Use EraseAll() operation before protecting flash banks.

C#:
```
int ProtectAll(out string strError);
string strError;
int hr = pp.ProtectAll(out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Flash Protected!");
else
    MessageBox.Show(strError);
```

```
C#: int ProtectAll(out string strError);
string strError;
int hr;
byte[] data = new byte[] { 1, 2, 3, 4, 5};
hr = pp.HEX_ReadFile("c:\\PSoC1.hex");
hr = pp.HEX_WriteProtect(0x00, data, out strError);
hr = pp.HEX_WriteFile("c:\\PSoC1.hex", out strError);
int hr = pp.ProtectAll(out strError);
if (SUCCEEDED(hr))
       MessageBox.Show("Flash Protected!");
else
       MessageBox.Show(strError);
```

PSoC3_CheckSum(IN arrayID, IN startRowID, IN noOfRows, uint OUT checksum, string OUT strError)

Calculates the checksum of the *noOfRows* rows starting at *startRowID* in the array *arrayID*. To find the checksum of the entire flash (all arrays), you should calculate the checksum of each array and then total them. There is no method to calculate the checksum of all flash in the PSoC 5 silicon. For PSoC 3, which has one array only, it can be done with one API call (see the example below).

```
C#: int PSoC3_CheckSum(int arrayID, int startRowID, int noOfRows,
    out uint checksum, out string strError);
string strError;
int arrayID = 0;
int startRowID = 0;
int noOfRows = 256;
uint checksum;
int hr = pp.PSoC3_CheckSum(arrayID, startRowID, noOfRows,
    out checksum, out strError);
if (SUCCEEDED(hr))
                MessageBox.Show("Checksum: "+checksum.ToString("X08"));
```

PSoC3_DebugPortConfig(IN value, string OUT strError)

Sets the value of Debug Port Configuration (DBGPRT_CFG) register of the Test Controller. This register allows you to adjust the auto-increment setting and transfer size. The transfer size options are 8, 16, and 32 bits. This register affects functionality of DAP_WriteIO() and DAP_ReadIO() functions and all others that are dependent on I/O communication with a PSoC 3 or PSoC 5 device in Test mode. Use fthe ollowing values to set up the transfer size:

- 0 – 8 bits
- 2 – 16 bits
- 4 – 32 bits

```
C#: int PSoC3_DebugPortConfig(int transferMode,
    out string strError);

string strError;
int hr = pp.PSoC3_DebugPortConfig(4, out strError);
if (SUCCEEDED(hr))
       MessageBox.Show("DebugPort Config register set successfully!");
```

```
else
  MessageBox.Show("Failed: " + strError);
```

PSoC3_EraseAll(string OUT strError)

Erases all user locations in flash for all flash arrays on the chip, and then erases all flash protection rows for all flash arrays on the chip. For flash arrays with ECC capability, this also erases all ECC bytes.

C#:
```
int PSoC3_EraseAll(out string strError);
string strError;
int hr = pp.PSoC3_EraseAll(out strError);
if (SUCCEEDED(hr))
                MessageBox.Show("Flash erased successfully!");
            else
             MessageBox.Show("Failed: " + strError);
```

PSoC3_GetEccStatus(OUT eccStatus, string OUT strError)

For PSoC 3 devices that support an Error Correction Code (ECC) in the flash, the returned *eccStatus* (bit in Customer NVL) determines whether ECC status is enabled.
Possible returned values: 0 – disabled, 1 – enabled.

C#:
```
int PSoC3_GetEccStatus(out int status, out string strError);

string strError;
int eccStatus;
int hr = pp.PSoC3_GetEccStatus(out eccStatus, out strError);
if (SUCCEEDED(hr))
                MessageBox.Show("ECCEN bit value: "+eccStatus+
     " --> ECC "+((eccStatus==0)?"Disabled":"Enabled"));
            else
             MessageBox.Show("Failed: " + strError);
```

PSoC3_GetFlashArrayInfo(IN arrayID, OUT rowSize, OUT rowsPerArray, OUT eccPresence, string OUT strError)

Retrieves characteristics of given flash array:

- rowSize – size of the flash row in bytes
- rowsPerArray – number of rows in this array
- eccPresence – whether the array supports ECC capability. (To find whether ECC is enabled, use PSoC3_GetEccStatus().)

To use this function, the device must either have been acquired using DAP_Acquire() or DAP_GetJtagID() must have been called once after the device was acquired.

C#:
```
int PSoC3_GetFlashArrayInfo(int arrayID, out int rowSize,
out int rowsPerArray, out int eccPresence,
out string strError);

string strError;
int arrayID = 0;
```

```
int rowSize, rowsPerArray, eccPresence;
int hr = pp.PSoC3_GetFlashArrayInfo(arrayID, out rowSize,
out rowsPerArray, out eccPresence, out strError);
if (SUCCEEDED(hr))
            {
                string msg = "Info about flash array ID = "+arrayID;
                msg += "\r\nRows Per Array: " + rowsPerArray;
                msg += "\r\nRow Size      : " + rowSize;
                msg += "\r\nECC capability: " + eccPresence;
            MessageBox.Show(msg);
        }
          else
            MessageBox.Show(strError);
```

PSoC3_GetSonosArrays(enumSonosArrays IN arrayType, OUT arraysInfo, string OUT strError)

Gets information about arrays of a given type. *arrayType* is one of the following values (or union of them):

```
public enum enumSonosArrays
{
        ARRAY_FLASH = 1,
        ARRAY_EEPROM = 2,
        ARRAY_NVL_USER = 4,
        ARRAY_NVL_FACTORY = 8,
        ARRAY_NVL_WO_LATCHES = 16,
        ARRAY_ALL = 31,
}
```

The *arraysInfo* output parameter contains the ID and size of each array of the given type. It is a two-dimensional array of the following view:

	0	1		N-1
0	arrayID	arrayID	...	arrayID
1	arraySize	arraySize	...	arraySize

C#:
```
int PSoC3_GetSonosArrays(enumSonosArrays arrayGroup,
out object arrays, out string strError);

string strError;
object data;
int hr = pp.PSoC3_GetSonosArrays(enumSonosArrays.ARRAY_ALL,
out data, out strError);
if (SUCCEEDED(hr))
          {
               string msg = "Information about All Sonos Arrays: ";
               Array arrayInfo = data as Array;
               for (int i = arrayInfo.GetLowerBound(1);
    i <= arrayInfo.GetUpperBound(1); i++)
```

```
            {
                int arrayID = (int)arrayInfo.GetValue(0, i);
                int arraySize = (int)arrayInfo.GetValue(1, i);
                msg += "\r\n ArrayID = "+arrayID.ToString("X2")+
" ArraySize = "+arraySize;
            }
                MessageBox.Show(msg);
            }
              else
                MessageBox.Show(strError);
```

PSoC3_ProgramRow(IN arrayID, IN rowID, nvector IN data, string OUT strError)

Programs the addressed row of the flash array with *data*. If the flash array supports ECC and it is disabled, *data* must contain user and config data of the row. For, example, the first 256 bytes may be user data and following 32 bytes may be config data. Requirements:

- *arrayID* must be within Flash or EEPROM range
- *rowID* must not be write protected

C#:
```
int PSoC3_ProgramRow(int arrayID, int rowID, object data,
        out string strError);
string strError;
int arrayID = 0, rowID = 255;
      byte[] data = new byte[288]; //256 + 32(Config data - ECC disabled)
    r (int i = 0; i < data.Length; i++) data[i] = (byte)i;

int hr = pp.PSoC3_ProgramRow(arrayID, rowID, data, out strError);
if (SUCCEEDED(hr))
      MessageBox.Show("Row "+rowID+" in array "+arrayID +
   " Programmed Successfully!");
   else
      MessageBox.Show(strError);
```

PSoC3_ProgramRowFromHex(IN arrayID, IN rowID, IN eccOption, string OUT strError)

Programs the addressed row of the flash array with data from the active hex file. The *eccOption* parameter determines whether the function should add config data to the user block. eccOption should be 0 if the flash array has no ECC capability, or if it has ECC capability and ECC is enabled. Otherwise, it must be 1.

C#:
```
int PSoC3_ProgramRowFromHex(int arrayID, int rowID,
int eccOption, out string strError);
string strError;
int arrayID = 0, rowID = 254, eccOption = 1;
int hr = pp.PSoC3_ProgramRowFromHex(arrayID, rowID, eccOption,
out strError);
if (SUCCEEDED(hr))
          MessageBox.Show("Row " + rowID + " in array " + arrayID +
   " Programmed Successfully!");
            else
```

```
            MessageBox.Show(strError);
```

PSoC3_ProtectAll(string OUT strError)

Programs the flash protection row in each flash array using data from the active hex file. This function can be run only if none of the protection bits are currently set.

C#:
```
int PSoC3_ProtectAll(out string strError);
string strError;
int hr = pp.PSoC3_ProtectAll(out strError);
if (SUCCEEDED(hr))
            MessageBox.Show("All Flash Arrays are protected!");
         else
            MessageBox.Show("Failed: "+strError);
```

C#:
```
 int ProtectAll(out string strError);
string strError;
byte[] data = new byte[] { 1, 2, 3, 4, 5};
int hr;
hr = pp.HEX_ReadFile("c:\\PSoC3.hex");
hr = pp.HEX_WriteProtect(0x00, data, out strError);
hr = pp.HEX_WriteFile("c:\\PSoC3.hex", out strError);
int hr = pp.PSoC3_ProtectAll(out strError);
if (SUCCEEDED(hr))
             MessageBox.Show("Flash Protected!");
else
             MessageBox.Show(strError);
```

PSoC3_ProtectArray(IN arrayID, nvector IN data, string OUT strError)

Programs the flash protection row with *data*. The size of the data array must correspond to the number of rows in the addressed array. This function can be run only if none of the protection bits are currently set.

C#:
```
int PSoC3_ProtectArray(int arrayID, object data,
out string strError);
string strError;
int arrayID = 0;
           byte[] data = new byte[64];
           for (int i = 0; i < data.Length; i++) data[i] = 0xFF;
int hr = pp.PSoC3_ProtectArray(arrayID, data, out strError);
if (SUCCEEDED(hr))
          MessageBox.Show("Flash Arrays "+arrayID+" is protected!");
       else
          MessageBox.Show("Failed: " + strError);
```

PSoC3_ReadNvlArray(IN arrayID, nvector OUT data, string OUT strError)

Reads the NVL array from the acquired chip. The *arrayID* parameter must address a valid NVL array (Custom NVL = 0x80, Write-Once = 0xF8, Factory = 0xC0).

C#:
```
int PSoC3_ReadNvlArray(int arrayID, out object data,
out string strError);
```

```
string strError;
int arrayID = 0x80;
            object data;
int hr = pp.PSoC3_ReadNvlArray(arrayID, out data, out strError);
if (SUCCEEDED(hr))
            {
                string msg = "Custom NVL Data read from Chip: \r\n";
                byte[] nvlData = data as byte[];
                for (int i = 0; i < nvlData.Length; i++)
                    msg += nvlData[i].ToString("X2") + " ";
                MessageBox.Show(msg);
            }
              else
                MessageBox.Show(strError);
```

PSoC3_ReadProtection(IN arrayID, nvector OUT data, string OUT strError)

Reads all protection bytes for *arrayID*. The array must be within a valid flash range.

C#:
```
int PSoC3_ReadProtection(int arrayID, out object data,
out string strError);
string strError;
int arrayID = 0x00;
          object data;
int hr = pp.PSoC3_ReadProtection(arrayID, out data, out strError);
if (SUCCEEDED(hr))
          {
              string msg = "Protection data of array: "+arrayID+"\r\n";
              byte[] protData = data as byte[];
              for (int i = 0; i < protData.Length; i++)
                  msg += protData[i].ToString("X2") + " ";
              MessageBox.Show(msg);
          }
          else
              MessageBox.Show(strError);
```

PSoC3_ReadRow(IN arrayID, IN rowID, IN eccOption, nvector OUT data, string OUT strError)

Reads an entire flash row addressed by *arrayID* and *rowID*. The *eccOption* parameter specifies whether the user (0) or configuration (1) area of the row is read out. The row must not be in a read-protected state.

C#:
```
int PSoC3_ReadRow(int arrayID, int rowID, int eccSpace,
out object data, out string strError);
string strError;
int arrayID = 0x00, rowID = 0;
            object data;
int hr = pp.PSoC3_ReadRow(arrayID, rowID, 0, out data,out strError);
if (SUCCEEDED(hr))
       {
          string msg = "Row Data of ArrayID=" + arrayID +
```

```
" RowID="+rowID+"\r\n";
            byte[] rowData = data as byte[];
            for (int i = 0; i < rowData.Length; i++)
            msg += rowData[i].ToString("X2") + " ";
            MessageBox.Show(msg);
          }
          else
            MessageBox.Show(strError);
```

PSoC3_VerifyProtect(string OUT strError)

Verifies the protection area of all flash arrays against the active hex file.

C#:
```
int PSoC3_VerifyProtect(out string strError);
    string strError;
    int hr = pp.PSoC3_VerifyProtect(out strError);
    if (SUCCEEDED(hr))
                MessageBox.Show("Protection area verified successfully!");
              else
                MessageBox.Show(strError);
```

PSoC3_VerifyRowFromHex(IN arrayID, IN rowID, IN eccOption, OUT verResult, string OUT strError)

Verifies the row specified by *arrayID* and *rowID* against the active hex file. If *eccOption* = 0, only the user area of the row is verified; otherwise, configuration data is included. The *verResult* output parameter = 0 if verification fails and 1 if passed.

C#:
```
int PSoC3_VerifyRowFromHex(int arrayID, int rowID,
int eccOption, out int verResult, out string strError);
string strError;
int arrayID = 0, rowID = 5, eccOption = 0;
int verResult = 0;
int hr = pp.PSoC3_VerifyRowFromHex(arrayID, rowID, eccOption,
out verResult, out strError);
if (SUCCEEDED(hr))
            MessageBox.Show("Verfication Result: " + verResult + " -> " +
(verResult == 0 ? "Failed" : "Passed"));
        else
           MessageBox.Show(strError);
```

PSoC3_WriteNvlArray(IN arrayID, nvector IN data, string OUT strError)

Writes the entire NVL array specified by *arrayID*. The size of the *data* parameter must be equal to the size of the array. For this operation to succeed, no flash protection bits should be set. The possible values for *arrayID* can be found in the PSoC3_ReadNvlArray() method. Note that Factory NVLs are not allowed for programming in this field; it will break the silicon irreversibly.

C#:
```
int PSoC3_WriteNvlArray(int arrayID, object data,
 out string strError);
 string strError;
 int arrayID = 0x80;
```

```
            byte[] data = new byte[4] { 0,0,0,8 };
int hr = pp.PSoC3_WriteNvlArray(arrayID, data, out strError);
if (SUCCEEDED(hr))
                MessageBox.Show("NVL array programmed successfully!");
            else
                MessageBox.Show(strError);
```

PSoC3_WriteRow(IN arrayID, IN rowID, nvector IN data, string OUT strError)

Erases the addressed row and then programs it with *data* (both user data and ECC/configurtation data, if supported by the flash array). In contrast to PSoC3_ProgramRow(), this function does not fail if the row is already programmed. Requirements:

- *arrayID* must be within the flash or EEPROM range
- row must not be externally write protected.

C#:
```
int PSoC3_WriteRow(int arrayID, int rowID, object data,
   out string strError);
   string strError;
   int arrayID = 0, rowID = 255;
            byte[] data = new byte[288]; //256 + 32(ECC disabled)
            for (int i = 0; i < data.Length; i++) data[i] = (byte)i;

   int hr = pp.PSoC3_WriteRow(arrayID, rowID, data, out strError);
   if (SUCCEEDED(hr))
               MessageBox.Show("Row " + rowID + " in array " + arrayID +
  " Programmed Successfully!");
            else
             MessageBox.Show(strError);
```

PSoC3_GetEepromArrayInfo (IN arrayID, OUT rowSize, OUT rowsPerArray, string OUT strError)

Retrieves characteristics of given EEPROM array:

- *rowSize* - size of the EEPROM row in bytes
- *rowsPerArray* - number of rows in the array

To use this function, the device must either have been acquired using DAP_Acquire() or DAP_GetJtagID() must have been called once after the device was acquired.

C#:
```
int PSoC3_GetEepromArrayInfo(int arrayID, out int rowSize,
out int rowsPerArray, out string strError);
string strError;
int arrayID = 0x40;
int rowSize, rowsPerArray;
int hr = pp.PSoC3_GetEepromArrayInfo(arrayID, out rowSize,
           out rowsPerArray, out strError);
if (SUCCEEDED(hr))
{
   string msg = "Info about EEPROM array ID = " + arrayID;
   msg += "\r\nRows Per Array: " + rowsPerArray;
```

```
    msg += "\r\nRow Size      : " + rowSize;
    MessageBox.Show(msg);
}
else
    MessageBox.Show(strError);
```

PSoC3_EraseSector(IN arrayID, IN sectorID, string OUT strError)

Erases a given sector of flash or EEPROM array. A sector is a block of 64 contiguous rows that is set on a 64-row boundary. For flash arrays with ECC capability, this also erases associated ECC bytes. This API is useful when you must erase all EEPROM.

Requirements:

- *arrayID* must be within flash or EEPROM range
- *rowID* must not be write protected

C#:
```
int PSoC3_EraseSector(int arrayID, int sectorID,
                                  out string strError);
string strError;
//Power-on EEPROM
pp.DAP_WriteIO(0x43AC, 0x11, out strError); //PSoC3
//pp.DAP_WriteIO(0x400043AC, 0x11, out strError); //
//Erases All EEPROM - Sector 0 and 1
int arrayID = 0x40;
int sectorID = 0;
int hr = pp.PSoC3_EraseSector(arrayID, sectorID, out strError);
sectorID = 1;
hr = pp.PSoC3_EraseSector(arrayID, sectorID, out strError);

if (SUCCEEDED(hr))
    MessageBox.Show("EEPROM erased succesfully!");
else
    MessageBox.Show("Failed: " + strError);
```

PSoC3_EraseRow(IN arrayID, IN rowID, string OUT strError)

Erases a flash or EEPROM array. This API works only for PSoC 3 ES3 (Production) and PSoC 5LP.

Requirements:

- *arrayID* must be within flash or EEPROM range
- *rowID* must not be write protected

C#:
```
int PSoC3_EraseRow(int arrayID, int rowID, out string strError);
string strError;
//Erases Flash Row
int arrayID = 0, rowID = 5;
int hr = pp.PSoC3_EraseRow(arrayID, rowID, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Flash row erased succesfully!");
else
    MessageBox.Show("Failed with Flash: " + strError);
//Erases EEPROM Row
```

```
arrayID = 0x40; rowID = 0;
//Power-on EEPROM
pp.DAP_WriteIO(0x43AC, 0x11, out strError);//PSoC3
//pp.DAP_WriteIO(0x400043AC, 0x11, out strError); //
hr = pp.PSoC3_EraseRow(arrayID, rowID, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("EEPROM row erased succesfully!");
else
MessageBox.Show("Failed with EEPROM: " + strError);
```

PSoC4_CheckSum(IN rowID, OUT checksum, string OUT strError)

Calculates the checksum for *rowID*. If *rowID* = 0x8000, the checksum is calculated for the whole flash (privileged and user rows).

C#:
```
int PSoC4_CheckSum(int rowID, out int checksum, out string strError);
   string strError = "";
   int rowID = 0x01;
   int checksum;
   hr = pp.PSoC4_CheckSum(rowID, out checksum, out strError);
   if (SUCCEEDED(hr))
      MessageBox.Show("Checksum: " + checksum.ToString("X08"));
```

PSoC4_EraseAll(string OUT strError)

Erases all flash content including security rows and chip-level protection.

Refer to the PSoC4_WriteRow API to erase one row.

C#:
```
int PSoC4_EraseAll(out string strError);
string strError;
int hr = pp.PSoC4_EraseAll(out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Flash erased successfully!");
else
   MessageBox.Show("Failed: " + strError);
```

PSoC4_GetFlashInfo(string OUT rowsPerFlash, OUT rowSize, string strError)

Returns flash characteristics of the acquired device.

Note that the `rowsPerFlash` parameter returns the maximum possible number of rows for the family, but the acquired part number can have lesser flash rows. See the Ordering Information in the datasheet for the actual flash size for the part number of the interest and divide the flash size by the row size to get the actual number of rows.

C#:
```
int PSoC4_GetFlashInfo(out int rowsPerFlash, out int rowSize, out
string strError);
string strError;
int hr = pp.PSoC4_GetFlashInfo(out rowsPerFlash, out rowSize, out
strError);
if (SUCCEEDED(hr)){
```

```
    string msg = "Info about flash:";
    msg += "\r\nRows Per Flash: " + rowsPerFlash;
    msg += "\r\nRow Size : " + rowSize;
    MessageBox.Show(msg);
}
else
    MessageBox.Show(strError);
```

PSoC4_GetSiliconID(nvector OUT siliconID, string OUT strError)

Reads the silicon ID from a PSoC 4 device in SWD mode. This function can be called after device is successfully acquired by the DAP_Acquire() functions. The returned vector is four bytes in length:

- siliconID[0] - high byte of PSoC silicon ID
- siliconID[1] - low byte of PSoC silicon ID
- siliconID[2] - Revision ID of the PSoC device
- siliconID[3] - Family ID of the PSoC device

C#: `int PSoC4_SiliconID(out object siliconID, out string strError);`

```
string strError;
object siliconID;
int hr = pp.PSoC4_GetSiliconID(out siliconID, out strError);
if (SUCCEEDED(hr)) {
    string msg = "Silicon ID: ";
    byte[] id = siliconID as byte[];
    for (int i = 0; i < id.Length; i++)
      msg += id[i].ToString("X2") + " ";
    MessageBox.Show(msg);
}
else
    MessageBox.Show(strError);
```

PSoC4_ProgramRow(IN rowID, nvector IN data, string OUT strError)

Programs *data* to *row*. The row must be unprotected and erased.

C#:
```
int PSoC4_ProgramRow(int rowID, int data, object data,
out string strError);
string strError;
int rowID = 0;
byte[] data = new byte[128];
for (int i = 0; i < data.Length; i++) data[i] = (byte)i;
int hr = pp.PSoC4_ProgramRow(rowID, data, out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("Row " + rowID + " Programmed Successfully!");
else
     MessageBox.Show(strError);
```

PSoC4_ProgramRowFromHex(IN rowID, string OUT strError)

Programs *row* of flash using data from the hex file.

C#:
```
int PSoC4_ProgramRowFromHex(int rowID, out string strError);
string strError;
int rowID = 0x00;
int hr = pp.PSoC4_ProgramRowFromHex(rowID, out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("Row " + rowID + " Programmed Successfully!");
else
     MessageBox.Show(strError);
```

PSoC4_ProtectAll(string OUT strError)

Protects all flash arrays and writes chip-level protection using data from the hex file.

C#:
```
int PSoC4_ProtectAll(out string strError);
string strError;
int hr = pp.PSoC4_ProtectAll(out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("All Flash are protected!");
else
     MessageBox.Show("Failed: "+strError);
```

C#:
```
int ProtectAll(out string strError);
 strError;
byte[] data = new byte[] { 1, 2, 3, 4, 5};
int hr;
hr = pp.HEX_ReadFile("c:\\PSoC4.hex");
hr = pp.HEX_WriteProtect(0x00, data, out strError);
hr = pp.HEX_WriteFile("c:\\PSoC4.hex", out strError);
int hr = pp.PSoC4_ProtectAll(out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Flash Protected!");
else
   MessageBox.Show(strError);
```

PSoC4_ReadProtection(nvector OUT flashProtect, nvector OUT chipProtect, string OUT strError)

Reads flash and chip-level protection. Chip protection must be in the OPEN state.

C#:
```
int PSoC4_ReadProtection(out object flashProtect, out object
chipProtect, out string strError);
string strError = "";
object flashProtect, chipProtect;
int hr = pp.PSoC4_ReadProtection(out flashProtect, out chipProtect,
      out strError);
if (SUCCEEDED(hr))
   {
       string msg = "Flash protection data: \r\n";
       byte[] flProtect = flashProtect as byte[];
```

```
        for (int i = 0; i < flProtect.Length; i++)
            msg += flProtect[i].ToString("X2") + " ";
        msg += "\r\nChip protection data: \r\n";
        byte[] chProtect = chipProtect as byte[];
        for (int i = 0; i < chProtect.Length; i++)
            msg += chProtect[i].ToString("X2") + " ";
        MessageBox.Show(msg);
    }
    else
        MessageBox.Show(strError);
```

PSoC4_ReadRow(IN rowID, nvector OUT data, string OUT strError)

Reads the content of a flash row. The row must be unprotected, and chip-level protection must be in the OPEN state.

C#:
```
int PSoC4_ReadRow(int rowID, out object data, out string strError);
string strError;
   int rowID = 0;
   object data;
   int hr = pp.PSoC4_ReadRow(rowID, out data, out strError);
   if (SUCCEEDED(hr))
   {
       string msg = "Row ID " + rowID + "\r\n";
       byte[] rowData = data as byte[];
       for (int i = 0; i < rowData.Length; i++)
           msg += rowData[i].ToString("X2") + " ";
       MessageBox.Show(msg);
   }
   else
       MessageBox.Show(strError);
```

PSoC4_VerifyProtect(string OUT strError)

Verifies flash and chip-level protection using data from the opened hex file.

C#:
```
int PSoC4_VerifyProtect(out string strError);
string strError;
int hr = pp.PSoC4_VerifyProtect(out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Protection area verified successfully!");
else
   MessageBox.Show(strError);
```

PSoC4_VerifyRowFromHex(IN rowID, OUT verResult, string OUT strError)

Verifies a row of flash using data from the hex file. Flash must be unprotected.

C#:
```
int PSoC4_VerifyRowFromHex(int rowID, out int verResult, out string
strError);
string strError;
int rowID = 5;
int verResult = 0;
 int hr = pp.PSoC4_VerifyRowFromHex(rowID, out verResult, out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("Verfication Result: " + verResult + " -> " +
       (verResult == 0 ? "Failed" : "Passed"));
else
     MessageBox.Show(strError);
```

PSoC4_WriteProtection(nvector IN flashProtect, nvector IN chipProtect, string OUT strError)

Writes flash and chip-level protection.

C#:
```
int PSoC4_WriteProtection(object flashProtect, object chipProtect,
out
string strError)
string strError;
byte[] flashProtect = new byte[64];
byte[] chipProtect = new byte[] { 1 };
for (int i = 0; i < flashProtect.Length; i++)
  flashProtect[i] = 0xFF;
for (int i = 0; i < chipProtect.Length; i++)
  chipProtect[i] = 0x01;
int hr = pp.PSoC4_WriteProtection(flashProtect, chipProtect,out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("Protection Data written successfully!");
else
     MessageBox.Show(strError);
```

PSoC4_WriteRow(IN rowID, nvector IN data, string OUT strError)

Writes *data* to *rowID* of a flash array. The row must be unprotected and may contain any data.

Note: PSoC 4 does not have a separate API to erase a row. Write 0x00 into the row using PSoC4_WriteRow API to erase it.

C#:
```
int PSoC4_WriteRow(int rowID, object data, out string strError);
string strError;
int rowID = 0x00;
byte[] data = new byte[128];
for (int i = 0; i < data.Length; i++) data[i] = (byte)i;
int hr = pp.PSoC4_WriteRow(rowID, data, out strError);
if (SUCCEEDED(hr))
```

```
    MessageBox.Show("Row " + rowID + " Programmed Successfully!");
else
    MessageBox.Show(strError);
```

FM0_VerifyRowFromHexOneRead (IN rowID, IN size, nvector IN chipData, OUT verResult, string OUT strError)

Verifies a row of flash using data from the hex file and data, which will be read by the FM0_ReadRow API. Flash must be unprotected.

Note: This API can check data in all sectors for FM0+ devices.

C#:
```
int FM0_VerifyRowFromHexOneRead (int rowID, int rowSize, object data,
out byte result out string strError);
string strError;
int rowID = 0;
int rowSize;
object readData;
byte verResult;
int  hr = pp.FM0_GetFlashInfo(out rowsPerFlash, out rowSize, out m_sLas-
tError);
if(!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}

hr = pp.FM0_ReadRow(rowID, out readData, out m_sLastError);
if(!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}
hr = pp. FM0_VerifyRowFromHexOneRead(rowID, rowSize, readData, out
verResult, out m_sLastError);
if (SUCCEEDED(hr))
{
    MessageBox.Show("Verification Result: " + verResult + " -> " +
(verResult == 0 ? "Failed" : "Passed"));
}
else
{
    MessageBox.Show(strError);
}
```

FM0_ProgramRow (IN rowID, nvector IN data, string OUT strError)

Programs data to row. The row must be unprotected and erased.

C#:
```
int FM0_ProgramRow(int rowID, int data, object data, out string
strError);
string strError;
```

```
int rowID = 0;
byte[] data = new byte[128];
for (int i = 0; i < data.Length; i++) data[i] = (byte)i;
int hr = pp.FM0_ProgramRow(rowID, data, out strError);
if (SUCCEEDED(hr))
{
    MessageBox.Show("Row " + rowID + " Programmed Successfully!");
}
else
{
    MessageBox.Show(strError);
}
```

FM0_EraseAll(OUT strError)

Erases all flash content including security rows and chip-level protection. Refer to the FM0_Erase-Sector API to erase one sector.

C#:
```
int FM0_EraseAll(out strError);
string strError;
int hr = pp.FM0_EraseAll(out strError);
if (SUCCEEDED(hr))
{
    MessageBox.Show("Flash erased successfully!");
 }
else
{
    MessageBox.Show(strError);
}
```

FM0_CheckSum (IN rowID, OUT checksum, string OUT strError)

Calculates the checksum for rowID.

C#:
```
int FM0_CheckSum(int rowID, out int checksum, out string strError);
string strError = "";
int rowID = 0x01;
int checksum;
hr = pp. FM0_CheckSum(rowID, out checksum, out strError);
if (SUCCEEDED(hr))
{
    MessageBox.Show("Checksum: " + checksum.ToString("X08"));
}
```

FM0_ReadRow (IN rowID, nvector OUT data, string OUT strError)

Reads the content of a flash row. The row must be unprotected and chip-level protection must be in the OPEN state.

C#:
```
int FM0_ReadRow(int rowID, out object data, out string strError);
string strError;
int rowID = 0;
```

```
object data;
int hr = pp. FM0_ReadRow(rowID, out data, out strError);
if (SUCCEEDED(hr))
{
    string msg = "Row ID " + rowID + "\r\n";
    byte[] rowData = data as byte[];
    for (int i = 0; i < rowData.Length; i++)
        msg += rowData[i].ToString("X2") + " ";
    MessageBox.Show(msg);
 }
else
{
    MessageBox.Show(strError);
}
```

FM0_GetFlashInfo(string OUT rowsPerFlash, OUT rowSize, string strError)

Returns flash characteristics of the acquired device.

C#:

```
int FM0_GetFlashInfo(out int rowsPerFlash, out int rowSize, out string
strError);
string strError;
int hr = pp.FM0_GetFlashInfo(out rowsPerFlash, out rowSize, out strError);
if (SUCCEEDED(hr))
{
string msg = "Info about flash:";
    msg += "\r\nRows Per Flash: " + rowsPerFlash;
    msg += "\r\nRow Size : " + rowSize;
    MessageBox.Show(msg);
}
else
{
    MessageBox.Show(strError);
}
```

FM0_ProgramRowFromHex (IN rowID, string OUT strError)

Programs row of flash using data from the hex file. The row must be unprotected and erased.

C#:

```
int FM0_ProgramRowFromHex(int rowID, out string strError);
string strError;
int rowID = 0x00;
int hr = pp.FM0_ProgramRowFromHex(rowID, out strError);
 if (SUCCEEDED(hr))
{
    MessageBox.Show("Row " + rowID + " Programmed Successfully!");
}
else
{
    MessageBox.Show(strError);
}
```

FM0_VerifyRowFromHex (IN rowID, OUT verResult, string OUT strError)

Verifies a row of flash using data from the hex file. Flash must be unprotected.

C#:
```
int FM0_VerifyRowFromHex(int rowID, out int verResult, out string
strError);
string strError;
int rowID = 5;
int verResult = 0;
int hr = pp. FM0_VerifyRowFromHex(rowID, out verResult, out strError);
 if (SUCCEEDED(hr))
{
    MessageBox.Show("Verification Result: " + verResult + " -> " +
(verResult == 0 ? "Failed" : "Passed"));
 }
else
{
    MessageBox.Show(strError);
}
```

FM0_GetSiliconID (nvector IN siliconID, string OUT strError)

Reads the silicon ID from a FM0+ device in SWD mode. This function can be called after the device is successfully acquired by the DAP_Acquire() functions.

C#:
```
int FM0_SiliconID(out object siliconID, out string strError);
string strError;
object siliconID;
int hr = pp.FM0_GetSiliconID(out siliconID, out strError);
if (SUCCEEDED(hr))
{
    string msg = "Silicon ID: ";
    byte[] id = siliconID as byte[];
    for (int i = 0; i < id.Length; i++)
        msg += id[i].ToString("X2") + " ";
    MessageBox.Show(msg);
}
else
{
    MessageBox.Show(strError);
}
```

FM0_EraseSector (IN rowID, string OUT strError)

This API erases the full sector, which contains the current row ID. Flash must be unprotected.

C#:
```
int FM0_EraseSector (int rowID, out string strError);
string strError;
int rowID = 5;
int hr = pp.FM0_EraseSector(rowID, out strError);
if (SUCCEEDED(hr))
```

```
{
    MessageBox.Show("Sector with row " + rowID + " Erased Successfully!");
}
else
{
    MessageBox.Show(strError);
}
```

FL_Init(nvector IN data, string OUT strError)

This API loads the flash loader in RAM. When the load is complete, execute ‘configure CPU’ and run the Init function from flash loader.

Note: Use the correct flash loader to correct devices with correct flash memory. For more information, read the datasheet with the chip name. Use the FL_UnInit API after this API. You can use the FL_ProgramPage, FL_ProgramSector, and FM0_FL_ProgramRowFromHex APIs between the FL_Init and FL_UnInit APIs.

C#:
```
int FL_Init (object data, out string strError);
string strError;
string localFlashLoader = "C:\Program Files (x86)\Cypress\Program-
mer\FlashLoaders\S6E1C11X0.bin"
byte[] flashloaderData = File.ReadAllbytes(localFlashLoader);
byte[] flashloaderDataNew = {};
for(int i = 0; i < flashloaderData.Length; i++)
{
      int count = 0;
      const int lenghtRamRow = 256;
      for(int j = i; j < i + lenghtRamRow; j++)
       {
            if(j > flashloaderData.Length - 1)
            {
                 if((j - i) == count)
                     count = lenghtRamRow;
                     break;
                 }
                 if(flashloaderData[j] == 0x00)
                 {
                     count++;
                 }
            }
            if(count != lenghtRamRow)
            {
                  Array.Resize(ref flashloaderDataNew, lenghtRamRow + i);
                  for(int index = i; index < i + lenghtRamRow; index++)
                  {
                      if(i == 0)
                       {
                           flashloaderDataNew.SetValue(flashloader-
Data[index], index);
                       } else
                       {
```

```
                        flashloaderDataNew.SetValue(flashloaderData[index
- 1], index - 1);
                         }
                    }
                }
                i = i + lenghtRamRow - 1;
            }
            int hr = pp.FL_Init(flashloaderDataNew, out strError);
            if (SUCCEEDED(hr))
{
MessageBox.Show("Flash loader " + localFlashLoader + " is loaded
successfully!");
}
else
{
MessageBox.Show(strError);
}
            hr = pp.FL_UnInit (out strError);
            if (SUCCEEDED(hr))
{
MessageBox.Show("Flash loader " + localFlashLoader + " is unloaded
successfully!");
}
else
{
MessageBox.Show(strError);
}
```

FL_UnInit(string OUT strError)

This API executes the UnInit function from RAM and return the MCU in normal state.

Note: Use the FL_Init API before this API. You can use the FL_ProgramPage, FL_ProgramSector, and FM0_FL_ProgramRowFromHex APIs between the FL_Init and FL_UnInit APIs.

```
C#: int FL_UnInit (out string strError);
string strError;
string localFlashLoader = "C:\Program Files (x86)\Cypress\Program-
mer\FlashLoaders\S6E1C11X0.bin"
byte[] flashloaderData = File.ReadAllBytes(localFlashLoader);
byte[] flashloaderDataNew = {};
for(int i = 0; i < flashloaderData.Length; i++)
{
      int count = 0;
      const int lenghtRamRow = 256;
      for(int j = i; j < i + lenghtRamRow; j++)
       {
            if(j > flashloaderData.Length - 1)
            {
                 if((j - i) == count)
                     count = lenghtRamRow;
                     break;
                 }
                 if(flashloaderData[j] == 0x00)
```

```
                    {
                        count++;
                    }
                }
                if(count != lenghtRamRow)
                {
                     Array.Resize(ref flashloaderDataNew, lenghtRamRow + i);
                     for(int index = i; index < i + lenghtRamRow; index++)
                     {
                         if(i == 0)
                          {
                              flashloaderDataNew.SetValue
(flashloaderData[index], index);
                          } else
                          {
                           flashloaderDataNew.SetValue(flashloaderData[index
- 1], index - 1);
                            }
                        }
                    }
                    i = i + lenghtRamRow - 1;
                }
                int hr = pp.FL_Init(flashloaderDataNew, out strError);
                if (SUCCEEDED(hr))
{
    MessageBox.Show("Flash loader " + localFlashLoader + " is loaded
successfully!");
}
else
{
    MessageBox.Show(strError);
}
                hr = pp.FL_UnInit (out strError);
                if (SUCCEEDED(hr))
{
    MessageBox.Show("Flash loader " + localFlashLoader + " is unloaded
successfully!");
}
else
{
    MessageBox.Show(strError);
}
```

FM0_FL_ProgramRowFromHex(IN rowId, IN rowSize, OUT m_sLastError)

Programs row of flash using data from the hex file. The row must be unprotected and erased.

Note: You can use the FM0_FL_ProgramRowFromHex API between the FL_Init and FL_UnInit APIs.

C#: . `int FM0_FL_ProgramRowFromHex (int rowID, int data, object data, out string strError);`

```
string strError;
```

```
int rowID = 0;
int length = 4*1024;
int hr = pp. FM0_FL_ProgramRowFromHex (rowID, length, out strError);
if (SUCCEEDED(hr))
{
    MessageBox.Show("Row " + rowID + " Programmed Successfully!");
}
else
{
    MessageBox.Show(strError);
}
```

FL_ProgramSector (IN address, nvector IN data, out m_sLastError);

Programs data in sector use flash loader functions. The sector must be unprotected and erased.

Note: Use the correct flash loader to correct devices with correct flash memory. For more information, read the datasheet with the chip name. Use the FL_UnInit API after this API. You can use the FL_ProgramSector API between the FL_Init and FL_UnInit APIs.

C#:.

```
int FL_ProgramSector(int address, object data , out string strError);
object data;
int rowSize;
int address = 0x0;
int hr = pp.FM0_GetFlashInfo(out rowsPerFlash, out rowSize, out m_sLastEr-
ror);
if(!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}

hr = pp.HEX_ReadData(address, rowSize, out data, out m_sLastError);
if(!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}

hr = pp.FM0_EraseSector(address, out m_sLastError);
if(!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}

hr = pp.FL_ProgramSector(address, data, out m_sLastError);
if (SUCCEEDED(hr))
{
    MessageBox.Show("Sector program successfully!");
}
else
{
    MessageBox.Show(strError);
}
```

FL_ProgramPage(IN rowID, nvector IN data, out m_sLastError);

Programs data in rows use flash loader functions. The rows must be unprotected and erased.

Note: Use the correct flash loader to correct devices with correct flash memory. For more information, read the datasheet with the chip name. Use the FL_UnInit API after this API. You can use the FL_ProgramPage API between the FL_Init and FL_UnInit APIs.

C#:
```
int FL_ProgramPage(int address, object data , out string strError);
object data;
int rowSize;
int rowID = 0;
int hr = pp.FM0_GetFlashInfo(out rowsPerFlash, out rowSize, out m_sLastEr-
ror);
if(!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}
hr = pp.HEX_ReadData(rowID, rowSize, out data, out m_sLastError);
if(!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}

hr = pp.FM0_EraseSector(rowID, out m_sLastError);
if(!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}

hr = pp.FL_ProgramPage (rowID, data, out m_sLastError);
if (SUCCEEDED(hr))
{
    MessageBox.Show("Page program successfully!");
}
else
{
    MessageBox.Show(strError);
}
```

FL_LoadElf(string IN algoPath, string OUT strError)

Loads and parses algorithms from ELF (*.FLM) file. Only 32 bit and little endian ELF files are supported. *algoPath* represents path to the ELF file.

C#:
```
int FL_LoadElf(string path, out string strError);
string strError;
string installPath = @"c:\Program Files (x86)\Cypress\Programmer";
string algoListPath = Path.Combine(installPath, @"FlashLoaders\AlgoL-
ist.xml");
string algoPath = string.Empty;
XmlDocument algoDoc = new XmlDocument();
algoDoc.Load(algoListPath);
```

```
XmlNode algoInfo = algoDoc.SelectSingleNode(string.Format("//Algo-
Info[@Name='{0}']", "CY8C6xx7_FLASH"));
if (algoInfo != null && algoInfo.Attributes != null)
{
algoPath=Path.Combine(installPath, algoInfo.Attributes["Path"].Value);
}
int hr = pp.FL_LoadElf(algoPath, out strError);
if (SUCCEEDED(hr))
{
MessageBox.Show("Flash programming algorithm has been parsed success-
fully!");
}
else
{
MessageBox.Show(strError);
}
```

FL_IsApiExists(string IN apiName, bool OUT exists, string OUT strError)

Checks whether *apiName* exists in the ELF file being loaded. It can be used to check optional (implementation is not mandatory) flashloader functions (e.g. BlankCheck, EraseChip, Verify). Returns *exists*=true in case of existence, otherwise *exists*=false.

C#: int FL_IsApiExists(string apiName, out bool exists, out string strError);

```
bool exists;
string strError;
string apiName = "Verify";
int hr = pp.FL_LoadElf(@"C:\Program Files (x86)\Cypress\Programmer\Flash-
Loaders\PSoC6\CY8C6xx7.FLM",
                out strError);
if (!SUCCEEDED(hr))
{
    MessageBox.Show("Unable to parse programming algorithm" + strError);
}
hr = pp.FL_IsApiExists(apiName, out exists, out strError);
if (!SUCCEEDED(hr) || !exists)
{
    MessageBox.Show("API does not exist in the provided ELF file.");
}
```

FL_ExecAPI(string IN apiName, IN r0, IN r1, IN r2, IN r3, IN r4, IN r5, IN r6, IN r7, IN dataBufferAddr, IN timeout, nvector IN dataBuffer, OUT apiResult, string OUT strError)

Executes *apiName* from the loaded ELF file within provided parameters. These parameters are:

- *r0 - r7* – R0-R7 ARM general purpose registers
- *dataBufferAddr* – address of data to be written
- *timeout* – time to wait for API complete until termination
- *dataBuffer* – data to be written

- *apiResult* – API execution result

C#:
```
int FL_ExecAPI(string apiName, int r0, int r1, int r2, int r3, int r4,
int r5, int r6, int r7, int dataBufferAddr, object dataBuffer, int time-
Out, out int apiResult, out string strError);
string strError;
string apiName = "EraseSector";
int apiResult;
int flashBase = 0x10000000;
int ramBase = 0x08002400;
int ramSize = 0x8000;
int timeOut = 5000;
int regIgnore = -1;
int hr = pp.FL_LoadElf(@"C:\Program Files (x86)\Cypress\Programmer\Flash-
Loaders\PSoC6\CY8C6xx7.FLM",
                out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to parse programming algorithm" + strError);
}
hr = pp.FL_SetRamForAlgorithms(ramBase, ramSize, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to set RAM area for algorithm usage" + strError);
}
hr = pp.FL_PrepareTarget(false, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to prepare target for algorithm execution!" +
strError);
}
hr = pp.FL_ExecCmsisApiInit(flashBase, 0, 1, timeOut, out apiResult, out
strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
MessageBox.Show("MCU initialization has failed!");
}
hr = pp.FL_ExecAPI(apiName, flashBase, regIgnore, regIgnore, regIgnore,
regIgnore,  regIgnore, regIgnore, regIgnore, regIgnore, timeOut, new
byte[0], out apiResult, out strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
    MessageBox.Show(strError);
}
```

FL_SetRamForAlgorithms(IN baseAddr, IN maxSize, string OUT strError)

Sets RAM area for algorithms usage to a *baseAddr* with a size of *maxSize*.

C#:
```
int FL_SetRamForAlgorithms(int baseAddr, int maxSize, out string
strError);
string installPath = @"c:\Program Files (x86)\Cypress\Programmer";
```

```
string algoListPath = Path.Combine(installPath, @"FlashLoaders\AlgoL-
ist.xml");
string strError;
int ramBase = 0, ramSize = 0;
XmlDocument algoDoc = new XmlDocument();
algoDoc.Load(algoListPath);
XmlNode algoInfo = algoDoc.SelectSingleNode(string.Format("//Algo-
Info[@Name='{0}']", "CY8C6xx7_FLASH"));
if (algoInfo != null && algoInfo.Attributes != null)
{
Int32Converter conv = new Int32Converter();
ramBase = (int)conv.ConvertFromString( algoInfo.Attri-
butes["RamAddr"].Value);
ramSize = (int)conv.ConvertFromString( algoInfo.Attributes["Ram-
Size"].Value);
}
int hr = pp.FL_SetRamForAlgorithms(ramBase, ramSize, out strError);
if (SUCCEEDED(hr))
{
MessageBox.Show("RAM area for algorithms have been set successfully");
}
else
{
MessageBox.Show(strError);
}
```

FL_PrepareTarget(bool IN cacheRAM, OUT strError)

Loads programming algorithm into the RAM area and prepares CPU for algorithm usage. *cacheRAM* parameter is reserved, not implemented yet.

Note: before this API usage MCU should be configured by using the following APIs: `FL_LoadElf`, `FL_SetRamForAlgorithms`.

C#:
```
int FL_PrepareTarget(bool cacheRAM, out string strError);
string strError;
int hr = pp.FL_PrepareTarget(false, out strError);
if (SUCCEEDED(hr))
{
MessageBox.Show("Target is ready for algorithm execution!");
}
else
{
MessageBox.Show("Unable to prepare target for algorithm execution!" +
strError);
}
```

FL_ExecCmsisApiInit(IN adr, IN clk, IN fnc, IN timeout, OUT apiResult, string OUT strError)

Initializes the microcontroller for Flash programming. *adr* is the device's base address and *clk* is the clock frequency used for device programming. Each `FL_ExecCmsisApiInit` has to end up with `FL_ExecCmsisApiUnInit` once the work is done.

fnc should be one of the following:

- Erase – 1
- Program – 2
- Verify – 3

Note: before Flashloader API execution MCU should be configured by using the following APIs: `FL_LoadElf`, `FL_SetRamForAlgorithms`, `FL_PrepareTarget`.

C#:
```
int FL_ExecCmsisApiInit(int adr, int clk, int fnc, int timeOut, out int apiResult, out string strError);
string strError;
int apiResult;
int flashBase = 0x10000000;
int fncCodeProgram = 2;
int timeOut = 5000;
int hr = pp.FL_ExecCmsisApiInit(flashBase, 0, fncCodeProgram, timeOut, out apiResult, out strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
MessageBox.Show("MCU initialization has failed!");
}
else
{
MessageBox.Show("MCU initialization completed!");
}
```

FL_ExecCmsisApiUnInit(IN fnc, IN timeout, out apiResult, string OUT strError)

De-initializes the MCU configured by `FL_ExecCmsisApiInit` and is invoked at the end of an erasing, programming, or verification step which should be specified by the *fnc* parameter. Returns *apiResult*=0 on success, *apiResult*=1 on failure.

C#:
```
int FL_ExecCmsisApiUnInit(int fnc, int timeOut, out int apiResult, out string strError);
string strError;
int apiResult;
int timeOut = 5000;
int fncCodeErase = 1;
int hr = pp.FL_ExecCmsisApiUnInit(fncCodeErase, timeOut, out apiResult, out strError);
if (SUCCEEDED(hr) && apiResult == 0)
{
MessageBox.Show("UnInit operation succeed!");
}
else
{
MessageBox.Show("UnInit operation failed!");
}
```

FL_ExecCmsisApiEraseSector(IN adr, IN timeout, OUT apiResult, string OUT strError)

Deletes the content of the sector starting at the address specified by the *adr* parameter. Returns *apiResult*=0 if erasing succeeds, otherwise erasing fails.

Note: first, the MCU must be prepared for flash algorithm execution by using: `FL_LoadElf`, `FL_SetRamForAlgorithms`, `FL_PrepareTarget`, `FL_ExecCmsisApiInit`.

C#:
```
int FL_ExecCmsisApiEraseSector(int adr, int timeOut, out int apiResult,
out string strError);
string strError;
int apiResult;
int flashBase = 0x10000000;
int ramBase = 0x08002400;
int ramSize = 0x8000;
int timeOut = 5000;
int fncCodeErase = 1;
int hr = pp.FL_LoadElf(@"C:\Program Files (x86)\Cypress\Programmer\Flash-
Loaders\PSoC6\CY8C6xx7.FLM", out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to parse programming algorithm" + strError);
}
hr = pp.FL_SetRamForAlgorithms(ramBase, ramSize, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to set RAM area for algorithm usage" + strError);
}
hr = pp.FL_PrepareTarget(false, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to prepare target for algorithm execution!" +
strError);
}
hr = pp.FL_ExecCmsisApiInit(flashBase, 0, fncCodeErase, timeOut, out
apiResult, out strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
MessageBox.Show("MCU initialization has failed!");
}
hr = pp.FL_ExecCmsisApiEraseSector(flashBase, timeOut, out apiResult, out
strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
MessageBox.Show("EraseSector API execution failed");
}
hr = pp.FL_ExecCmsisApiUnInit(fncCodeErase, timeOut, out apiResult, out
strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("UnInit operation failed!");
}
```

```
hr = pp.FL_ReleaseTarget(false, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("MCU release alter flash programming algorithms execution
failed!");
}
```

FL_ExecCmsisApiProgramPage(IN adr, IN sz, IN timeout, nvector IN dataBuffer, OUT apiResult, string OUT strError)

Writes code into the Flash memory. Code that being written is *dataBuffer* at the specified address=*adr* with the size=*sz*. Returns status information: 0 on success, 1 on failure.

Note: First, the MCU must be prepared for flash algorithm execution by using: `FL_LoadElf`, `FL_SetRamForAlgorithms`, `FL_PrepareTarget`, `FL_ExecCmsisApiInit`.

C#:
```
int FL_ExecCmsisApiProgramPage(int adr, int sz, object dataBuffer, int
timeOut, out int apiResult, out string strError);
string strError;
object data;
int apiResult;
int imageSize;
int flashBase = 0x10000000;
int ramBase = 0x08002400;
int ramSize = 0x8000;
int timeOut = 5000;
int fncCodeProgram = 2;
int programSize = 0x200;
int hr = pp.FL_LoadElf(@"C:\Program Files (x86)\Cypress\Programmer\Flash-
Loaders\PSoC6\CY8C6xx7.FLM",
                out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to parse programming algorithm" + strError);
}
hr = pp.FL_SetRamForAlgorithms(ramBase, ramSize, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to set RAM area for algorithm usage" + strError);
}
hr = pp.FL_PrepareTarget(false, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to prepare target for algorithm execution!" +
strError);
}
hr = pp.FL_ExecCmsisApiInit(flashBase, 0, fncCodeProgram, timeOut, out
apiResult, out strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
MessageBox.Show("MCU initialization has failed!");
}
```

```
hr = pp.FL_ExecCmsisApiEraseSector(flashBase, timeOut, out apiResult, out
strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
MessageBox.Show(strError);
}
hr = pp.HEX_ReadFile("C:\\DualCoreBlinky.hex", out imageSize, out
strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show(strError);
}
hr = pp.HEX_ReadData(flashBase, programSize, out data, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show(strError);
}
hr = pp.FL_ExecCmsisApiProgramPage(flashBase, programSize, timeOut, data,
out apiResult, out strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
    MessageBox.Show(strError);
}
else
{
    MessageBox.Show("ProgramPage API executed successfully!");
}
hr = pp.FL_ExecCmsisApiUnInit(fncCodeProgram, timeOut, out apiResult, out
strError);
if (!SUCCEEDED(hr))
{
    MessageBox.Show("UnInit operation failed!");
}
hr = pp.FL_ReleaseTarget(false, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("MCU release alter flash programming algorithms execution
failed!");
}
```

FL_ExecCmsisApiVerify(IN adr, IN sz, IN timeout, nvector IN dataBuffer, OUT apiResult, string OUT strError)

Compares the content of the Flash memory at *adr* with *sz* bytes of *dataBuffer*. Returns *apiResult=adr+sz* on success and on failure returns any other number, which represents the failing address.

Note: First, the MCU must be prepared for flash algorithm execution by using: `FL_LoadElf`, `FL_SetRamForAlgorithms`, `FL_PrepareTarget`, `FL_ExecCmsisApiInit`.

C#:
```
int FL_ExecCmsisApiVerify(int adr, int sz, int timeOut, object dataBuf-
fer, out int apiResult, out string strError);
string strError;
```

```
object data;
int apiResult;
int imageSize;
int flashBase = 0x10000000;
int ramBase = 0x08002400;
int ramSize = 0x8000;
int timeOut = 5000;
int fncCodeVerify = 3;
int verifSize = 0x200;
int hr = pp.FL_LoadElf(@"C:\Program Files (x86)\Cypress\Programmer\Flash-
Loaders\PSoC6\CY8C6xx7.FLM", out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to parse programming algorithm" + strError);
}
hr = pp.FL_SetRamForAlgorithms(ramBase, ramSize, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to set RAM area for algorithm usage" + strError);
}
hr = pp.FL_PrepareTarget(false, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("Unable to prepare target for algorithm execution!" +
strError);
}
hr = pp.FL_ExecCmsisApiInit(flashBase, 0, fncCodeVerify, timeOut, out
apiResult, out strError);
if (!SUCCEEDED(hr) || apiResult != 0)
{
MessageBox.Show("MCU initialization has failed!");
}
hr = pp.HEX_ReadFile("C:\\DualCoreBlinky.hex", out imageSize, out
strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show(strError);
}
hr = pp.HEX_ReadData(flashBase, verifSize, out data, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show(strError);
}
hr = pp.FL_ExecCmsisApiVerify(flashBase, verifSize, timeOut, data, out
apiResult, out strError);
if (!SUCCEEDED(hr) || apiResult != flashBase + verifSize)
{
if (!SUCCEEDED(hr))
{
MessageBox.Show("Verify API execution failed");
}
else
{
```

```
MessageBox.Show(string.Format("Failed data verification at address:
0x{0:x8}!", apiResult));
}
}
hr = pp.FL_ExecCmsisApiUnInit(fncCodeVerify, timeOut, out apiResult, out
strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("UnInit operation failed!");
}
hr = pp.FL_ReleaseTarget(false, out strError);
if (!SUCCEEDED(hr))
{
MessageBox.Show("MCU release alter flash programming algorithms execution
failed!");
}
```

FL_ReleaseTarget(bool IN restoreRAM, string OUT strError)

Releases CPU after flash algorithm execution. Uses as the final step in flashloader APIs sequence. *restoreRAM* parameter is reserved, not implemented yet.

C#:

```
int FL_ReleaseTarget(bool restoreRAM, out string strError);
string strError;
int hr = pp.FL_ReleaseTarget(false, out strError);
if  (SUCCEEDED(hr))
{
MessageBox.Show("MCU has been released after programming algorithms execu-
tion!");
}
else
{
MessageBox.Show("MCU release alter flash programming algorithms execution
failed!");
}
```

PSoC6_WriteRow(IN rowID, nvector IN data, string OUT strError)

Writes *data* to *rowID* of a flash array. The row must be unprotected and may contain any data.

C#:

```
int PSoC6_WriteRow(int rowID, object data, out string strError);
string strError;
int rowID = 0x00;
byte[] data = new byte[128];
for (int i = 0; i < data.Length; i++) data[i] = (byte)i;
int hr = pp.PSoC6_WriteRow(rowID, data, out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Row " + rowID + " Programmed Successfully!");
else
MessageBox.Show(strError);
```

PSoC6_ProgramRow (IN rowID, nvector IN data, string OUT strError)

This API programs *data* to *row*. The row must be unprotected and erased.

C#:
```
int PSoC6_ProgramRow(int rowID, object data, out string strError);
string strError;
int rowID = 0;
byte[] data = new byte[128];
for (int i = 0; i < data.Length; i++) data[i] = (byte)i;
int hr = pp.PSoC6_ProgramRow(rowID, data, out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Row " + rowID + " Programmed Successfully!");
else
MessageBox.Show(strError);
```

PSoC6_EraseAll(string OUT strError)

Erases all flash content including security rows and chip-level protection.

C#:
```
int PSoC6_EraseAll(out string strError);
string strError;
int hr = pp.PSoC6_EraseAll(out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Flash erased successfully!");
else
MessageBox.Show("Failed: " + strError);
```

PSoC6_CheckSum(IN rowID, OUT checksum, string OUT strError)

Calculates the checksum for *rowID*. If *rowID* = 0xFFFFFFFF, the checksum is calculated for the whole flash (privileged and user rows).

C#:
```
int PSoC6_CheckSum(int rowID, out int checksum, out string strError);
string strError = "";
int rowID = -1;
int checksum;
hr = pp.PSoC6_CheckSum(rowID, out checksum, out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Checksum: " + checksum.ToString("X08"));
```

PSoC6_WriteProtection(byte IN lifeCycle, nvector IN secureRestrict, nvector IN deadRestrict, bool IN voltageVerification, string OUT strError)

Includes the Write Chip Protection, Secure, and Dead Access Restriction properties. For programming eFuse, you need 2.5 V. If voltageVerification properties has true check, voltage will be executed. If voltageVerification properties has false check is ignore. Each byte of secureRestrict and deadRestrict arrays maps to a single eFuse bit.

- 0x01 – blow eFuse
- 0x00 – do not blow eFuse
- 0xFF - ignore

C#:
```
int PSoC6_WriteProtection(byte lifeCycle, byte[] secureRestrict,
byte[] deadRestrict, bool voltageVerification, out string strError)
string strError;
byte[]secureRestrict = new byte[16];
```

```
byte[]deadRestrict = new byte[16];
bool voltageVerification = false;
byte lifecycle = 1;

for (int i = 0; i < secureRestrict.Length; i++)
secureRestrict [i] = 0xFF;

for (int i = 0; i < deadRestrict.Length; i++)
deadRestrict [i] = 0xFF;

int hr = pp.PSoC6_WriteProtection(lifecycle, secureRestrict, deadRestrict,
voltageVerification, out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Protection Data written successfully!");
else
MessageBox.Show(strError);
```

PSoC6_ReadRow(IN rowID, nvector OUT data, string OUT strError)

Reads the content of a flash row. The row must be unprotected and chip-level protection must be in the OPEN state.

C#:

```
int PSoC6_ReadRow(int rowID, out object data, out string strError);
string strError;
int rowID = 0;
object data;
int hr = pp.PSoC6_ReadRow(rowID, out data, out strError);
if (SUCCEEDED(hr))
{
string msg = "Row ID " + rowID + "\r\n";
byte[] rowData = data as byte[];
for (int i = 0; i < rowData.Length; i++)
msg += rowData[i].ToString("X2") + " ";
MessageBox.Show(msg);
}
else
MessageBox.Show(strError);
```

PSoC6_ReadProtection(byte OUT chipProtect, string OUT strError)

Reads chip-level protection.

C#:

```
int PSoC6_ReadProtection(out byte chipProtect, out string strError);
string strError = "";
byte chipProtect;
int hr = pp.PSoC6_ReadProtection(out chipProtect, out strError);
if (SUCCEEDED(hr))
{
MessageBox.Show(string.Format("Chip protection data: {0}",chipProtect));
}
else
MessageBox.Show(strError);
```

PSoC6_ProtectAll(bool IN voltageVerification, bool OUT *treatErrorAsWarning, string OUT strError)

Writes chip-level protection using data from hex file. EFuse should be programmed with 2.5 V. If voltageVerification is set to true, voltage check will be done.

C#:
```
int PSoC6_ProtectAll(bool voltageVerification, out bool treatErrorAs-
Warning, out string strError);
string strError;
bool treatErrorAsWarning;
int hr = pp.PSoC6_ProtectAll(false, out treatErrorAsWarning, out
strError);
if (SUCCEEDED(hr)) {
MessageBox.Show("Chip is protected!");
} else {
if (treatErrorAsWarning) {
//This case should be treated as SUCCESS
hr = 0; // resetting hr
//Report warning and proceed to next steps
MessageBox.Show("Warning: "+strError);
} else {
MessageBox.Show("Failed: "+strError);
}
}
```

PSoC6_GetFlashInfo(string OUT rowsPerFlash, OUT rowSize, string strError)

Returns the flash characteristics of the acquired device.

C#:
```
int PSoC6_GetFlashInfo(out int rowsPerFlash, out int rowSize, out
string strError);
string strError;
int hr = pp.PSoC6_GetFlashInfo(out rowsPerFlash, out rowSize, out
strError);
if (SUCCEEDED(hr)){
string msg = "Info about flash:";
msg += "\r\nRows Per Flash: " + rowsPerFlash;
msg += "\r\nRow Size : " + rowSize;
MessageBox.Show(msg);
}
else
MessageBox.Show(strError);
```

PSoC6_ProgramRowFromHex(IN hexRowID, string OUT strError)

Programs data from row id in the hex file into the flash array. The flash must be unprotected and erased.

C#:
```
int PSoC6_ProgramRowFromHex(int hexRowID, out string strError);
string strError;
int rowID = 0x00;
int hr = pp.PSoC6_ProgramRowFromHex(rowID, out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Row " + rowID + " Programmed Successfully!");
```

```
else
MessageBox.Show(strError);
```

PSoC6_VerifyRowFromHex(IN hexRowID, OUT verResult, string OUT strError)

Verifies a row of flash using data from the hex file. Flash must be unprotected.

C#:
```
int PSoC6_VerifyRowFromHex(int hexRowID, out int verResult, out string
strError);
string strError;
int rowID = 5;
int verResult = 0;
int hr = pp.PSoC6_VerifyRowFromHex(rowID, out verResult, out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Verfication Result: " + verResult + " -> " +
(verResult == 0 ? "Failed" : "Passed"));
else
MessageBox.Show(strError);
```

PSoC6_GetSiliconID(nvector OUT siliconID, OUT familyIdHi, OUT familyIdLo, OUT revisionIdMaj, OUT revisionIdMin, OUT siliconIdHi, OUT siliconIdLo, OUT sromFmVersionMaj, OUT sromFmVersionMin, OUT protectState, string OUT strError)

Reads the silicon ID from a PSoC 6 device in SWD mode. This function can be called after device is successfully acquired by the DAP_Acquire() functions. The returned vector is four bytes in length:

- siliconID[0] - high byte of PSoC silicon ID
- siliconID[1] - low byte of PSoC silicon ID
- siliconID[2] - Revision ID of the PSoC device
- siliconID[3] - Family ID of the PSoC device

C#:
```
int PSoC6_SiliconID(out object SiliconID, out int familyIdHi, out int fami-
lyIdLo, out int revisionIdMaj, out int revisionIdMin, out int siliconIdHi, out int
siliconIdLo, out int sromFmVersionMaj, out int sromFmVersionMin, out int protect-
State, out string strError);
string strError;
object siliconID;
int familyIdHi;
int familyIdLo;
int revisionIdMaj;
int revisionIdMin;
int siliconIdHi;
int siliconIdLo;
int sromFmVersionMaj;
int sromFmVersionMin;
int protectState;
int hr = pp.PSoC6_GetSiliconID(out siliconID, out familyIdHi, out family-
IdLo, out revisionIdMaj, out revisionIdMin, out siliconIdHi, out siliconIdLo,
out sromFmVersionMaj, out sromFmVersionMin, out protectState, out
strError);
if (SUCCEEDED(hr)) {
```

```
string msg = "Silicon ID: ";
byte[] id = siliconID as byte[];
for (int i = 0; i < id.Length; i++)
msg += id[i].ToString("X2") + " ";

msg = string.Format("{0}{1}familyIdHi-{2}{1} familyIdLo -{3}{1} revision-
IdMaj -{4}{1} revisionIdMin -{5}{1} siliconIdHi -{6}{1} siliconIdLo -
{7}{1} sromFmVersionMaj -{8}{1} sromFmVersionMin -{9}{1} protectState -
{10}",msg, "/n", familyIdHi, familyIdLo, revisionIdMaj, revisionIdMin, silico-
nIdHi, siliconIdLo, sromFmVersionMaj, sromFmVersionMin, protectState);

MessageBox.Show(msg);
}
else
MessageBox.Show(strError);
```

PSoC6_WriteRowFromHex (IN hexRowID, string OUT strError)

Writes data from row id in hex file into the flash array. The row must be unprotected and may contain any data.

C#:
```
int PSoC6_WriteRowFromHex(int hexRowID, out string strError);
string strError;
int rowID = 0x00;
int hr = pp.PSoC6_WriteRowFromHex(rowID, out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Row " + rowID + " Programmed Successfully!");
else
MessageBox.Show(strError);
```

PSoC6_EraseRow(int IN rowAddr, string OUT strError)

Erases row at given address.

C#:
```
int PSoC6_EraseRow(int rowAddr, out string strError);
string strError;
int hr = pp. PSoC6_EraseRow(0x10000000, out strError);
if (SUCCEEDED(hr)) {
MessageBox.Show("Erased row at address 0x10000000");
} else {
MessageBox.Show("Failed: "+strError);
}
```

HEX_GetRowAddress (IN hexRowID, OUT rowAddr, string OUT strError)

Gets address in flash for current row in hex file.

C#:
```
int HEX_GetRowAddress (int hexRowID, out int rowAddr, out string
strError);
string strError;
int hexRowID = 0x00;
int rowAddr;

int hr = pp.HEX_GetRowAddress (hexRowID, out rowAddr, out strError);
if (SUCCEEDED(hr))
```

```
MessageBox.Show("Row in hex" + hexRowID + " has address " + rowAddr);
else
MessageBox.Show(strError);
```

HEX_GetRowsCount (OUT rowsPerHex)

Gets the count of rows in hex file.

C#:
```
int HEX_GetRowsCount (out int rowsPerHex, out string strError);
string strError;
int rowsCount;

int hr = pp.HEX_GetRowsCount (out rowsCount, out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Hex file contains " + rowsCount + " rows.");
else
MessageBox.Show(strError);
```

DAP_JTAGtoSWD(string OUT strError)

Switches DAP with JTAG to SWD.

C#:
```
int DAP_JTAGtoSWD(out string strError);
string strError;
int hr = pp. DAP_JTAGtoSWD(out strError);
if (SUCCEEDED(hr))
MessageBox.Show("DAP switch with JTAG to SWD!");
else
MessageBox.Show("Failed: " + strError);
```

DAP_SWDtoJTAG(string OUT strError)

Switches DAP with JTAG to SWD.

C#:
```
int DAP_SWDtoJTAG(out string strError);
string strError;
int hr = pp.DAP_SWDtoJTAG(out strError);
if (SUCCEEDED(hr))
MessageBox.Show("DAP switch with SWD to JTAG!");
else
MessageBox.Show("Failed: " + strError);
```

DAP_ JTAGtoDS(string OUT strError)

Switches DAP with JTAG to DS.

C#:
```
int DAP_JTAGtoDS(out string strError);
string strError;
int hr = pp.DAP_JTAGtoDS(out strError);
if (SUCCEEDED(hr))
MessageBox.Show("DAP switch with JTAG to DS!");
else
MessageBox.Show("Failed: " + strError);
```

DAP_ SWDtoDS (string OUT strError)

Switches DAP with SWD to DS.

C#:
```
int DAP_SWDtoDS(out string strError);
string strError;
int hr = pp.DAP_SWDtoDS(out strError);
if (SUCCEEDED(hr))
MessageBox.Show("DAP switch with SWD to DS!");
else
MessageBox.Show("Failed: " + strError);
```

DAP_DStoSWD(string OUT strError)

Switches DAP with DS to SWD.

C#:
```
int DAP_DStoSWD(out string strError);
string strError;
int hr = pp.DAP_DStoSWD(out strError);
if (SUCCEEDED(hr))
MessageBox.Show("DAP switch with DS to SWD!");
else
MessageBox.Show("Failed: " + strError);
```

DAP_DStoJTAG(string OUT strError)

Switches DAP with DS to SWD.

C#:
```
int DAP_DStoJTAG(out string strError);
string strError;
int hr = pp.DAP_DStoJTAG(out strError);
if (SUCCEEDED(hr))
MessageBox.Show("DAP switch with DS to JTAG!");
else
MessageBox.Show("Failed: " + strError);
```

ReadBlock(IN blockID, nvector OUT data, OUT sscResult, string OUT strError)

Reads one block of flash defined by *blockID* in the current bank. The *BlockID* parameter is in the range [0..BlocksPerBank-1].

C#:
```
int ReadBlock(int blockID, out object data, out int sscResult, out
string strError);
object readData;
int sscResult;
string strError;
int hr = pp.ReadBlock(1, out readData, out sscResult, out strError);
if (SUCCEEDED(hr))
{
 string str = "--->ReadBlock completed successfully. sscResult = " +
  sscResult.ToString("X2");
    str+="\r\n";
    byte[] data = readData as byte[];
    for (int i = data.GetLowerBound(0);
```

```
 i <= data.GetUpperBound(0); i++)
        str += data[i].ToString("X2");
    MessageBox.Show(str);
}
else
    MessageBox.Show(strError);
```

ReadBlock1(IN bank, IN blockID, nvector OUT data, OUT sscResult, string OUT strError)

Reads one block of flash defined by *blockID* in the bank set by the *bank* parameter. The *BlockID* parameter is in the range [0..BlocksPerBank–1].

C#:

```
int ReadBlock1(int bank, int blockID, object out data,
out int sscResult, out string strError);
object readData;
int sscResult;
string strError;
int hr = pp.ReadBlock1(3, 127, out readData, out sscResult,
out strError); //bank=0, block=1
if (SUCCEEDED(hr))
{
  string str = "--->ReadBlock completed successfully. sscResult = "+
sscResult.ToString("X2");
    str += "\r\n";
    byte[] data = readData as byte[];
    for (int i = data.GetLowerBound(0);
 i <= data.GetUpperBound(0); i++)
        str += data[i].ToString("X2");
    MessageBox.Show(str);
}
 else
    MessageBox.Show(strError);
```

ReadIO(IN addr, IN size, nvector OUT data, string OUT strError)

Reads *size* bytes from the I/O registers of the PSoC device, starting at the *addr* address. The device must have previously been acquired using the Acquire function.

C#:

```
int ReadIO(int addr, int size, out object data, out string strError);

object readIoData;
string strError;
int hr = pp.ReadIO(0, 10, out readIoData, out strError);
if (SUCCEEDED(hr))
{
    byte[] data = readIoData as byte[];
    string s = "PSoC IO registers: \r\n";
    for (int i = 0; i < 10; i++) s += data[i].ToString("X2") + " ";
    MessageBox.Show(s);
}
else
```

```
MessageBox.Show(strError);
```

ReadProtection(IN bank, nvector OUT block, OUT sscResult, string OUT strError)

Reads a block of protection area from *bank*.. Before using this function, the correct PSoC device should be set either by the Acquire() function or the GetSiliconId() function. If the PSoC device has no banks, you should set this parameter to 0.

C#:

```
int ReadProtection(int bank, out object block,
 out int sscResult, out string strError);

object readData;
int sscResult;
string strError;
int hr = pp.ReadProtection(3,out readData,out sscResult,
    out strError);
if (SUCCEEDED(hr))
{
    byte[] data = readData as byte[];
    string s = "Protection Area: sscResult = "+sscResult +"\r\n";
    for (int i = data.GetLowerBound(0);
 i <= data.GetUpperBound(0); i++)
        s += data[i].ToString("X2") + " ";
    MessageBox.Show(s);
}
else
    MessageBox.Show(strError);
```

ReadRAM(IN addr, IN size, nvector OUT data, string OUT strError)

Reads *size* bytes starting at *addr* from the SRAM of the PSoC device. To get the correct data from PSoC, you must acquire the device using the Acquire() function. Note that this function operates on the current SRAM bank.

C#:

```
int ReadRAM(int addr, int size, out object data, out string
strError);

object readRamData;
string strError;
int hr = pp.ReadRAM(0, 10, out readRamData, out strError);
if (SUCCEEDED(hr))
{
    byte[] data = readRamData as byte[];
    string s = "PSoC SRAM area: \r\n";
    for (int i = 0; i < 10; i++) s += data[i].ToString("X2") + " ";
    MessageBox.Show(s);
}
else
 MessageBox.Show(strError);
```

ReleaseChip(string OUT strError)

Powers the chip off, and leaves the Test mode of PSoC. If the chip is acquired in Reset mode, it is reset using the XRES pin. You can use the SetAutoReset() function to disable reset if necessary.

C#: `int ReleaseChip(out string strError);`

```
string strError;
int hr = pp.ReleaseChip(out strError);
if (SUCCEEDED(hr))
                MessageBox.Show("Chip Released Successfully!");
            else
                MessageBox.Show(strError);
```

SetAcquireMode(string IN mode, string OUT strError)

Sets the acquire mode. *mode* can be Power, Reset, or PowerDetect.

C#: `int SetAcquireMode(string mode, out string strError);`

```
string strError;
int hr = pp.SetAcquireMode("Power", out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("The Acquire mode set Successfully!");
else
    MessageBox.Show(strError);
```

SetAutoReset(IN mode, string OUT strError)

Defines whether the chip will be reset during release operation by ReleaseChip() ot DAP_Release-Chip() functions. This option is applicable only if the chip was acquired in Reset mode. The *mode* parameter can be one of: 0x00 – autoreset disabled, 0x01 – autoreset enabled.

C#:
```
int SetAutoReset(int mode, out string strError);
string strError;
int hr = pp.SetAutoReset(0x01, out strError);
```

SetBank(IN bank, string OUT strError)

Sets the bank of the flash for the next operations.

C#: `int SetBank(int bank, out string strError);`

```
string strError;
int hr = pp.SetBank(1, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("The Flash Bank set Successfully!");
else
    MessageBox.Show(strError);
```

SetChipType(IN familyID, string OUT strError)

To acquire devices from certain families with the ICE programmer (CY7C602xx, CY7C633xx, CY7C638xx, and CY7C639xx), you must first set the chip type to value = 4. For other programmers/ families, this function can be ignored. The meaning of *familyID* parameter is described in the GetDeviceInfo() function.

C#.
```
int SetChipType(int familyID, out string strError);

string strError;
int hr = pp.SetChipType(4, out strError); //enCoRe
if (!SUCCEEDED(hr))
{
    MessageBox.Show(strError);
}
```

SetPowerVoltage(string IN voltage, string OUT strError)

Sets the voltage applied when PowerOn() is called or when acquiring with power cycle.

Some programmers apply only specific volages. For example, ICE-4000, MiniProg1, FirstTouch work with only a 5.0 V power source. ICE-cube supplies either 3.3 V or 5.0 V to the target device. If you set an unsupported voltage to the programmer, this function is ignored.

C#:
```
int SetPowerVoltage(string voltage, out string strError);

string strError;
int hr = pp.SetPowerVoltage("3.3", out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("The Voltage applied Successfully!");
else

    MessageBox.Show(strError);
```

SetProtocol(enumInterfaces IN protocol, string OUT strError)

Activates a protocol for the programmer. The possible protocols are listed below. This API also lists the names of the supported protocols by the active programmer call function.

The MiniProg3 can be configured for TX8 (UART) mode data receiving. This requires that set the mode to SWD_SWV and use the SWV_Setup() API for configuration.

```
public enum enumInterfaces
      {
       JTAG = 1,
       ISSP = 2,
       I2C = 4,
       SWD = 8,
       SPI = 16,
       SWD_SWV = 32
       }
```

C#:
```
int SetProtocol(enumInterfaces protocol, out string strError);
```

```
string strError;
            enumInterfaces protocol = enumInterfaces.ISSP;
int hr = pp.SetProtocol(protocol, out strError);
if (SUCCEEDED(hr))
      MessageBox.Show("Protocol set successfully to " + protocol);
    else
      MessageBox.Show("Failed to set protocol: "+strError);
```

SetProtocolClock(enumFrequencies IN clock, string OUT strError)

Sets bus frequency for the active protocol. Currently, this function is applicable only to the MiniProg3 device in SWD/JTAG mode. The following code sample shows possible values for frequency:

```
public enum enumFrequencies
    {
      FREQ_48_0 = 0,
      FREQ_24_0 = 4,
      FREQ_16_0 = 16,
      FREQ_03_2 = 24,
      FREQ_06_0 = 96,
      FREQ_12_0 = 132,
      FREQ_08_0 = 144,
      FREQ_01_6 = 152,
      FREQ_01_5 = 192,
      FREQ_03_0 = 224,
      FREQ_RESET = 252,
}
```

C#:
```
int SetProtocolClock(enumFrequencies clock,
      out string strError);
string strError;
            enumFrequencies clock = enumFrequencies.FREQ_03_0;
int hr = pp.SetProtocolClock(clock, out strError);
if (SUCCEEDED(hr))
      MessageBox.Show("Protocol Clock set successfully to " + clock);
    else
      MessageBox.Show("Failed to set protocol clock: " + strError);
```

SetProtocolConnector(IN connector, string OUT strError)

Activates *connector* of the programmer. This function is applicable to the MiniProg3 device in SWD mode and for TrueTouch Bridge in I^2C. The possible values for MiniProg3 in SWD mode are:

- 0 – 5-pin connector
- 1 – 10-pin connector

In I^2C mode for TrueTouch Bridge, this function selects a pair of I^2C pins on the bridge header. The possible values are:

- 0 – original I^2C - Pins 6, 8
- 1 – ISSP: Clk, Data - Pins 9, 7
- 2 – SPI: Clk, Data - Pins 13,15

```
C#: int SetProtocolConnector(int connector, out string strError);
string strError;
int connector = 1; // 0 - 5 pin, 1 - 10 pin
int hr = pp.SetProtocolConnector(connector, out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("Connector set successfully to " +
   ((connector==1)?"10-pin":"5-pin"));
        else
     MessageBox.Show("Failed to set connector: " + strError);
```

SPI_ConfigureBus(enumSpiBitOrder IN bitOrder, enumSpiMode IN mode, IN frequency, nvector IN extra, string OUT strError)

Sets configuration of the SPI master. This function should be called after SPI mode has been set using SetProtocol().

frequency (Hz) - one of the frequencies returned by SPI_GetSupportedFreq().

extra = array of bytes. The zero element of this array is used to set the EzI2C SPI interface. The possible values are: Gen3 - 0 and Gen4 - 1.

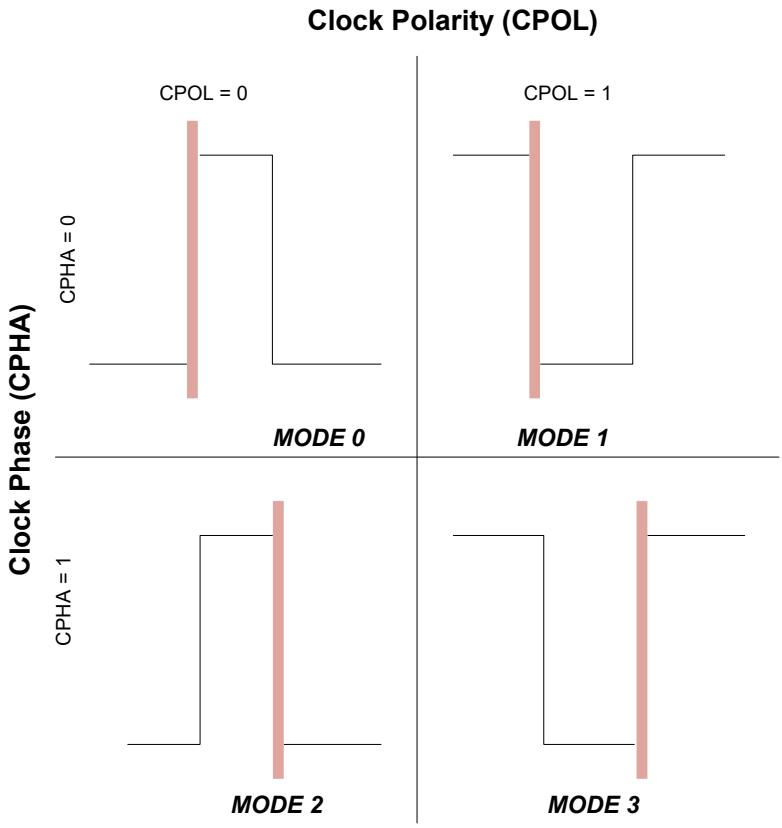
For proper communication with the slave, you must use SPI mode. The master must use the same clock polarity (CPOL) and phase (CPHA) as the slave. The *mode* parameter allows you to define the necessary SPI communication mode.

It is possible to configure the four modes using CPOL and CPHA values, as shown in the following table.

Table 2-5. SPI Mode Configuration using CPOL and CPHA

Mode	CPOL	CPHA
0	0	0
1	1	0
2	0	1
3	1	1

The timing diagram for all the modes is shown here.



At CPOL = 0 normal clock mode.

- For CPHA = 0, data is captured on the clock's rising edge (low¨high transition), and data is propagated on a falling edge (high¨low clock transition).
- For CPHA = 1, data is captured on the clock's falling edge, and data is propagated on a rising edge.

At CPOL = 1 inverse clock mode.

- For CPHA = 0, data is captured on clock's falling edge, and data is propagated on a rising edge.
- For CPHA = 1, data is captured on clock's rising edge, and data is propagated on a falling edge

```
public enum enumSpiBitOrder
{
MSB = 0,
LSB = 1
}
public enum enumSpiMode
{
Mode_00 = 0,
Mode_01 = 1,
Mode_02 = 2,
Mode_03 = 3
}
```

C#:
```
int SPI_ConfigureBus(enumSpiBitOrder bitOrder, enumSpiMode mode,
int freq, object extra, out string strError);
string strError;
int hr = pp.SPI_ConfigureBus(enumSpiBitOrder.MSB, enumSpiMode.Mode_00,
1000000, null, out strError);
if (SUCCEEDED(hr))
     MessageBox.Show("Successfully configured SPI-master");
else
     MessageBox.Show("Failed! " + strError);
```

SPI_DataTransfer(IN mode, nvector IN dataIN, nvector OUT dataOUT, string OUT strError)

Generates a transaction on the SPI bus.

mode - bitmask that controls the state of the SS line during transaction. Mask 0x02 - assert SS, 0x08 - deassert SS.

dataIN - data to be sent to the slave,

dataOUT - data received from the slave. Size of *dataOUT* is equal to the size of *dataIN*.

C#:
```
int SPI_DataTransfer(int mode, object dataIN, out object dataOUT, out
string strError);
string strError;
byte[] dataIN = new byte[] { 0x20, 0x01, 0x00, 0x02,0xFF, 0x03, 0x00,
0x04, 0xFF};
object dataOUT;
int hr = pp.SPI_DataTransfer(0x0A, dataIN, out dataOUT,
out strError);
if (SUCCEEDED(hr))
{
      string strSpi = "SPI Transaction succeeded:\r\nSent Data";
      byte[] readData = dataOUT as byte[];
      for (int i = 0; i < dataIN.Length; i++)
      strSpi += " "+dataIN[i].ToString("X2");
      strSpi += "\r\nRead Data:";
      for (int i = 0; i < readData.Length; i++)
      strSpi += " "+readData[i].ToString("X2");
      MessageBox.Show(strSpi);
}
   else
      MessageBox.Show("Failed! "+strError);
```

SPI_GetSupportedFreq (nvector OUT freq, string OUT strError)

Returns a set of frequencies that the master can generate. Frequencies are expressed in Hz. One of these frequencies should be used in the SPI_ConfigureBus() function for correct bus initialization.

C#:
```
int SPI_GetSupportedFreq(out object freq, out string strError);
object freqs;
string strError;
pp.SetProtocol(enumInterfaces.SPI, out strError);
```

```
int hr = pp.SPI_GetSupportedFreq(out freqs, out strError);
if (SUCCEEDED(hr))
{
        string strFreqs = "Supported Frequencies (Hz): \r\n";
        int[] data = freqs as int[];
        for (int i = 0; i < data.Length; i++)
{
        strFreqs += data[i].ToString()+"\t";
        if ((i+1) % 10 == 0) strFreqs += "\r\n";
}
        MessageBox.Show(strFreqs);
}
     else //Operation Failed
{
        MessageBox.Show(strError, "Error!", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
```

_StartSelfTerminator(int IN clientProcessID, int OUT serverProcessID)

Automatically releases all resources of the COM object if the client application is terminated abnormally. As a rule, if the client application finishes in a normal way then the associated COM object is freed automatically. You should call this function immediately after you start the COM object. There is no reason to call this function more than once; if the thread is already started, the function returns immediately.

This function returns the process ID of the COM object. In some cases, this ID can be used to terminate the COM object forcibly during a long operation.

C#: `int _StartSelfTerminator(int clientProcessID);`

```
int serverProcessID, clientProcessID;
clientProcessID = System.Diagnostics.Process.GetCurrentProcess().Id;
serverProcessID = pp._StartSelfTerminator(clientProcessID);
```

SWDIOR(IN addr, OUT data)

Reads the content of the register of a PSoC 3, PSoC 4, or PSoC 5 device using the SWD interface.

C#:

```
//PSoC4 example
int dta;
int hr = com313.swdior(0x20000000, out dta);
if (!SUCCEEDED(hr))
     MessageBox.Show("Failed to read content of the register!");
 else
     hr = com313.swdiow(0x20000000, 0x12345678);
 if (!SUCCEEDED(hr))
     MessageBox.Show("Failed to write data to the register!");
 else
     hr = com313.swdior(0x20000000, out dta);
 if (!SUCCEEDED(hr))
    MessageBox.Show("Failed to read content of the register!");
```

SWDIOR_RAW(nvector IN input, nvector OUT output)

Performs one read transaction on the SWD bus.

input - 1-byte vector defining the APnDP register's area (bit 2, mask 0x04) and 2-bit register address (bits 1:0, mask 0x03).

output - 5-byte vector, where the first byte is the status of the transaction, and the other four bytes are read from the register.

You should consider only three LSB bits of the status byte. They are in the raw format of the SWD transaction:

b'001 - ACK

b'010 - WAIT

b'100 - FAULT

Other combinations can be considered as NACK (especially b'111, which means the device does not respond at all).

C#:
```
//PSoC4 example
byte[] dataIN = new byte[1];
byte[] dataOUT;
dataIN[0] = 0x00;
int hr = com313.swdior_raw(dataIN, out dataOUT);
    if (!SUCCEEDED(hr))
      MessageBox.Show("Failed to read transaction on SWD-bus!");
    else
      hr = com313.swdiow_raw(dataIN, out dataOUT);
    if (!SUCCEEDED(hr))
      MessageBox.Show("Failed to write transaction on SWD-bus!");
```

SWDIOW(IN addr, IN data)

Writes *data* to the register of the PSoC 3, PSoC 4, or PSoC 5 device using the SWD interface.

To see an example, see *SWDIOR(IN addr, OUT data)*.

SWDIOW_RAW(nvector IN input, nvector OUT output)

Performs one write transaction on the SWD bus.

input - 5-byte array with 1-byte header and 4-byte data. The header defines the APnDP register's area (bit 2, mask 0x04) and register address (bits 1:0, mask 0x03).

output - 1-byte vector containing the status of the transaction.

For a description of the Status byte, see SWDIOR_RAW () API.

To see an example, see *SWDIOR_RAW(nvector IN input, nvector OUT output)*.

SWD_LineReset(string OUT strError)

Resets and reinitializes the SWD bus by sending >=50 clock bits on SWDCLK line, which keeps SDATA high.

```
C#: int SWD_LineReset(out string strError);
string strError;
int hr = pp.SWD_LineReset(out strError);
if (!SUCCEEDED(hr))
   MessageBox.Show("Failed to reset SWD line!");
   else
//Initialize SWD-device by reading silicon ID
byte[] dataIN = new byte[1], data;
object dataOUT;
dataIN[0] = 0x00;
hr = pp.swdior_raw(dataIN, out dataOUT);
data = dataOUT as byte[];
if (!SUCCEEDED(hr) || ((data[data.Length - 1] & 0x01) != 0x01))
        MessageBox.Show("Failed to read silicon ID!");
   else
        MessageBox.Show("Silicon ID read successful!");
```

SWV_ReadData(nvector OUT dataOUT, string OUT strError)

Allows the client to retrieve SWV data received from the slave device. This method is expected to be continuously called by the client to check whether new data is available.

The *dataOUT* array contains received SWV data generated since the last request. If there no new SWV data, the API fails and *dataOUT* is empty.

Only MiniProg3 supports this method. Note that the SWV protocol must be set in the programmer before using this method. Use SetProtocol() to set the SWV protocol and SWV_Setup() to configure frequency and encoding type (UART or Manchester) of data transfer.

```
C#: int SWV_ReadData(out object dataOUT, out string strError);
long hr;
object dataOUT;
string strError;
hr = pp.SWV_ReadData(out dataOUT, out strError);
if (!SUCCEEDED(hr))
   MessageBox.Show("Failed to read data from SWV line!");
else
{
byte[] packet = dataOUT as byte[];
string str = "";
for (int i = 0; i < packet.Length; i++)
str += packet[i].ToString("X2") + " ";
   MessageBox.Show("Received data id: " + str);
}
```

SWV_Setup(enumSWVMode IN mode, IN targetFreq, nvector IN extra, string OUT strError)

Configures the MiniProg3 for SWV data sampling by allowing the MiniProg3 to handle multiple different SWV data modes, and multiple bus frequencies.

Only MiniProg3 supports this method. The SWV mode must be set by SetProtocol() before you use this method. The MiniProg3 supports SWV data sampling on both the 5-pin and 10-pin connectors. The SWV input uses the XRES line on the 5-pin header and the TDO line for the 10-pin header.

mode - specifies whether to use Manchester (0x02) encoding or UART (0x01).

targetFreq – the frequency on which the slave device generates SWV traffic. Must be in range 188.235 kHz – 24 MHz for UART and 188.235 kHz – 12 MHz for Manchester. This parameter is passed in Hz.

extra – Reserved for future use.

```
enum enumSWVMode
{
TX8 = 0x01,
MANCHESTER = 0x02
} enumSWVMode;
```

C#:
```
int SWV_Setup(enumSWVMode mode, int targetFreq, object extra, out
string strError);
int hr = 0;
string strError;
enumSWVMode enumMode = Protocol.UART;
hr = pp.SWV_Setup(enumMode, 230400, null, out strError);
if (!SUCCEEDED(hr))
    MessageBox.Show("Failed to setup SWV configuration!");
else
    MessageBox.Show("SWV configured!");
```

TableRead(IN tableID, OUT xa, nvector OUT table, OUT sscResult, string OUT strError)

Does a TableRead supervisory operation (see the Technical Reference Manual). The *tableID* is the number of the table to read. If the function succeeds, then *table* is an 8-byte vector with table content, and *xa* contains values from the X and A registers of the PSoC device after the supervisory operation is finished. The PSoC device must be acquired by the programmer before this function can complete successfully.

C#:
```
int TableRead(int tableID, out int xa, out object table,
  out int sscResult);

int xa,sscResult;
object table;
string strError;
int hr = pp.TableRead(0, out xa, out table, out sscResult,
    out strError);
if (SUCCEEDED(hr)) {
    string s = "Table Read result: \r\n";
    byte x, a;
    x = (byte)((xa >> 8) & 0xFF);
    a = (byte)(xa & 0xFF);
    s+="X="+x.ToString("X2")+"  A="+a.ToString("X2")+"\r\n";
    byte[] data = table as byte[];
    for (int i = data.GetLowerBound(0);
             i<=data.GetUpperBound(0); i++)
        s+=data[i].ToString("X2")+" ";
    MessageBox.Show(s);
}
```

```
  else
    MessageBox.Show(strError);
```

TestClock(IN numberOfClocks, string OUT strError)

Sends *numberOfClocks* through the SCLK line to the acquired PSoC device.

C#:

```
int TestClock(int number, out string strError);

string strError;
int hr = pp.TestClock(11, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("TestClock() is succeeded!");
  else
    MessageBox.Show(strError);
```

ToggleReset(IN polarity, IN duration, string OUT strError)

Generates a reset signal on XRES line depending on polarity and duration parameters.

Where, polarity: XRES_POLARITY_NEGATIVE = 0x00;

XRES_POLARITY_POSITIVE = 0x01.

Duration = a length of reset signal, specifies in milliseconds.

C#:

```
int ToogleReset(long polarity, long duration, out string strError);
string strError;
long polarity = 0x01, duration = 0x05;
int hr = pp.ToogleReset(polarity, duration, out strError);
if (SUCCEEDED(hr))
   MessageBox.Show("Reset signal generate successful!");
  else
   MessageBox.Show("Failed to generate reset signal!");
```

UpdateProgrammer(string IN arguments, string OUT strError)

Updates the programmer to the version of firmware from the home (install) directory of PSoC Programmer. The current implementation of this function does not use the *arguments* parameter. Note, this is a blocking (sync) function. The client application wait until the function is finished (succeeded or failed). You should call this function only if a port is opened.

C#:

```
int UpdateProgrammer(string arguments, out string strError);

string strError;
int hr = pp.UpdateProgrammer("", out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Programmer is Updated Successfully!");
  else
    MessageBox.Show(strError);
```

UpdateProgrammer1(IN cmd, string IN arguments, int OUT blockNo, string OUT strError)

Allows you to manually drive the UpdateProgrammer process. You must use a strictly defined algorithm to update your programmer manually. This function allows you to track the status of the update. The *cmd* parameter defines the current phase of the update. The following table shows all possible values.

INITIALIZE	0x00
UPGRADE_BLOCK	0x01
VERIFY_BLOCK	0x02
FINALIZE	0x03

An example of this function is as follows. Typically, this algorithm is implemented by the UpdateProgrammer() function.

C#:
```
int UpdateProgrammer1(int cmd, string arguments,
 out int blockNo, out string strError);

int r;
int totalBlocks, blockNo;
string strError;
string sLog = "Firmware Update started...\r\n";
r = pp.UpdateProgrammer1((int)enumUpgradeFirmware.INITIALIZE, "",
  out totalBlocks, out strError);
if (!SUCCEEDED(r))
{
    sLog += "\r\nERROR! "+ strError;
    goto ShowResult;
}
sLog += "Total Blocks to Upgrade = " + totalBlocks+"\r\n";
sLog += "Programming: \r\n";
for (int i = 0; i < totalBlocks; i++)
{
    r = pp.UpdateProgrammer1((int)enumUpgradeFirmware.UPGRADE_BLOCK,
      "", out blockNo, out strError);
    if (!SUCCEEDED(r))
    {
        sLog += "\r\nERROR! " + strError;
        pp.UpdateProgrammer1((int)enumUpgradeFirmware.FINALIZE, "",
         out blockNo, out strError);
        goto ShowResult;
    }
    sLog += blockNo.ToString("X2") + " ";
 //Here Progress Bar of Upgrading can be refreshed for "i-th" block
    Application.DoEvents();
}
sLog += "\r\nVerification: \r\n";
for (int i = 0; i < totalBlocks; i++)
{
```

```
    r = pp.UpdateProgrammer1((int)enumUpgradeFirmware.VERIFY_BLOCK,
 "", out blockNo, out strError);
    if (!SUCCEEDED(r))
    {
        sLog += "\r\nERROR! " + strError;
        pp.UpdateProgrammer1((int)enumUpgradeFirmware.FINALIZE, "",
       out blockNo, out strError);
        goto ShowResult;
    }
    sLog += blockNo.ToString("X2") + " ";
//Here Progress Bar of Verification can be refreshed for "i-th" block
    Application.DoEvents();
}
r = pp.UpdateProgrammer1((int)enumUpgradeFirmware.FINALIZE, "",
                         out blockNo, out strError);
    sLog += "\r\nFirmware Update completed. ";
        ShowResult:
    MessageBox.Show(sLog);
```

USB2IIC_AsyncMode(IN mode, string OUT strError)

Sets the Sync (0) or Async (1) mode of USB2IIC protocol from the client application standpoint. This function is applicable for USB2IIC Bridge/TrueTouch Bridge and compatible devices (FirstTouch). In Sync mode, works with I2C_xxx functions and all service calls (such as SetPowerVoltage, PowerOn, and GetProgrammerVersion). In Async mode, the client application manually sends data down to the bridge using USB2IIC_SendData() function and receives responses in USB2IIC_ReceivedData() event. This mode is useful in high-speed applications with required speeds up to 64000 bytes/sec of input traffic (from IN-endpoint).

C#: `int USB2IIC_AsyncMode(int fMode, out string strError);`

```
string strError;
int mode = 1; //1 - Asynchrounous, 0 - Synchronous
int hr = pp.USB2IIC_AsyncMode(mode, out strError);
```

USB2IIC_AsyncMode1(IN mode, nvector extra, string OUT strError)

Same as the previous function except that it also allows you to set bulk async mode.

This method is only useful for MiniProg3 in SWV mode. You must configure the MiniProg3 for SWV data sampling before using of method.

mode – defines Async/Sync mode for the current Bridge.

extra – vector of the byte that defines parameters for the *mode* parameter.

The following table gives a detailed description of the *mode* parameter.

Table 2-6. Description of fMode Parameter

Mode	Params	Description
0	-	Stop Async mode
1	-	TrueTouchBridge: Start INT (HID) Async mode, maximum packet size is 62 bytes
2	extra[0,3] - size of bulk packet to be received. Order:LSb X X MSb.	TrueTouchBridge: Starts bulk async mode, where packet size is related for USB packet (not to application interface I2C/SPI, and so on)
	-	MiniProg3 - starts receiving SWV data asynchronously and sends it to client via event USB2IIC_DataTransferred() from COM interface
X	Y	Reserved for future use

C#:
```
int USB2IIC_AsyncMode1(int fMode, object out extra, string strError);
string strError = "";
int hr = 0;
int bulk_size = 512; // Size of read bulk transaction (2-bytes).
byte[] buffer = new byte[2];
buffer[0] = (byte)(bulk_size & 0xFF); //LSb two bytes - size of read bulk
transaction
buffer[1] = (byte)((bulk_size >> 8) & 0xFF); //MSb
hr = pp.USB2IIC_AsyncMode1(2, buffer, out strError); //Start bulk async
mode in COM
```

USB2IIC_DataTransfer(nvector IN dataIN, nvector OUT dataOUT, string OUT strError)

Sends *dataIN* packet down to tthe Bridge and returns response in *dataOUT*. This function is intended for use with USB2IIC Bridge and compatible devices (FirstTouch) supporting its protocol. The Sync mode of communication is required to use this function.

C#:
```
int USB2IIC_DataTransfer(object dataIN, out object dataOUT,
   out string strError);
   string strError;
   byte[] dataIN = new byte[4];
   object dataOUT;
   dataIN[0] = 0x02; //Start condition
   dataIN[1] = 0x00; //not used
   dataIN[2] = 0x80; //Extended address - service functions of Bridge
   dataIN[3] = 0x01; //+5.0 V
 int hr = pp.USB2IIC_DataTransfer(dataIN, out dataOUT, out strError);
```

USB2IIC_ReceivedData(nvector IN dataIN)

This event is fired when a data input packet is received (from IN-endpoint) from the bridge. It is only triggered in the Async mode of communication. If the packet received has errors, the size of the *dataIN* array is 0.

```
C#: event
_IPSoCProgrammerCOM_ObjectEvents_USB2IIC_ReceivedDataEventHandler
              USB2IIC_ReceivedData;

        public void InitializeEvent()
        {

            pp.USB2IIC_ReceivedData += new
_IPSoCProgrammerCOM_ObjectEvents_USB2IIC_ReceivedDataEventHandler(USB2IIC
_ReceivedData);
        }

        void USB2IIC_ReceivedData(object dataIN)
        {
            byte[] data = dataIN as byte[];
            if (data.Length == 0) {
                MessageBox.Show("Error happened while reading packet!");
                return;
            }
            //Process packet

        }
```

USB2IIC_SendData(nvector IN dataIN, string OUT strError)

Sends *dataIN* packet down to the Bridge and returns immediately. The respond packet must be received in the USB2IIC_ReceivedData() event. This function is recommended for usage in Async mode only.

```
C#: int USB2IIC_SendData(object dataIN, out string strError);
 string strError;
 byte[] dataIN = new byte[4];
 dataIN[0] = 0x02; //Start condition
 dataIN[1] = 0x00; //not used
 dataIN[2] = 0x80; //Extended address - service functions of Bridge
 dataIN[3] = 0x01; //+5.0 V
 int hr = pp.USB2IIC_SendData(dataIN, out strError);
```

VerifyBlock(IN blockID, nvector IN block, OUT verResult, string OUT strError)

Verifies a block of flash given by the *blockID* parameter, according to data from *block* vector. This function operations on the current bank; the blockID parameter is in the range [0..BlocksPerBank–1]. The *block* vector must have size equal to BytesPerFlashBlock of the current PSoC device. This function returns i*verResult* =1 if verification succeeds, or 0 if verification fails.

```
C#:  int VerifyBlock(int blockID, object block, out int verResult, out
string strError);

string strError;
byte[] data = new byte[64];
```

```
for (int i = 0; i < 64; i++) data[i] = 0x30;
data[0] = 0x05; data[63] = 0x07;
int verResult;
int hr = pp.VerifyBlock(127,data,out verResult, out strError);
if (SUCCEEDED(hr))
{
    if (verResult != 0)
        MessageBox.Show("Flash Block Verified Successfully!");
    else
        MessageBox.Show(strError);
}
else
   MessageBox.Show(strError);
```

VerifyBlock1(IN bank, IN blockID, nvector IN block, OUT verResult, string OUT strError)

Verifies a block of flash defined by the *blockID* and *bank* parameters, according to data from *block* vector. This vector must have a size equal to BytesPerFlashBlock of the current PSoC device. This function returns *verResult* = 1 if verification succeeds, or 0 if verification fails.

C#:

```
int VerifyBlock1(int bank, int blockID, object block,
    out int verResult, out string strError);
string strError;
byte[] data = new byte[64];
for (int i = 0; i < 64; i++) data[i] = 0x30;
data[0] = 0x05; data[63] = 0x07;
int verResult;
int hr = pp.VerifyBlock1(3,127, data, out verResult, out strError);
if (SUCCEEDED(hr))
{
    if (verResult != 0)
        MessageBox.Show("Flash Block Verified Successfully!");
    else
        MessageBox.Show(strError);
}
else
 MessageBox.Show(strError);
```

VerifyBlockFromHex(IN hexBlockID, OUT verResult, string OUT strError)

Verifies a block from the hex file given by its linear index *hexBlockID*, to the corresponding flash block of the acquired PSoC device. This function returns *verResult* = 1 if verification succeeds, or 0 if verification fails. Make sure the hex file is loaded before you use this function.

C#:

```
int VerifyBlockFromHex(int hexBlockID, out int verResult, out
string strError);

string strError;
int verResult;
int hr = pp.VerifyBlockFromHex(511,out verResult, out strError);
if (SUCCEEDED(hr))
```

```
{
    if (verResult != 0)
        MessageBox.Show("Flash Block Verified Successfully!");
    else
        MessageBox.Show(strError);
}
else
        MessageBox.Show(strError);
```

VerifyProtect(string OUT strError)

Verifies that the protection of the blocks equals the protection level specified for that block in the hex file. It does not verify that the contents of the blocks match with the hex file.

C#:
```
int VerifyProtect(out string strError);
string strError;
int hr = pp.VerifyProtect(out strError);
if (SUCCEEDED(hr))
MessageBox.Show("Protection area Verified Successfully!");
else
MessageBox.Show(strError);
```

Version(string OUT version)

Returns the version of this COM object.

C#:
```
string Version(out string strError);

string version = pp.Version();
MessageBox.Show("COM-object Version: "+version);
```

WriteBlock(IN blockID, nvector IN block, OUT sscResult, string OUT strError)

Writes one block to flash defined by the *blockID* parameter in the current bank. *blockID* is therefore in the range [0..BlocksPerBank–1]. The vector with *block* must have a size equal to the block's size of the acquired PSoC device.

It is recommended that you analyze *sscResult* after the operation succeeds. It must be 0 if the block is saved in flash, and not 0 if the supervisory operation failed (for example, because of the protection level of the flash block). Also, the proper PSoC device must be set before using this function; call GetSiliconId() immediately after the device is acquired.

blockID must be erased before using this function.

C#:
```
int WriteBlock(int blockID, object block, out int sscResult, out
string strError);

string strError;
byte[] block = new byte[64];
for (byte i = 0; i < 64; i++) block[i] = i;
int sscResult;
int hr = pp.WriteBlock(1, block, out sscResult, out strError);
if (SUCCEEDED(hr))
```

```
    MessageBox.Show("WriteBlock succeeded. sscResult = "+sscResult);
else
    MessageBox.Show(strError);
```

WriteBlock1(IN bank, IN blockID, nvector IN block, OUT sscResult, string OUT strError)

Same as the WriteBlock() function, but it only operates on the flash bank defined by *bank* parameter.

C#:
```
int WriteBlock1(int bank, int blockID, object block,
 out int sscResult, out string strError);

string strError;
byte[] block = new byte[64];
for (byte i = 0; i < 64; i++) block[i] = i;
int sscResult;
int hr = pp.WriteBlock1(0,1, block, out sscResult, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("WriteBlock1 succeeded. sscResult = " +
         sscResult);
else
    MessageBox.Show(strError);WriteBlockFromHex(IN hexBlockID, IN flash-
BlockID, OUT sscResult, string OUT strError)
```

Writes a block from the hex file given by the *hexBlockID* parameter to the flash block *flashBlockID* in the current bank. *hexBlockID* is in the range [0..TotalFlashBlocksCount–1] and the *flashBlockID* lies in [0..FlashBlocksPerBank–1]. This function calls HEX_ReadData() and then WriteBlock().

C#:
```
 int WriteBlockFromHex(int hexBlockID, int flashBlockID,
  out int sscResult, out string strError);

string strError;
int sscResult;
int hr = pp.WriteBlockFromHex(1, 1, out sscResult,out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Operation succeeded. sscResult = " +
   sscResult);
else
    MessageBox.Show(strError);
```

WriteBlockFromHex1(IN hexBlockID, IN bank, IN flashBlockID, OUT sscResult, string OUT strError)

Same as the WriteBlockFromHex() function, but it operates only on the flash bank defined by the *bank* parameter. This function calls HEX_ReadData() and then WriteBlock1().

C#:
```
 int WriteBlockFromHex1(int hexBlockID, int bank,
  int flashBlockID, out int sscResult, out string strError);

int sscResult;
string strError;
int hr = pp.WriteBlockFromHex1(1, 0, 1, out sscResult,
```

```
        out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("Operation succeeded. sscResult = " +
   sscResult);
else
    MessageBox.Show(strError);
```

WriteIO(IN address, nvector IN data, string OUT strError)

Writes *data* vector to the I/O registers of the acquired PSoC device in the current register's bank. The data is written in the consecutive registers starting from *address*.

C#: `int WriteIO(int addr, object data, out string strError);`

```
string strError;
byte[] data = new byte[]{0x00, 0x12, 0xA0 };
int hr = pp.WriteIO(0x03,data, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("WriteIO Succeeded!");
else
    MessageBox.Show(strError);
```

WriteRAM(IN address, nvector IN data, string OUT strError)

Writes *data* vector to the SRAM of the acquired PSoC device in the current SRAM bank. The data is written in the consecutive addresses starting from *address*.

C#: `int WriteRAM(int addr, object data, out string strError);`

```
string strError;
byte[] data = new byte[] { 0x00, 0x12, 0xA0 };
int hr = pp.WriteRAM(0x03, data, out strError);
if (SUCCEEDED(hr))
    MessageBox.Show("WriteRAM Succeeded!");
else
    MessageBox.Show(strError);
```

3. Examples

This chapter gives examples of how to use the COM object in different programming languages.

3.1 PSoC 1 ISSP Examples

3.1.1 Locations

.\Examples\Programming\PSoC1\ISSP\C_Sharp\

.\Examples\Programming\PSoC1\ISSP\C_Sharp_Lib\

.\Examples\Programming\PSoC1\ISSP\Perl_Ex\

.\Examples\Programming\PSoC1\ISSP\Cpp_Ex\

.\Examples\Programming\PSoC1\ISSP\Python_Ex\

3.1.2 C_Sharp ISSP Example

```
using System;
using System.Collections.Generic;
using System.Text;
using PP_COM_Wrapper;

namespace C_Sharp
{
    class Program
    {
        static PSoCProgrammerCOM_ObjectClass pp = new PSoCProgrammerCOM_ObjectClass();
        static string m_sLastError;

        static bool SUCCEEDED(int hr)
        {
            return hr >= 0;
        }

        static private int OpenPort()
        {
            int hr;
            //Open Port - get last (connected) port in the ports list
            object portArray;
            hr = pp.GetPorts(out portArray, out m_sLastError);
```

```
            if (!SUCCEEDED(hr)) return hr;

            string[] ports = portArray as string[];
            if (ports.Length <= 0)
            {
                m_sLastError = "Connect any Programmer to PC";
                return -1;
            }

            string portName = ports[ports.GetUpperBound(0)];
            hr = pp.OpenPort(portName, out m_sLastError);

            return hr;
        }

        static private int ProgramOnly(int hexImageSize)
        {
            int Banks, BlocksPerBank, BlockSize;
               pp.GetFlashCharacteristics(out BlockSize, out Banks, out BlocksPerBank, out
m_sLastError);
            int hr = -1;
            int hexBlockID = 0;
            int totalHexBlocks = hexImageSize / BlockSize;
            bool fDone = false;
            for (int bank = 0; bank < Banks; bank++)
            {
                pp.SetBank(bank, out m_sLastError);
                for (int block = 0; block < BlocksPerBank; block++)
                {
                    int sscResult;
                     hr = pp.WriteBlockFromHex(hexBlockID, block, out sscResult, out m_sLas-
tError);
                    if (!SUCCEEDED(hr)) return hr;
                    hexBlockID++;
                    if (hexBlockID >= totalHexBlocks) { fDone = true; break; }
                }
                if (fDone) break;
            }
            return hr;
        }

        static private int VerifyOnly(int hexImageSize)
        {
            int Banks, BlocksPerBank, BlockSize;
pp.GetFlashCharacteristics(out BlockSize, out Banks, out BlocksPerBank, out m_sLastError);
            int hr = -1;
```

```
            int totalHexBlocks = hexImageSize / BlockSize;
            for (int i = 0; i<totalHexBlocks; i++)
            {
                int verResult;
                hr = pp.VerifyBlockFromHex(i, out verResult, out m_sLastError);
                if (!SUCCEEDED(hr)) return hr;
                if (verResult == 0) return hr; //Block Not Verified
            }
            return hr;
        }
        static int ProgramAll()
        {
            if (pp == null) return -1;

            int hr;
            //Open Port - get last (connected) port in the ports list
            hr = OpenPort();
            if (!SUCCEEDED(hr)) return hr;
            //Set Protocol ISSP
            hr = pp.SetProtocol(enumInterfaces.ISSP, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Set Power Voltage
            hr = pp.SetPowerVoltage("5.0", out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            // Set Hex File
            int hexImageSize;
            hr = pp.HEX_ReadFile("c:\\cy29466.hex", out hexImageSize, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Set Acquire Mode
            pp.SetAcquireMode("Power", out m_sLastError);
            //Acquire Device
            hr = pp.Acquire(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Calibrate PSoC1 device right after acquire
            pp.Calibrate(1, out m_sLastError);
           //Check HEX file
            int Banks, BlocksPerBank, BlockSize;
pp.GetFlashCharacteristics(out BlockSize, out Banks, out BlocksPerBank, out m_sLastError);
            int flashImageSize = Banks * BlocksPerBank * BlockSize;
            if (hexImageSize > flashImageSize) {
                m_sLastError = "Hex File Size is bigger than Device's Flash Size";
                return -1;
            }
            //Erase All
            hr = pp.EraseAll(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
```

```
    //Program
    hr = ProgramOnly(hexImageSize);
    if (!SUCCEEDED(hr)) return hr;
    //Verify
    hr = VerifyOnly(hexImageSize);
    if (!SUCCEEDED(hr)) return hr;
    //Protect
    hr = pp.ProtectAll(out m_sLastError);
    if (!SUCCEEDED(hr)) return hr;
    //Verify Protect
    hr = pp.VerifyProtect(out m_sLastError);
    if (!SUCCEEDED(hr)) return hr;
    //CheckSum
    ushort flashCS=1, hexCS=2;
    pp.checksum(0, out flashCS, out m_sLastError);
    pp.HEX_ReadChecksum(out hexCS, out m_sLastError);
    if (flashCS != hexCS)
    {
        m_sLastError = "Mismatch of Checksum";
        return -1;
    }
    //Release PSoC Device
    pp.ReleaseChip(out m_sLastError);

    return hr;
}

static private int UpgradeBlock()
{
    if (pp == null) return -1;

    int hr;
    //Open Port - get last (connected) port in the ports list
    hr = OpenPort();
    if (!SUCCEEDED(hr)) return hr;
    //Set Protocol ISSP
    hr = pp.SetProtocol(enumInterfaces.ISSP, out m_sLastError);
    if (!SUCCEEDED(hr)) return hr;
    //Set Power Voltage
    hr = pp.SetPowerVoltage("5.0", out m_sLastError);
    if (!SUCCEEDED(hr)) return hr;
    //Set Acquire Mode
    pp.SetAcquireMode("Power", out m_sLastError);
    //Acquire Device
    hr = pp.Acquire(out m_sLastError);
    if (!SUCCEEDED(hr)) return hr;
```

```
            //Calibrate PSoC1 device right after acquire
            pp.Calibrate(1, out m_sLastError);
            //Erase Block 2, in bank 1
            int bank = 1, block = 2;
            int sscResult;
            hr = pp.EraseBlock1(block, bank, out sscResult, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            if (sscResult != 0) {
                m_sLastError = "Block is Write Protected!";
                return -1;
            }
            //Write Block
            byte[] data = new byte[64];
            for (byte i = 0; i < 64; i++) data[i] = i;
            hr = pp.WriteBlock1(bank, block, data, out sscResult, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Verify Block
            int verResult = 0;
            hr = pp.VerifyBlock1(bank, block, data, out verResult, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            if (verResult == 0) return -1; //Block is not verified
            //Release PSoC Device
            pp.ReleaseChip(out m_sLastError);
            return hr;
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Program All using COM-object interface only");
            int hr;
            string str;
            hr = ProgramAll();
            //hr = UpgradeBlock();
            if (SUCCEEDED(hr)) str = "Succeeded!";
            else str = "Failed! " + m_sLastError;
            Console.WriteLine(str);

            Console.ReadKey();
        }
    }
}
```

3.2 PSoC 3 / PSoC 5 (SWD or JTAG) Examples

3.2.1 Locations

```
.\Examples\Programming\PSoC3_5\SWD\C_Sharp\
.\Examples\Programming\PSoC3_5\SWD\C_Sharp_EEPROM\
.\Examples\Programming\PSoC3_5\SWD\C_Sharp_Lib\
.\Examples\Programming\PSoC3_5\SWD\Perl_Ex\
.\Examples\Programming\PSoC3_5\SWD\Cpp_Ex\
.\Examples\Programming\PSoC3_5\SWD\Python_Ex\

.\Examples\Programming\PSoC3_5\JTAG\C_Sharp\
.\Examples\Programming\PSoC3_5\JTAG\C_Sharp_Lib\
.\Examples\Programming\PSoC3_5\JTAG\Perl_Ex\
.\Examples\Programming\PSoC3_5\JTAG\Cpp_Ex\
.\Examples\Programming\PSoC3_5\JTAG\Python_Ex\
```

3.2.2 C_Sharp SWD Example

```
using System;
using System.Collections.Generic;
using System.Text;
using PP_COM_Wrapper;

namespace C_Sharp
{
    class Program
    {
        static PSoCProgrammerCOM_ObjectClass pp = new PSoCProgrammerCOM_ObjectClass();
        static string m_sLastError;

        //Distinguishing identifier of PSoC3/5 families
        const int LEOPARD_ID = 0xE0;
        const int PANTHER_ID = 0xE1;

        static bool SUCCEEDED(int hr)
        {
            return hr >= 0;
        }

        //Check JTAG ID of device - identify family PSoC3 or PSoC5
        static int GetGenerationByJtagID(byte[] JtagID)
        {
            int distinguisher = (((JtagID[0] & 0x0F) << 4) | (JtagID[1] >> 4));
            return distinguisher;
        }

        static bool IsPSoC3ES3(byte[] jtagID)
        {
            if (GetGenerationByJtagID(jtagID) == LEOPARD_ID)
            {
                if ((jtagID[0] >> 4) >= 0x01) return true; //silicon version==0x01 saved in
bits [4..7]
            }                                                   //For ES0-2==0x00, ES3==0x01
            return false;
        }

        static private int OpenPort()
        {
            int hr;
```

```
            //Open Port - get last (connected) port in the ports list
            object portArray;
            hr = pp.GetPorts(out portArray, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;

            string[] ports = portArray as string[];
            if (ports.Length <= 0)
            {
                m_sLastError = "Connect any Programmer to PC";
                return -1;
            }

            bool bFound = false;
            string portName = "";
            for (int i = 0; i < ports.Length; i++) {
                if (ports[i].StartsWith("MiniProg3") || ports[i].StartsWith("DVKProg") ||
                ports[i].StartsWith("FirstTouch") || ports[i].StartsWith("Gen-FX2LP")) {
                    portName = ports[i];
                    bFound = true;
                    break;
                }
            }

            if (!bFound) {
                m_sLastError = "Connect any MiniProg3/DVKProg/FirstTouch/Gen-FX2LP device to
                the PC";
                return -1;
            }

           //Port should be opened just once to connect Programmer device (MiniProg1/3,etc).
            //After that you can use Chip-/Programmer- specific APIs as long as you need.
            //No need to reopen port when you need to acquire chip 2nd time, just call
Acquire() again.
            //This is true for all other APIs which get available once port is opened.
           //You have to call OpenPort() again if port was closed by ClosePort() method, or
            //when there is a need to connect to other programmer, or
            //if programmer was physically reconnected to USB-port.

            hr = pp.OpenPort(portName, out m_sLastError);

            return hr;
        }

        static int CheckHexAndDeviceCompatibility(out bool result)
        {
            int hr;
            object jtagID;
            byte[] hexJtagID, chipJtagID;
            result = false;

            hr = pp.DAP_GetJtagID(out jtagID, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            chipJtagID = jtagID as byte[];

            hr = pp.HEX_ReadJtagID(out jtagID, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            hexJtagID = jtagID as byte[];

            result = true;
            for (byte i = 0; i < 4; i++)
            {
                if (hexJtagID[i] != chipJtagID[i])
                {
                    result = false;
```

```
                    break;
                }
            }
            return 0;
        }

        static int CheckSum_All(out uint cs)
        {
            int hr = 0;
            cs = 0;

            object arrays;
           hr = pp.PSoC3_GetSonosArrays(enumSonosArrays.ARRAY_FLASH, out arrays, out m_sLas-
tError);
            if (!SUCCEEDED(hr)) return hr;

            //Checksum all the Flash arrays
            Array arrayInfo = arrays as Array;
            for (int i = arrayInfo.GetLowerBound(1); i <= arrayInfo.GetUpperBound(1); i++)
            {
                int arrayID = (int)arrayInfo.GetValue(0, i);
                int arraySize = (int)arrayInfo.GetValue(1, i);
                //Get info about flash array
                int rowSize, rowsPerArray, eccPresence;
                hr = pp.PSoC3_GetFlashArrayInfo(arrayID, out rowSize, out rowsPerArray, out
eccPresence, out m_sLastError);
                if (!SUCCEEDED(hr)) return hr;
                //Calculate checksum of the array
                uint arrayChecksum = 0;
                hr = pp.PSoC3_CheckSum(arrayID, 0, rowsPerArray, out arrayChecksum, out
m_sLastError);
                if (!SUCCEEDED(hr)) return hr;
                //Sum checksum
                cs += arrayChecksum;
            }

            return hr;
        }

        static int ProgramNvlArrays(enumSonosArrays nvlArrayType)
        {
            int hr;
            object arrays;
            hr = pp.PSoC3_GetSonosArrays(nvlArrayType, out arrays, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            Array arrayInfo = arrays as Array;
            int addrHex = 0;
            for (int i = arrayInfo.GetLowerBound(1); i <= arrayInfo.GetUpperBound(1); i++)
            {
                int arrayID = (int)arrayInfo.GetValue(0, i);
                int arraySize = (int)arrayInfo.GetValue(1, i);
                //Read data from Hex file
                object data;
                if (nvlArrayType == enumSonosArrays.ARRAY_NVL_USER)
                  hr = pp.HEX_ReadNvlCustom(addrHex, arraySize, out data, out m_sLastError);
                else //enumSonosArrays.ARRAY_NVL_WO_LATCHES
                    hr = pp.HEX_ReadNvlWo(addrHex, arraySize, out data, out m_sLastError);
                if (!SUCCEEDED(hr)) return hr;
                byte[] hexData = data as byte[];
                addrHex += arraySize;
                //Read data from device
                hr = pp.PSoC3_ReadNvlArray(arrayID, out data, out m_sLastError);
                if (!SUCCEEDED(hr)) return hr;
                byte[] chipData = data as byte[];
```

```
                //Compare data from Chip against corresponding Hex-block
                if (chipData.Length != hexData.Length)
                {
                   m_sLastError = "Hex file's NVL array differs from corresponding device's
one!";
                    return -1;
                }
                bool fIdentical = true;
                for (int j = 0; j < arraySize; j++)
                {
                    if (chipData[j] != hexData[j])
                    {
                        fIdentical = false;
                        break;
                    }
                }
               if (fIdentical) continue; //Arrays are equal, skip programming, goto following
array
                //Program NVL array
                hr = pp.PSoC3_WriteNvlArray(arrayID, hexData, out m_sLastError);
                if (!SUCCEEDED(hr)) return hr;
                //PSoC3 ES3 support - check whether ECCEnable bit is changed and reacquire
device
                if (nvlArrayType == enumSonosArrays.ARRAY_NVL_USER)
                {
                    object JtagID;
                    hr = pp.DAP_GetJtagID(out JtagID, out m_sLastError);
                    if (!SUCCEEDED(hr)) return hr;
                    //ES3 and probably newer revisions
                    if (IsPSoC3ES3(JtagID as byte[]))
                    {
                        bool eccEnableChanged = false;
                        if (hexData.Length >= 4)
                            eccEnableChanged = ((hexData[3] ^ chipData[3]) & 0x08) == 0x08;
                       //need to reacquire chip if EccEnable bit was changed to apply it for
flash rows.
                        if (eccEnableChanged)
                            hr = pp.DAP_Acquire(out m_sLastError);
                    }
                    return hr;
                }
            }
            return hr;
        }

        static int GetEccOption(int arrayID)
        {
            int hr;
            int rowSize, rowsPerArray, eccPresence;
            hr = pp.PSoC3_GetFlashArrayInfo(arrayID, out rowSize, out rowsPerArray, out
eccPresence, out m_sLastError);
            int eccHwStatus;
            hr = pp.PSoC3_GetEccStatus(out eccHwStatus, out m_sLastError); //get ecc status
of the acquired device
            if (!SUCCEEDED(hr)) return hr;
            int eccOption = ((eccPresence != 0) && (eccHwStatus == 0)) ? 1 : 0; //take into
account data from the Config area
            return eccOption;
        }

        static int ProgramFlashArrays(int flashSize)
        {
            int hr;
            object arrays;
```

```
            hr = pp.PSoC3_GetSonosArrays(enumSonosArrays.ARRAY_FLASH, out arrays, out m_sLas-
tError);
            if (!SUCCEEDED(hr)) return hr;

            int flashProgrammed = 0;
            //Program Flash arrays
            Array arrayInfo = arrays as Array;
            for (int i = arrayInfo.GetLowerBound(1); i <= arrayInfo.GetUpperBound(1); i++)
            {
                int arrayID = (int)arrayInfo.GetValue(0, i);
                int arraySize = (int)arrayInfo.GetValue(1, i);
                //Program flash array from hex file taking into account ECC (Config Data)
                int rowSize, rowsPerArray, eccPresence;
                hr = pp.PSoC3_GetFlashArrayInfo(arrayID, out rowSize, out rowsPerArray, out
eccPresence, out m_sLastError);
                if (!SUCCEEDED(hr)) return hr;
               int eccOption = GetEccOption(arrayID); //take into account data from the Con-
fig area
                for (int rowID = 0; rowID < rowsPerArray; rowID ++)
                {
                   hr = pp.PSoC3_ProgramRowFromHex(arrayID, rowID, eccOption, out m_sLastEr-
ror);
                    if (!SUCCEEDED(hr)) return hr;
                    //Limit programming to the device flash size
                    flashProgrammed += rowSize;
                    if (flashProgrammed >= flashSize) return hr;
                }
            }
            return hr;
        }

        static int ProgramAll()
        {
            if (pp == null) return -1;

            int hr;
            //Port Initialization
            //Open Port - get last (connected) port in the ports list
            hr = OpenPort();
            if (!SUCCEEDED(hr)) return hr;
            //Setup Power - "2.5V" and internal
            hr = pp.SetPowerVoltage("2.5", out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            hr = pp.PowerOn(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Set protocol, connector and frequency
            hr = pp.SetProtocol(enumInterfaces.SWD, out m_sLastError); //SWD-protocol
            if (!SUCCEEDED(hr)) return hr;
            hr = pp.SetProtocolConnector(1, out m_sLastError); //10-pin connector
            if (!SUCCEEDED(hr)) return hr;
           hr = pp.SetProtocolClock(enumFrequencies.FREQ_03_0, out m_sLastError); //3.0 MHz
clock on SWD bus
            // Set Hex File
            int hexImageSize;
            hr = pp.HEX_ReadFile("c:\\Design12.hex", out hexImageSize, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Set Acquire Mode
            pp.SetAcquireMode("Reset", out m_sLastError);

            //The "Programming Flow" proper
            //Acquire Device
            hr = pp.DAP_Acquire(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Check Hex File and Device compatibility
```

```
            bool fCompatibility;
            hr = CheckHexAndDeviceCompatibility(out fCompatibility);
            if (!SUCCEEDED(hr)) return hr;
            if (!fCompatibility)
            {
                m_sLastError = "The Hex file does not match the acquired device, please con-
nect the appropriate device";
                return -1;
            }
            //Erase All
            hr = pp.PSoC3_EraseAll(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Program User NVL arrays
            hr = ProgramNvlArrays(enumSonosArrays.ARRAY_NVL_USER);
            if (!SUCCEEDED(hr)) return hr;
            //Program Flash arrays
            hr = ProgramFlashArrays(hexImageSize);
            //Protect All arrays
            hr = pp.PSoC3_ProtectAll(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //CheckSum
            ushort hexChecksum;
            uint flashChecksum;
            hr = CheckSum_All(out flashChecksum);
            if (!SUCCEEDED(hr)) return hr;
            hr = pp.HEX_ReadChecksum(out hexChecksum, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            flashChecksum &= 0xFFFF;
            if (flashChecksum != hexChecksum)
            {
                m_sLastError = "Mismatch of Checksum: Expected 0x" + flashCheck-
sum.ToString("X") + ", Got 0x" + hexChecksum.ToString("X");
                return -1;
            }
            else
                Console.WriteLine("Checksum 0x" + flashChecksum.ToString("X"));

            //Release PSoC3 device
            hr = pp.DAP_ReleaseChip(out m_sLastError);

            return hr;
        }

        static private int UpgradeBlock()
        {
            if (pp == null) return -1;

            int hr;
            //Port Initialization
            //Open Port - get last (connected) port in the ports list
            hr = OpenPort();
            if (!SUCCEEDED(hr)) return hr;
            //Setup Power - "2.5V" and internal
            hr = pp.SetPowerVoltage("2.5", out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            hr = pp.PowerOn(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Set protocol, connector and frequency
            hr = pp.SetProtocol(enumInterfaces.SWD, out m_sLastError); //SWD-protocol
            if (!SUCCEEDED(hr)) return hr;
            hr = pp.SetProtocolConnector(1, out m_sLastError); //10-pin connector
            if (!SUCCEEDED(hr)) return hr;
            hr = pp.SetProtocolClock(enumFrequencies.FREQ_03_0, out m_sLastError); //<=3.0
MHz for Read operations
```

```
    //Set Acquire Mode
    pp.SetAcquireMode("Reset", out m_sLastError);

    //Acquire Device
    hr = pp.DAP_Acquire(out m_sLastError)
    if (!SUCCEEDED(hr)) return hr;
    //Write Row, use PSoC3_WriteRow() instead PSoC3_ProgramRow()
    byte[] writeData = new byte[288]; //User and Config area of the Row (256+32)
    for (int i = 0; i < 288; i++) writeData[i] = (byte)i;
    int arrayID = 0, rowID = 255;
    hr = pp.PSoC3_WriteRow(arrayID, rowID, writeData, out m_sLastError);
    if (!SUCCEEDED(hr)) return hr;
    //Verify Row - only user area (without Config one)
    object readData;
    hr = pp.PSoC3_ReadRow(arrayID, rowID, 0, out readData, out m_sLastError);
    if (!SUCCEEDED(hr)) return hr;
    byte[] readRow = readData as byte[];
    for (int i = 0; i < readRow.Length; i++) //check 256 bytes
    {
        if (readRow[i] != writeData[i])
        {
            hr = -1;
            break;
        }
    }
    if (!SUCCEEDED(hr))
    {
        m_sLastError = "Verification of User area failed!";
        return hr;
    }
    //Verify Row - Config are only
    hr = pp.PSoC3_ReadRow(arrayID, rowID, 1, out readData, out m_sLastError);
    if (!SUCCEEDED(hr)) return hr;
    readRow = readData as byte[];
    for (int i = 0; i < readRow.Length; i++) //check 32 bytes
    {
        if (readRow[i] != writeData[256 + i])
        {
            hr = -1;
            break;
        }
    }
    if (!SUCCEEDED(hr))
    {
        m_sLastError = "Verification of Config area failed!";
        return hr;
    }

    //Release PSoC3 chip
    hr = pp.DAP_ReleaseChip(out m_sLastError);

    return hr;
}

static void Main(string[] args)
{
    Console.WriteLine("Program All using COM-object interface only");
    int hr;
    string str;

    hr = ProgramAll();
    //hr = UpgradeBlock();
    if (SUCCEEDED(hr)) str = "Succeeded!";
```

```
            else str = "Failed! " + m_sLastError;
            Console.WriteLine(str);

            Console.ReadKey();
        }
    }
}
```

3.2.3 C_Sharp_EEPROM SWD Example

```
using System;
using System.Collections.Generic;
using System.Text;
using PP_COM_Wrapper;

namespace C_Sharp_EEPROM
{
    class Program
    {
        static PSoCProgrammerCOM_ObjectClass pp = new PSoCProgrammerCOM_ObjectClass();
        static string m_sLastError;

        //Distinguishing identifier of PSoC3/5 families
        const int LEOPARD_ID = 0xE0;
        const int PANTHER_ID = 0xE1;

        enum enumChipFamily { LEOPARD, PANTHER }

        static enumChipFamily CHIP_FAMILY;

        static bool SUCCEEDED(int hr)
        {
            return hr >= 0;
        }

        //Check JTAG ID of device - identify family PSoC3 or PSoC5
        static int GetGenerationByJtagID(byte[] JtagID)
        {
            int distinguisher = (((JtagID[0] & 0x0F) << 4) | (JtagID[1] >> 4));
            return distinguisher;
        }

        static int GetChipFamily()
        {
            int hr = 0;
            object JtagID;
            hr = pp.DAP_GetJtagID(out JtagID, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;

            int chipFamily = GetGenerationByJtagID(JtagID as byte[]);

            if (chipFamily == PANTHER_ID)
                CHIP_FAMILY = enumChipFamily.PANTHER;
            else
            if (chipFamily == LEOPARD_ID)
                CHIP_FAMILY = enumChipFamily.LEOPARD;
            return hr;
        }

        static private int OpenPort()
        {
            int hr;
            //Open Port - get last (connected) port in the ports list
            object portArray;
```

```
            hr = pp.GetPorts(out portArray, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;

            string[] ports = portArray as string[];
            if (ports.Length <= 0)
            {
                m_sLastError = "Connect any Programmer to PC";
                return -1;
            }

            bool bFound = false;
            string portName = "";
            for (int i = 0; i < ports.Length; i++) {
                if (ports[i].StartsWith("MiniProg3") || ports[i].StartsWith("DVKProg") ||
                ports[i].StartsWith("FirstTouch") || ports[i].StartsWith("Gen-FX2LP")) {
                    portName = ports[i];
                    bFound = true;
                    break;
                }
            }

            if (!bFound) {
               m_sLastError = "Connect any MiniProg3/DVKProg/FirstTouch/Gen-FX2LP device to
               the PC";
                return -1;
            }

           //Port should be opened just once to connect Programmer device (MiniProg1/3,etc).
            //After that you can use Chip-/Programmer- specific APIs as long as you need.
            //No need to repoen port when you need to acquire chip 2nd time, just call
Acquire() again.
            //This is true for all other APIs which get available once port is opened.
           //You have to call OpenPort() again if port was closed by ClosePort() method, or
            //when there is a need to connect to other programmer, or
            //if programmer was physically reconnected to USB-port.

            hr = pp.OpenPort(portName, out m_sLastError);

            return hr;
        }

        static int DAP_WriteIO(int address, int data)
        {
            int hr = 0;
            if (CHIP_FAMILY == enumChipFamily.PANTHER)
            {
                data <<= ((address & 3)*8);
                address |= 0x40000000;
            }
            hr = pp.DAP_WriteIO(address, data, out m_sLastError)
            return hr;
        }

        static int DAP_ReadIO(int address, out int data)
        {
            int hr = 0;
            if (CHIP_FAMILY == enumChipFamily.PANTHER)
            {
                address |= 0x40000000;
            }
            hr = pp.DAP_ReadIO(address, out data, out m_sLastError);
            if (CHIP_FAMILY == enumChipFamily.PANTHER)
            {
                data = (data >> ((address & 3)*8)) & 0xFF;
```

```
    }
    return hr;
}

static int InitEeprom()
{
    int hr = 0;
    int addr = 0x43AC;
    int data = 0x11;
    hr = DAP_WriteIO(addr, data);
    return hr;

}

static int EraseAllEEPROM(int arrayID, int rowsPerArray)
{
    int hr = 0;
    int sectorsCount = rowsPerArray/64;
    for (int i = 0; i < sectorsCount; i++)
    {
        hr = pp.PSoC3_EraseSector(arrayID, i, out m_sLastError);
        if (!SUCCEEDED(hr)) return hr;
    }
    return hr;
}

static int GetEepromRowData(int rowID, int rowSize, out byte[] data)
{
    data = new byte[rowSize];
    for (int i = 0; i < data.Length; i++)
        data[i] = (byte)(i + rowID + 1);
    return 0;
}

static int ProgramEepromArray(int arrayID, int rowSize, int rowsPerArray)
{
    int hr = 0;
    string str = " RowIDs: ";

    Console.Write(str);
    for (int rowID = 0; rowID < rowsPerArray; rowID++)
    {
        str = " " + rowID.ToString();
        Console.Write(str); // Show current RowID

        byte[] data;
        GetEepromRowData(rowID, rowSize, out data);
       hr = pp.PSoC3_WriteRow(arrayID, rowID, data, out m_sLastError);
        if (!SUCCEEDED(hr)) break;
    }
    return hr;
}

static int VerifyEepromArray(int arrayID, int rowSize, int rowsPerArray)
{
    int hr = 0;
    string str = " RowIDs: ";

    Console.Write(str);
    for (int rowID = 0; rowID < rowsPerArray; rowID++)
    {
        str = " " + rowID.ToString();
        Console.Write(str); // Show current RowID
```

```
                byte[] dataExpected;
                GetEepromRowData(rowID, rowSize, out dataExpected);

                int offset = rowID * rowSize;
                int[] dataRead = new int[rowSize];

                for(int i = 0; i < rowSize; i++)
                {
                    hr = DAP_ReadIO(0x8000 + offset + i, out dataRead[i]);
                    if (!SUCCEEDED(hr)) return hr;
                    if (dataExpected[i] != dataRead[i]) return -1;
                }
            }
            return hr;
        }

        static int ProgramEEPROMAll()
        {
            if (pp == null) return -1;

            int hr;
            string strSucceeded = " Succeeded!";

            //Port Initialization
            //Open Port - get last (connected) port in the ports list
            hr = OpenPort();
            if (!SUCCEEDED(hr)) return hr;
            //Setup Power - "2.5V" and internal
            hr = pp.SetPowerVoltage("2.5", out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            hr = pp.PowerOn(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            //Set protocol, connector and frequency
            hr = pp.SetProtocol(enumInterfaces.SWD, out m_sLastError); //SWD-protocol
            if (!SUCCEEDED(hr)) return hr;
            hr = pp.SetProtocolConnector(1, out m_sLastError); //10-pin connector
            if (!SUCCEEDED(hr)) return hr;
           hr = pp.SetProtocolClock(enumFrequencies.FREQ_03_0, out m_sLastError); //3.0 MHz
clock on SWD bus
            //Set Acquire Mode
            pp.SetAcquireMode("Reset", out m_sLastError);

            //The "Programming Flow" proper
            //Acquire Device
            Console.WriteLine("\r\n>Acquire Device ...");
            hr = pp.DAP_Acquire(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            Console.WriteLine(strSucceeded);

            //Get Chip Family
            hr = GetChipFamily();
            if (!SUCCEEDED(hr)) return hr;

            //EEPROM Array ID
            int arrayID = 0x40, rowSize, rowsPerArray;

            // Get EEPROM Array Info
            hr = pp.PSoC3_GetEepromArrayInfo(arrayID, out rowSize, out rowsPerArray, out
m_sLastError);
            if (!SUCCEEDED(hr)) return hr;

            // Initialize EEPROM
            hr = InitEeprom();
            if (!SUCCEEDED(hr)) return hr;
```

```
            //Erase all EEPROM - works for PSoC3 ES3, and doesn't for PSoC3 ES2 and PSoC5 ES1
            //                   This API is useful when whole EEPROM must be erased.
            //Console.WriteLine(">Eraising EEPROM ...");
           //hr = EraseAllEEPROM(arrayID, rowsPerArray); // Not nessesary, this example shows
how to erase whole EEPROM
            //if (!SUCCEEDED(hr)) return hr;
            //Console.WriteLine(strSucceeded);

            // Program EEPROM array
            Console.WriteLine("\r\n>Programming EEPROM ...");
            hr = ProgramEepromArray(arrayID, rowSize, rowsPerArray);
            Console.WriteLine();
            if (!SUCCEEDED(hr)) return hr;
            Console.WriteLine(strSucceeded);

            // Verify EEPROM array
            Console.WriteLine("\r\n>Verify EEPROM ...");
            hr = VerifyEepromArray(arrayID, rowSize, rowsPerArray);
            Console.WriteLine();
            if (!SUCCEEDED(hr)) return hr;
            Console.WriteLine(strSucceeded);

            //Release PSoC3 device
            hr = pp.DAP_ReleaseChip(out m_sLastError);

            return hr;
        }

        static void Main(string[] args)
        {
            int hr;
            string str;
            Console.WriteLine("*** EEPROM program Example PSoC3/5 ***");
            hr = ProgramEEPROMAll();
            if (!SUCCEEDED(hr))
            {
                str = " Failed! " + m_sLastError;
                Console.WriteLine(str);
            }
            Console.WriteLine("\r\n>Press Any Key ...");
            Console.ReadKey();
        }
    }
}
```

3.3 I²C Examples

3.3.1 Locations

`.\Examples\Protocols\I2C\C_Sharp\`

`.\Examples\Protocols\I2C\Perl_Ex\`

`.\Examples\Protocols\I2C\Cpp_Ex\`

`.\Examples\Protocols\I2C\Python_Ex\`

3.3.2 C_Sharp I²C Example

```
using System;
```

```
using System.Collections.Generic;
using System.Text;
using System.Threading;
using PP_COM_Wrapper;

namespace C_Sharp
{
    class Program
    {
        static PSoCProgrammerCOM_ObjectClass pp = new PSoCProgrammerCOM_ObjectClass();
        static string m_sLastError;

        static bool SUCCEEDED(int hr)
        {
            return hr >= 0;
        }

        static private int OpenPort()
        {
            int hr;
            //Open Port - get last (connected) port in the ports list
            object portArray;
            hr = pp.GetPorts(out portArray, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;

            string[] ports = portArray as string[];
            if (ports.Length <= 0)
            {
                m_sLastError = "Connect any Programmer to PC";
                return -1;
            }

           //Port should be opened just once to connect Programmer device (MiniProg1/3,etc).
           //After that you can use Chip-/Programmer- specific APIs as long as you need.
           //No need to repoen port when you need to acquire chip 2nd time, just call
           //Acquire() again.
           //This is true for all other APIs which get available once port is opened.
           //You have to call OpenPort() again if port was closed by ClosePort() method, or
           //when there is a need to connect to other programmer, or
           //if programmer was physically reconnected to USB-port.

            string portName = ports[ports.GetUpperBound(0)];
            hr = pp.OpenPort(portName, out m_sLastError);

            return hr;
        }

        static private int I2C_Operations()
        {
            int hr;
            string strOut = "";

            hr = OpenPort();
            if (!SUCCEEDED(hr)) return hr;
            //Set Power 5.0V
            pp.SetPowerVoltage("5.0", out m_sLastError);
            //Power On
            pp.PowerOn(out m_sLastError);
            //Set I2C Protocol
            hr = pp.SetProtocol(enumInterfaces.I2C, out m_sLastError);
            //Reset bus
            pp.I2C_ResetBus(out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            Console.WriteLine("Reset bus!");
```

```
            //Get device list
            object deviceList;
            pp.I2C_GetDeviceList(out deviceList, out m_sLastError);
            byte[] data = deviceList as byte[];
            if (!SUCCEEDED(hr))
            {
                m_sLastError = "Failed to enumerate I2C devices";
                return hr;
            }
            //Show devices
            byte[] devices = deviceList as byte[];
            if (devices.Length == 0)
            {
                m_sLastError = "No devices found";
                return hr;
            }
            Console.Write("Devices list:  8bit  7bit");
            for (int i = 0; i < devices.Length; i++)
                strOut += "\r\n     address:  " + Convert.ToString(devices[i] << 1,
16).PadLeft(2, '0').ToUpper() + "    " + Convert.ToString(devices[i], 16).PadLeft(2,
'0').ToUpper();
            Console.WriteLine(strOut);
            //Set I2C speed
            pp.I2C_SetSpeed(enumI2Cspeed.CLK_100K, out m_sLastError);
            Console.WriteLine("Set speed: 100K!");
            //Get I2C speed
            enumI2Cspeed eI2Cspeed;
            pp.I2C_GetSpeed(out eI2Cspeed, out m_sLastError);
            if(eI2Cspeed == enumI2Cspeed.CLK_100K)
                strOut = "100K";
            else
            if(eI2Cspeed == enumI2Cspeed.CLK_50K)
                strOut = "50K";
            else
            if(eI2Cspeed == enumI2Cspeed.CLK_400K)
                strOut = "400K";
            Console.WriteLine("Get speed: " + strOut + "!");
            //w 00 04 01 00 01 ; //ShutDown LED1
            int deviceAddress = 0x00;
            int mode = 0x0A;
            int readLen = 0;
            object dataIN, dataOUT;
            dataIN = new byte[] { 0x04, 0x01, 0x00, 0x01 };
            hr = pp.I2C_DataTransfer(deviceAddress, mode, readLen, dataIN, out dataOUT, out
m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            Console.WriteLine("ShutDown LED1!");
            //w 00 04 01 FF 01 ; //Power On LED1
            data = new byte[] { 0x04, 0x01, 0xFF, 0x01 };
            hr = pp.I2C_SendData(devices[0], data, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            Console.WriteLine("Power On LED1!");
            //Read 4 bytes from device
            hr = pp.I2C_ReadData(devices[0], 4, out dataOUT, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            data = dataOUT as byte[];
            strOut = "";
            strOut = "Read 4 bytes from device: ";
            for(int i = 0; i < data.Length; i++)
                strOut += data[i].ToString("X2") + " ";
            Console.WriteLine(strOut);
            //Read data from register, address 0x02
            object dataIn;
            byte[] ReadAddress = new byte[1] { 0x02 };
```

```
            hr = pp.I2C_ReadDataFromReg(0x00, ReadAddress, 5, out dataIn, out m_sLastError);
            if (!SUCCEEDED(hr)) return hr;
            data = dataIn as byte[];
            strOut = "";
            for (byte i = 0; i < data.Length; i++)
                strOut = strOut + data[i].ToString("X2") + " ";
            Console.WriteLine("Read data from addr 0x02 of device addr 0x00: " + strOut);
            return hr;
        }

        static void Main(string[] args)
        {
            int hr = 0;
            string str = "";
            hr = I2C_Operations();
            if (m_sLastError != "") str += m_sLastError + "\r\n";
            if (SUCCEEDED(hr)) str += "Succeeded!";
            else str = "Failed! " + m_sLastError;
            Console.WriteLine(str);
            Console.ReadKey();
        }
    }
}
```

3.4 SWV Examples

3.4.1 Locations

```
.\Examples\Protocols\SWV\C#\MiniProg3_SWV\
.\Examples\Protocols\SWV\Firmware_SWV\
```

See the *ReadMe.txt* file (in the SWV folder) for details on how to use these examples (such as the hardware, chip, programmer, and connections to use).

3.5 UART Examples

3.5.1 Locations

```
.\Examples\Protocols\UART\C#\MiniProg3_TX8\
.\Examples\Protocols\UART\Firmware_TX8\
```

See the *ReadMe.txt* file (in the UART folder) for details on how to use these examples (such as the hardware, chip, programmer, and connections to use).

Revision History

Document Revision History

Document Title: PSoC Programmer Component Object Model (COM) Interface Guide Document Number: 001-45209			
Revision	**ECN#**	**Issue Date**	**Description of Change**
**	2505946	May 20, 2008	New COM Interface for PSoC Programmer.
*A	2794465	Oct 29, 2009	Added more than 50 APIs- PSoC 3, I2C and other new APIs, command overview table added
*B	2828334	Dec 17, 2009	Correcting Page Numbering
*C	2878957	May 03, 2010	New APIs, updated two APIs
*D	2970155	July 05, 2010	Updated Revision History
*E	3607618	May 03, 2012	Comprehensive updates on all APIs, update to PSoC Programmer 3.15. Updated description of SetProtocolConnector to include SWD mode for TrueTouch Bridge, most sections rewritten to conform to current quality standards
*F	3729151	Aug 30, 2012	Updated Template. Updated Section 1.2 Using the COM Object - Late Binding Updated Section 3.1.2 C_Sharp ISSP Example. Updated API SetProtocolConnector() API, JTAG_SetChainConfig(), and PSoC3_EraseRow()
*G	3823871	Nov 28, 2012	Added New API: HEX_WriteEEPROM(), DAP_ReadRaw(), DAP_WriteRaw(), JTAG_SetIR(), JTAG_ShiftDR() Obsolete API: PSoC3_SetJtagChainConfig()
*H	3903995	Feb 14, 2013	Updated Table 1-1. with "JTAG_EnumerateDevices" and JTAG_ShiftIR() Added API JTAG_ShiftIR().
*I	4093376	Aug 12, 2013	Removed "30 PSoC Programmer COM Interface Guide, Doc. # 001-45209 Rev. *E Command Descriptions" from the GetPower2() function.
*J	4272642	Feb 06, 2014	Updated 'C_Sharp SWD Example' and 'C_Sharp_EEPROM SWD Example' sections (added verification about whether the connected device is MiniProg3, DVK-Prog, FirstTouch, or Gen-FX2LP. Removed references to obsolete APIs.
*K	4535967	Oct 13, 2014	Updated the description of the following APIs: PSoC3_CheckSum(), PSoC3_ReadNvlArray(), and PSoC3_WriteNvlArray().
*L	4724490	April 15, 2015	Added "GetDeviceListBySiliconID" API description.
*M	4953757	Oct 08, 2015	Updated descriptions for PSoC4_EraseAll(string OUT strError) and PSoC4_WriteRow(IN rowID, nvector IN data, string OUT strError).

Document Revision History *(continued)*

Document Title: PSoC Programmer Component Object Model (COM) Interface Guide Document Number: 001-45209			
*N	5283060	Aug 26, 2016	Updated the PSoC4_GetFlashInfo command. Updated Table 1-1: Added APIs for GetRowsPerArrayInFlash API and FM0+ devices. Updated Chapter 2 with new APIs.
*O	5560941	Dec 20, 2016	Added APIs for PSoC 6 devices.
*P	5840674	July 24, 2017	Migrated to the new template.
*Q	5958513	Nov 11, 2017	Updated Commands: PSoC6_ProtectAll VerifyProtect Added Command: PSoC6_EraseRow
*R	6213098	Jun 25, 2018	No technical updates. Completing Sunset Review.
*S	6398278	Nov 30, 2018	Added new APIs: HEX_GetDataInRange, HEX_GetDataSizeInRange, FL_LoadElf, FL_IsApiExists, FL_ExecAPI, FL_SetRamForAlgorithms, FL_PrepareTarget, FL_ExecCmsisApiInit, FL_ExecCmsisApiUnInit, FL_ExecCmsisApiEraseSector, FL_ExecCmsisApiProgramPage, FL_ExecCmsisApiVerify, FL_ReleaseTarget. Fixed typos in few APIs.
*T	6678889	Sep 19, 2019	Added a note to the PSoC4_GetFlashInfo command