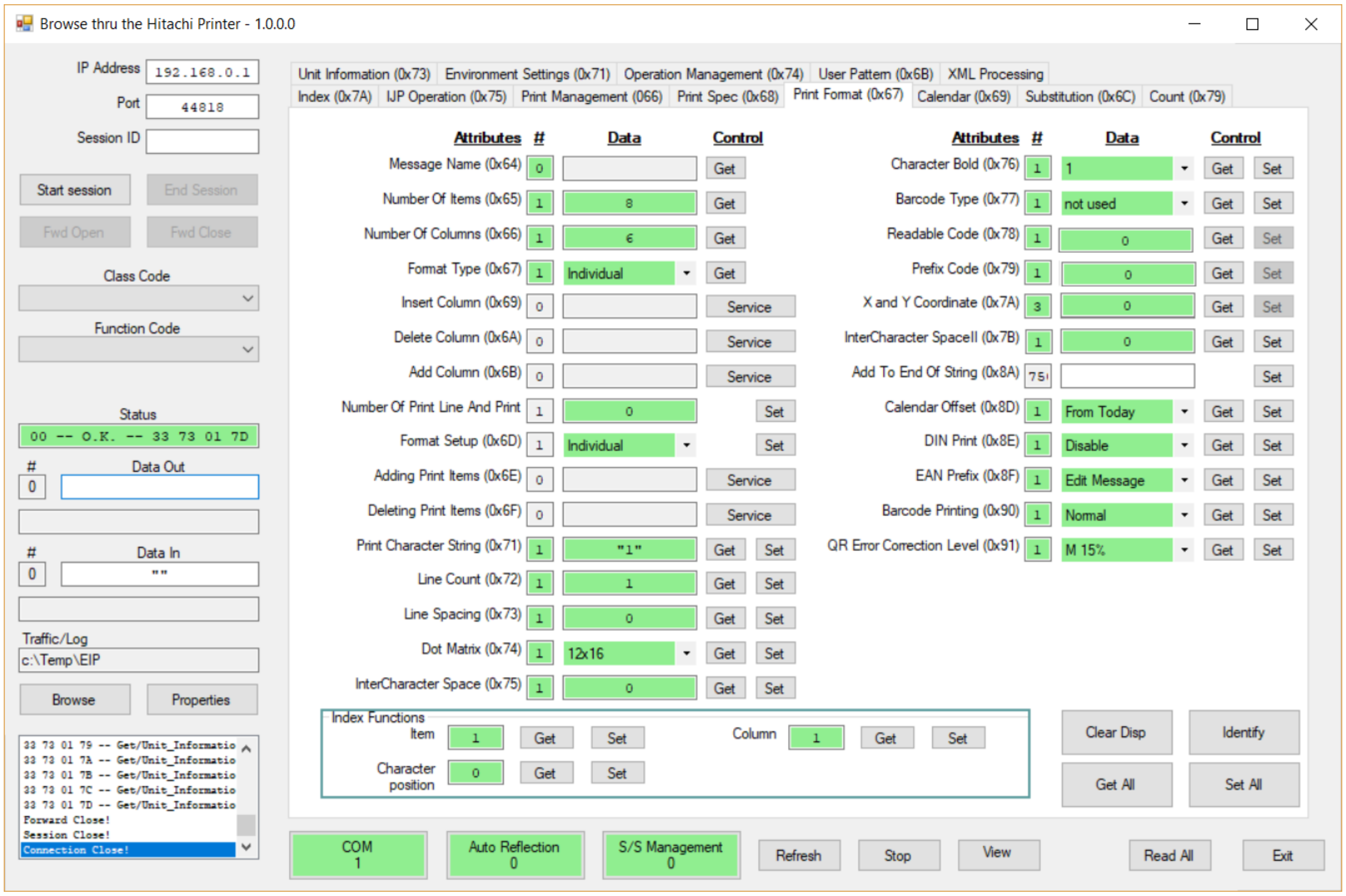
Hitachi UX Model 161
EtherNet/IP
Library and Browser

Test program up and running in minutes.

Interactive tool for all Get/Set/Service even faster.

XML with or without Serialization



IP Address	192.168.0.1
Port	44818
Session ID	
Class Code	
Function Code	
Status	00 -- O.K. -- 33 73 01 7D
# / Data Out	0 /
# / Data In	0 / ""
Traffic/Log	c:\Temp\EIP

```
33 73 01 79 -- Get/Unit_Informatio
33 73 01 7A -- Get/Unit_Informatio
33 73 01 7B -- Get/Unit_Informatio
33 73 01 7C -- Get/Unit_Informatio
33 73 01 7D -- Get/Unit_Informatio
Forward Close!
Session Close!
Connection Close!
```

Attributes	#	Data	Control	
Message Name (0x64)	0		Get	
Number Of Items (0x65)	1	8	Get	
Number Of Columns (0x66)	1	6	Get	
Format Type (0x67)	1	Individual	Get	
Insert Column (0x69)	0		Service	
Delete Column (0x6A)	0		Service	
Add Column (0x6B)	0		Service	
Number Of Print Line And Print	1	0		Set
Format Setup (0x6D)	1	Individual		Set
Adding Print Items (0x6E)	0		Service	
Deleting Print Items (0x6F)	0		Service	
Print Character String (0x71)	1	"1"	Get	Set
Line Count (0x72)	1	1	Get	Set
Line Spacing (0x73)	1	0	Get	Set
Dot Matrix (0x74)	1	12x16	Get	Set
InterCharacter Space (0x75)	1	0	Get	Set

Attributes	#	Data	Control	
Character Bold (0x76)	1	1	Get	Set
Barcode Type (0x77)	1	not used	Get	Set
Readable Code (0x78)	1	0	Get	Set
Prefix Code (0x79)	1	0	Get	Set
X and Y Coordinate (0x7A)	3	0	Get	Set
InterCharacter SpaceII (0x7B)	1	0	Get	Set
Add To End Of String (0x8A)	75			Set
Calendar Offset (0x8D)	1	From Today	Get	Set
DIN Print (0x8E)	1	Disable	Get	Set
EAN Prefix (0x8F)	1	Edit Message	Get	Set
Barcode Printing (0x90)	1	Normal	Get	Set
QR Error Correction Level (0x91)	1	M 15%	Get	Set

Index Functions

Item	1	Get	Set
Character position	0	Get	Set
Column	1	Get	Set

COM	1
Auto Reflection	0
S/S Management	0

Contents

1. Hitachi EtherNet/IP Library

The interface to the Hitachi UX Model 161 printer is implemented using Microsoft Visual Studio 2017 and C# Version 7.3. The forms are type Windows Application, the DLLs are type Class Library.

The source code, except for the “Test Drive” application, is not being released at this time.

There are five levels of support provided by the following services:

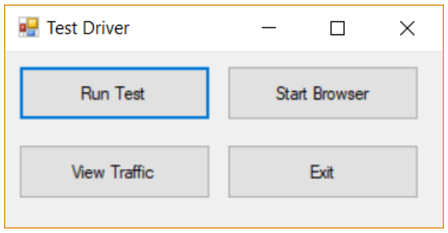
1. EIP – A DLL that hides all the EtherNet/IP Protocol and the internal structure of the Hitachi UX Model 161 printer. Since the I/O is Synchronous, it is useful for single printer applications.
2. Browser – A DLL that utilizes the EIP DLL to expose all the internal structure of the Hitachi UX Model 161 printer.
3. AsyncIO – A DLL that uses the EIP DLL to implement Concurrent Asynchronous I/O to one or more Hitachi printers.
4. Browser – The same functionality as item #2 but as a standalone application.
5. Test Drive – An application released in source form that will allow you to use the level 1, 2, and 3 DLLs above to build and test your own application.

These services provide all the tools needed to take full advantage of the power of the Hitachi Printers. They go from Shrink Wrapped to Custom Developed applications

2. Testing

So, how easy is it?

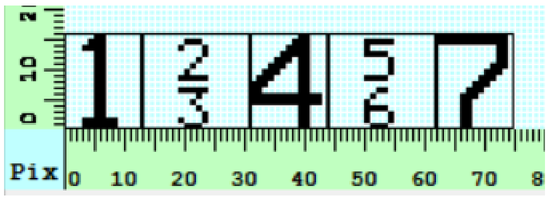
2.1. The Program



Consider you are developing an application that just wants to send a text message to the printer. You want to test sequences and, after they work, move them into your application.

Once you have built your test application, you need to add a reference to the DLL and some code for each of the buttons.

```
using System;
using System.Windows.Forms;
using EIP_Lib;

namespace H_EIP {
   public partial class TestEIP : Form {

      Browser browser = null;
      EIP EIP = null;

      public TestEIP() {
         InitializeComponent();
      }

      private void TestEIP_Load(object sender, EventArgs e) {
         // Comment out next line if browser not needed
         browser = new Browser("192.168.0.1", 44818, @"C:\Temp\EIP");
         if (browser == null) {
            EIP = new EIP("192.168.0.1", 44818, @"C:\Temp\EIP");
         } else {
            EIP = browser.EIP;
         }
      }

      private void cmdViewTraffic_Click(object sender, EventArgs e) {
         EIP.CloseExcelFile(true);
      }

      private void cmdStartBrowser_Click(object sender, EventArgs e) {
         browser?.ShowDialog();
      }

      private void cmdExit_Click(object sender, EventArgs e) {
         Application.Exit();
      }

      private void TestEIP_FormClosing(object sender, FormClosingEventArgs e) {
         EIP.CloseExcelFile(false);
         EIP = null;
         browser = null;
      }

   }
}
```

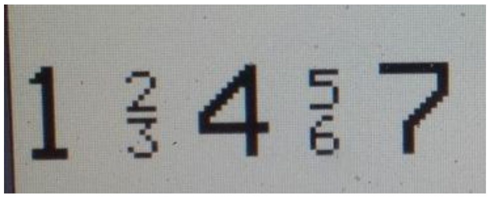


Here is an example of building a simple message. Just copy and paste it into your sample application.

```
// Create a simple message
private void cmdTest_Click(object sender, EventArgs e) {
   EIP.UseAutomaticReflection = true; // Speed up processing
   if (EIP.StartSession(true)) {      // Open a session
      if (EIP.ForwardOpen()) {        // open a data forwarding path
         try {
            EIP.DeleteAllButOne();                    // Clear the printer
            for (int i = 2; i <= 5; i++) {            // Add four more columns
               EIP.ServiceAttribute(ccPF.Add_Column);
            }
            EIP.SetAttribute(ccIDX.Item, 2);          // Stack column 2
            EIP.SetAttribute(ccPF.Line_Count, 2);
            EIP.SetAttribute(ccIDX.Item, 4);          // Stack column 4
            EIP.SetAttribute(ccPF.Line_Count, 2);
            for (int i = 1; i <= 7; i++) {
               EIP.SetAttribute(ccIDX.Item, i);       // Select item
               if (i == 1 || i == 4 || i == 7) {      // Set the font and text
                  EIP.SetAttribute(ccPF.Print_Character_String, $"{i}");
                  EIP.SetAttribute(ccPF.Dot_Matrix, "12x16");
               } else {
                  EIP.SetAttribute(ccPF.Print_Character_String, $" {i} ");
                  EIP.SetAttribute(ccPF.Dot_Matrix, "5x8");
               }
            }
         } catch (EIPIOException e1) {
            // In case of an EIP I/O error
            string name = $"{EIP.GetAttributeName(e1.ClassCode, e1.Attribute)}";
            string msg = $"EIP I/O Error on {e1.AccessCode}/{e1.ClassCode}/{name}";
            MessageBox.Show(msg, "EIP I/O Error", MessageBoxButtons.OK);
         } catch (Exception e2) {
            // You are on your own here
         }
      }
      EIP.ForwardClose(); // Must be outside the ForwardOpen if block
   }
   EIP.EndSession();      // Must be outside the StartSession if block
}
```

2.2. The Results

Once you click the Run Text button, the printer will update the display to:



If you want to see what was required to accomplish the task, just click “View Traffic”. All traffic is saved in an Excel Spreadsheet (You will need Microsoft Excel installed).

Access	Class	Attribute	#In	Data In	Raw In	#Out	Data Out	Raw Out
Get	Print format	Number Of Columns	1	5	05			
Set	Index	Automatic reflection				1	1	01
Set	Index	Column				2	1	00 01
Service	Print format	Delete Column						
Service	Print format	Delete Column						
Service	Print format	Delete Column						
Service	Print format	Delete Column						
Set	Index	Item				2	1	00 01
Set	Print format	Line Count				1	1	01
Service	Print format	Add Column						
Service	Print format	Add Column						
Service	Print format	Add Column						
Service	Print format	Add Column						
Set	Index	Item				2	2	00 02
Set	Print format	Line Count				1	2	02
Set	Index	Item				2	4	00 04
Set	Print format	Line Count				1	2	02
Set	Index	Item				2	1	00 01
Set	Print format	Print Character String				2	"1"	31 00
Set	Print format	Dot Matrix				1	12x16	07
Set	Index	Item				2	2	00 02
Set	Print format	Print Character String				4	" 2 "	20 32 20 00
Set	Print format	Dot Matrix				1	5x8	03
Set	Index	Item				2	3	00 03
Set	Print format	Print Character String				4	" 3 "	20 33 20 00
Set	Print format	Dot Matrix				1	5x8	03
Set	Index	Item				2	4	00 04
Set	Print format	Print Character String				2	"4"	34 00
Set	Print format	Dot Matrix				1	12x16	07
Set	Index	Item				2	5	00 05
Set	Print format	Print Character String				4	" 5 "	20 35 20 00
Set	Print format	Dot Matrix				1	5x8	03
Set	Index	Item				2	6	00 06
Set	Print format	Print Character String				4	" 6 "	20 36 20 00
Set	Print format	Dot Matrix				1	5x8	03
Set	Index	Item				2	7	00 07
Set	Print format	Print Character String				2	"7"	37 00
Set	Print format	Dot Matrix				1	12x16	07
Set	Index	Automatic reflection				1	0	00
Set	Index	Start Stop Management Flag				1	2	02

Access, Class, and Attribute lets you know about the request. . “#Out”, “Data Out”, and “Raw Out” describe the data sent to the printer. “#In”, “Data In”, and “Raw In” describe the response from the printer

There is actually much more information in the traffic spreadsheet. Here are the columns I deleted to make the previous table appear on one page.

Date/Time	Elapsed	Count OK	Data OK	Status/Path
19/03/31 13:50:53.2478	1971.8931	N/A	N/A	Connection Open!
19/03/31 13:50:53.2487	1971.8941	N/A	N/A	Session Open!
19/03/31 13:50:53.2537	1971.8991	N/A	N/A	Forward Open!
19/03/31 13:50:53.2867	0.0330	True	True	00 -- O.K. -- 33 67 01 66
19/03/31 13:50:53.3157	0.0290	True	True	00 -- O.K. -- 32 7A 01 65
		Deleted to save space		
19/03/31 13:50:54.0243	0.0160	True	True	00 -- O.K. -- 32 7A 01 65
19/03/31 13:50:54.0783	0.0540	True	True	00 -- O.K. -- 32 7A 01 64
19/03/31 13:50:54.1083	0.8545	N/A	N/A	Forward Close!
19/03/31 13:50:54.1083	0.8595	N/A	N/A	Session Close!
19/03/31 13:50:54.1093	0.8615	N/A	N/A	Connection Close!

“Date/Time” reflect the time when the request was made. “Elapsed” is the time between I/O requests. “Count OK” and “Data OK” indicates whether or not the request matched the EtherNet/IP specification for the printer. “Status/Path” shows the command sent and the response from the printer. The “Elapsed” time for Forward/Session/Comnnection Close is the time since the corresponding Open request.

The code in the Test Drive application performs the following:

- “`using EIP_Lib;`” After the DLLS have been added to the list of “References”, this allows references to the Class Library without needing to specify the Class Library Name Space on each reference.
- “`EIP = new EIP("192.168.0.1", 44818, @"C:\Temp\EIP");`” Creates a new instance of the EtherNet/IP Class. The three parameters are:
  - IP Address – String for TCP/IPv4 address of the printer.
  - IP Port – Int Port number.
  - Traffic Folder – Folder to use for saving the traffic to and from the printer. The traffic is saved in a Microsoft Excel Spreadsheet.
- “browser = `new HitachiBrowser("192.168.0.1", 44818, @"C:\Temp\EIP")`” Creates a new instance of the Browser Class. The three parameters are the same as the EIP Class.
- “`EIP.CloseExcelFile(true);`” This call is used to manage access to the Excel Traffic file. The parameters are:
  - true – Closes out the current traffic spreadsheet and opens it in Microsoft Excel. A new file will be opened for any new traffic.
  - false – Closes out the traffic Spreadsheet and throws it away. No new file is opened.
- “`browser.ShowDialog()`” Opens the Browser as a Windows Dialog Box.

The Browser and EIP can share the same traffic Excel Spreadsheet. Get the instance of EIP that the Browser generated.

```
// Comment out next line if browser not needed
browser = new Browser("192.168.0.1", 44818, @"C:\Temp\EIP");
if (browser == null) {
   EIP = new EIP("192.168.0.1", 44818, @"C:\Temp\EIP");
} else {
   EIP = browser.EIP;
}
```

You can add your own messages to the Excel Spreadsheet via:

```
EIP.FillInColData(“Your text”);
```

3. Passing Data

3.1. Connection to the device

EtherNet/IP Protocol defines two layers to control traffic to/from the device. They are the Session Layer and the Forward Layer. Hitachi adds one more layer with Auto Reflection and Start Stop Management.

- Session Layer – Is the connection to the device. It can be kept open for long periods. EIP implements it as:
    - StartSession() – Open a session and return the status.
    - EndSession() – Close a session. If StartSession() is called, EndSession() is also required.
- Forward Layer -- Is the path to the data inside the device. It can be kept open for short bursts of traffic. EIP implements it as:
    - ForwardOpen() – Open a path to the data and return a status.
    - ForwardClose() – Closes the path to the data. If ForwardOpen () is called, ForwardClose () is also required.
- Auto-Reflection – With Auto-Reflection set to 0, Traffic to the printer is executed immediately. With Auto-Reflection set to 1, the commands are "saved" and executed when Auto-Reflection is set to 0 and the Start Stop Management flag is set to 2. Note: If a Get Request is issued when Auto-Reflection is set to 1, the Get is ignored. There is no way for the EtherNet/IP Protocol to return the requested data later. Using Auto-Reflection to build a message is 4-times faster than not using Auto-Reflection.

3.2. Path to the data on the printer

There are two ways to manage the Session and Forward Layers:

- If only one command will be issued, don't bother with a Session of Forward. EIP will manage the Session/Forward for you.
- If multiple commands are to be sent, open a Session and a Forward around the commands.

If Sessions and Forwards are "Stacked". EIP manages the state of the layers. If a StartSession() call is made and a session is already open, the open session is used. If an EndSession() call is made and the session was already open when the corresponding StartSession() call was made, the Session is left open. The same applies to the Forward Layer.

```
private void cmdTest_Click(object sender, EventArgs e) {
   if (EIP.StartSession()) {    // Open a session
      if (EIP.ForwardOpen()) {  // open a data forwarding path

         int cols = EIP.GetAttribute(ccPF.Number_Of_Columns);
         EIP.SetAttribute(ccIDX.Automatic_reflection, 1); // Stack up all the operations

         EIP.SetAttribute(ccIDX.Item, 1);       // Set column 1(1 origin on Line Count)
         EIP.SetAttribute(ccPF.Line_Count, 1); // Set to 1 line

         EIP.SetAttribute(ccIDX.Automatic_reflection, 0);
         EIP.SetAttribute(ccIDX.Start_Stop_Management_Flag, 2);
      }
      EIP.ForwardClose(); // Must be outside the ForwardOpen if block
   }
   EIP.EndSession();      // Must be outside the StartSession if block
}
```

3.3. EtherNet/IP Traffic to the printer

The EtherNet/IP protocol defines four parameters for data passed through the Forward Layer. The Hitachi implementation of EtherNet/IP is documented in" EtherNetIP_UsersManual_4th.pdf" available from your Hitachi Distributor.

- Service – For Hitachi EIP, these are Get, Set, and Service (Enum "**`AccessCode`**")
- Class – For Hitachi, there are referred to as Class Codes (Enum "**`ClassCode`**")
- Instance – For Hitachi EIP, this is always 1.
- Attribute – For Hitachi EIP, their Enum names are:
  - **`ccPDM`** `// 0x66 Print data management function`
  - **`ccPF`** `// 0x67 Print format function`
  - **`ccPS`** `// 0x68 Print specification function`
  - **`ccCal`** `// 0x69 Calendar function`
  - **`ccUP`** `// 0x6B User pattern function`
  - **`ccSR`** `// 0x6C Substitution rules function`
  - **`ccES`** `// 0x71 Environment setting function`
  - **`ccUI`** `// 0x73 Unit Information function`
  - **`ccOM`** `// 0x74 Operation management function`
  - **`ccIJP`** `// 0x75 IJP operation function`
  - **`ccCount`** `// 0x79 Count function`
  - **`ccIDX`** `// 0x7A Index function`
- Data – The data associated with a service request is in the form of a byte array and is defined in the Hitachi Protocol. To shield the user from having to match the specification, the data format is all resolved by EIP. The user can build the data stream if needed. Format routines are provided and will always match the level of the Hitachi EtherNet/IP implementation.

EIP defines three methods to pass data to/from the Hitachi printer. It is up to the user to determine the best Session/Forward layers controls:

- GetAttribute – Issues a Get Service request.
- SetAttribute – Issues a Set Service request.
- ServiceAttribute – Issues a Service Service Request.

Consider out test program. It has Get, Set, and Service requests:

- A Get -- "`int cols = EIP.GetAttribute(ccPF.Number_Of_Columns);`" – This simply returns the number of columns by getting the data from the printer by specifying only the attribute "". No need for the user to know what to send or how to interpret the result.
- A Set – "`EIP.SetAttribute(ccIDX.Automatic_reflection, 1);`" – Again, sets the flag in the Index section of the printer. Again, the data structure is hidden from the user.
- A Service – "`EIP.ServiceAttribute(ccPF.Delete_Column);`" – The whole point of this in the program is to repeat the deleting of a certain column.

Further hiding of the data structure of the printer from the user. Consider something like setting the font. Depending on the make and model of the printer, the index associated with the 5X8 font varies. The UX, UX TUPI and UX InterNet/IP all have different mappings. EIP resolves that by allowing the following:

- `EIP.SetAttribute(ccIDX.Item, i);`
- `EIP.SetAttribute(ccPF.Print_Character_String, $" {i} ");`
- `EIP.SetAttribute(ccPF.Dot_Matrix, "5x8");`

The mapping of the Font to the target printer is resolved by EIP.

Status/Path	Access	Class	Attribute	#Out	Data Out	Raw Out
00 -- O.K. -- 32 7A 01 66	Set	Index	Item	2	2	00 02
00 -- O.K. -- 32 67 01 71	Set	Print_format	Print_Character_String	4	" 2 "	20 32 20 00
00 -- O.K. -- 32 67 01 74	Set	Print_format	Dot_Matrix	1	5x8	03

EIP hides the fact that:

- The index attribute is a 16-bit number.
- The text is UTF8 with a trailing null character.
- The setting for a 5X8 font is a 3 on this printer.

4. Using Human Readable Parameters

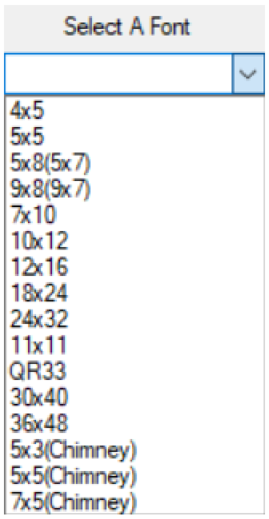
The Font names are not the only parameters that can be passed in Human Readable form. There are currently 23 different attributes that can be referenced symbolically.

```
public enum fmtDD {
   None = -1,
   Decimal = 0,
   EnableDisable = 1,
   DisableSpaceChar = 2,
   Hour12_24 = 3,
   CurrentTime_StopClock = 4,
   OnlineOffline = 5,
   None_Signal_1_2 = 6,
   UpDown = 7,
   ReadableCode = 8,
   BarcodeType = 9,
   NormalReverse = 10,
   M15Q25 = 11,
   EditPrint = 12,
   YesterdayToday = 13,
   FontType = 14,
   Orientation = 15,
   ProductSpeedMatching = 16,
   HighSpeedPrint = 17,
   TargetSensorFilter = 18,
   UserPatternFont = 19,
   Messagelayout = 20,
   ChargeRule = 21,
   TimeCount = 22,
}
```

The currently defined formats look like:

```
// Attribute DropDown conversion
static public string[][] DropDowns = new string[][] {
   new string[] { },                                        // 0 - Just decimal values
   new string[] { "Disable", "Enable" },                    // 1 - Enable and disable
   new string[] { "Disable", "Space Fill", "Character Fill" },  // 2 - Disable, space fill, character fill
   new string[] { "TwentyFour Hour", "Twelve Hour" },       // 3 - 12 & 24 hour
   new string[] { "Current Time", "Stop Clock" },           // 4 - Current time or stop clock
   new string[] { "Off Line", "On Line" },                  // 5 - Offline/Online
   new string[] { "None", "Signal 1", "Signal 2" },         // 6 - None, Signal 1, Signal 2
   new string[] { "Up", "Down" },                           // 7 - Up/Down
   new string[] { "None", "5X5", "5X7" },                   // 8 - Readable Code 5X5 or 5X7
   new string[] { "not used", "code 39", "ITF", "NW-7", "EAN-13", "DM8x32", "DM16x16", "DM16x36",
                  "DM16x48", "DM18x18", "DM20x20", "DM22x22", "DM24x24", "Code 128 (Code set B)",
                  "Code 128 (Code set C)", "UPC-A", "UPC-E", "EAN-8", "QR21x21", "QR25x25",
                  "QR29x29", "GS1 DataBar (Limited)", "GS1 DataBar (Omnidirectional)",
                  "GS1 DataBar (Stacked)", "DM14x14", },
                                                            // 9 - Barcode Types
   new string[] { "Normal", "Reverse" },                    // 10 - Normal/reverse
   new string[] { "M 15%", "Q 25%" },                       // 11 - M 15%, Q 25%
   new string[] { "Edit Message", "Print Format" },         // 12 - Edit/Print
   new string[] { "From Yesterday", "From Today" },         // 13 - From Yesterday/Today
   new string[] { "4x5", "5x5", "5x8(5x7)", "9x8(9x7)", "7x10", "10x12", "12x16", "18x24", "24x32",
                  "11x11", "QR33", "30x40", "36x48", "5x3(Chimney)", "5x5(Chimney)", "7x5(Chimney)"},
                                                            // 14 - Font Types
   new string[] { "Normal/Forward", "Normal/Reverse",
                  "Inverted/Forward", "Inverted/Reverse",},     // 15 - Orientation
   new string[] { "None", "Encoder", "Auto" },              // 16 - Product speed matching
   new string[] { "HM", "NM", "QM", "SM" },                 // 17 - High Speed Print
   new string[] { "Time Setup", "Until End of Print" },     // 18 - Target Sensor Filter
   new string[] { "4x5", "5x5", "5x8(5x7)", "9x8(9x7)", "7x10", "10x12", "12x16", "18x24", "24x32",
                  "11x11", "5x3(Chimney)", "5x5(Chimney)", "7x5(Chimney)", "QR33", "30x40", "36x48"  },
                                                            // 19 - User Pattern Font Types
   new string[] { "Individual", "Overall", "Free Layout" },     // 20 - Message Layout
   new string[] { "Standard", "Mixed", "Dot Mixed" },       // 21 - Charge Rule
   new string[] { "5 Minutes", "6 Minutes", "10 Minutes", "15 Minutes", "20 Minutes", "30 Minutes" },
                                                            // 22 - Time Count renewal period
};
```

Should the Hitachi Mappings change, the table will also change. The symbolic references will remain correct.



Just add the Combo Box on your screen and add the following code to your application. To take advantage of this table in your code, they can be referenced as follows:

```
cbFont.Items.AddRange(EIP.DropDowns[(int)fmtDD.FontType]);
```

The integer values returned by the printer are available in either the integer form or symbolic form. But, be careful. If you use the integer value returned by the printer to reference something in one of the lists, some entries are 0-origin index, others have 1-origin index. EIP can provide the origin value (See the Min and Max attributes in the AttrData Class Described Later).

5. Hitachi Data Layout

5.1. Types of Data

There are 16 different known layouts of information sent to and from the printer. There are more just referred to a "Struct" that I have not deciphered yet.

```
public enum DataFormats {
   None = -1,        // No formating
   Decimal = 0,      // Unsigned Decimal numbers up to 8 digits (Big Endian)
   DecimalLE = 1,    // Unsigned Decimal numbers up to 8 digits (Little Endian)
   SDecimal = 2,     // Signed Decimal numbers up to 8 digits (Big Endian)
   SDecimalLE = 3,   // Signed Decimal numbers up to 8 digits (Little Endian)
   UTF8 = 4,         // UTF8 characters followed by a Null character
   UTF8N = 5,        // UTF8 characters without the null character
   Date = 6,         // YYYY MM DD HH MM SS 6 2-byte values in Little Endian format
   Bytes = 7,        // Raw data in 2-digit hex notation
   XY = 8,           // x = 2 bytes, y = 1 byte
   N2N2 = 9,         // 2 2-byte numbers
   N2Char = 10,      // 2-byte number + UTF8 String + 0x00
   ItemChar = 11,    // 1-byte item number + UTF8 String + 0x00
   Item = 12,        // 1-byte item number
   GroupChar = 13,   // 1 byte group number + UTF8 String + 0x00
   MsgChar = 14,     // 2 byte message number + UTF8 String + 0x00
   N1Char = 15,      // 1-byte number + UTF8 String + 0x00
   N1N1 = 16,        // 2 1-byte numbers
}
```

The building of the byte arrays that are sent to/from the printer uses one of these formats specifications. The format might vary for the Get and Set of an Attribute. However, this is all hidden from the user. For users that just must dig deeper, here is how an Attribute is managed inside EIP.

Consider the Print Specification (Class Code 0x68) Attribute (Dot Matrix 0x74). Data within EIP is stored as:

```
new AttrData((byte)ccPF.Dot_Matrix, GSS.GetSet, false, 11,            // Dot Matrix 0x74
   new Prop(1, DataFormats.Decimal, 1, 16, fmtDD.FontType),           //   Data
   new Prop(0, DataFormats.Decimal, 0, 0, fmtDD.None),                //   Get
   new Prop(1, DataFormats.Decimal, 1, 16, fmtDD.FontType)),          //   Set
```

The data can be interpreted as:

- Line #1 == General Information:
  - The Enum value for the attribute.
  - Whether it has Get, Set, both Get and Set, or just service.
  - A flag to avoid this property because it causes a printer issue (reported to Hitachi).
  - Sort Order (sometimes I list by Attribute name, sometimes by attribute value).
- Line #2 == Description of the data in the printer
  - Number of bytes.
  - Format of the data.
  - Minimum value.
  - Maximum value.
  - Name conversion dropdown.
- Line #3 == Same as Line #2 but for the GetAttribute request.
- Line #4 == Same as Line #2 but for the SetAttribute request.

The same applies for service only functions. Consider the Print_data_management (Class Code 0x66) Attribute (Select Message 0x64).

```
 new AttrData((byte)ccPDM.Select_Message, GSS.Service, false, 9,        // Select Message 0x64
    new Prop(0, DataFormats.Decimal, 0, 0, fmtDD.None),                 //   Data
    new Prop(2, DataFormats.Decimal, 1, 2000, fmtDD.None)),             //   Service
```

Two bytes of decimal data must be sent on the request.

5.2. Accessing EtherNet/IP Data Descriptors

EIT provides access to this data thru two Classes:

The AttrData Class:

```
public ClassCode Class { get; set; }           // The class code is set when the dictionary is built
public byte Val { get; set; } = 0;             // The Attribute (Makes the tables easier to read)
public bool HasSet { get; set; } = false;      // Supports a Set Request
public bool HasGet { get; set; } = false;      // Supports a Get Request
public bool HasService { get; set; } = false;  // Supports a Service Request
public int Order { get; set; } = 0;            // Sort Order if Alpha Sort is requested
public bool Ignore { get; set; } = false;      // Indicates that the request will hang printer
// Four views of the printer data
public Prop Data { get; set; }     // As it appears in the printer
public Prop Get { get; set; }      // Data to be passed on a Get Request
public Prop Set { get; set; }      // Data to be passed on a Set Request
public Prop Service { get; set; }  // Data to be passed on a Service Request
```

The Prop Class

```
public int Len { get; set; }
public DataFormats Fmt { get; set; }
public long Min { get; set; }
public long Max { get; set; }
public fmtDD DropDown { get; set; }
```

To get all the information you might need about Getting/Setting the dot matrix for an item, use the following.

```
AttrData attr = EIP.GetAttrData(ccPF.Dot_Matrix);
```

6. One level deeper

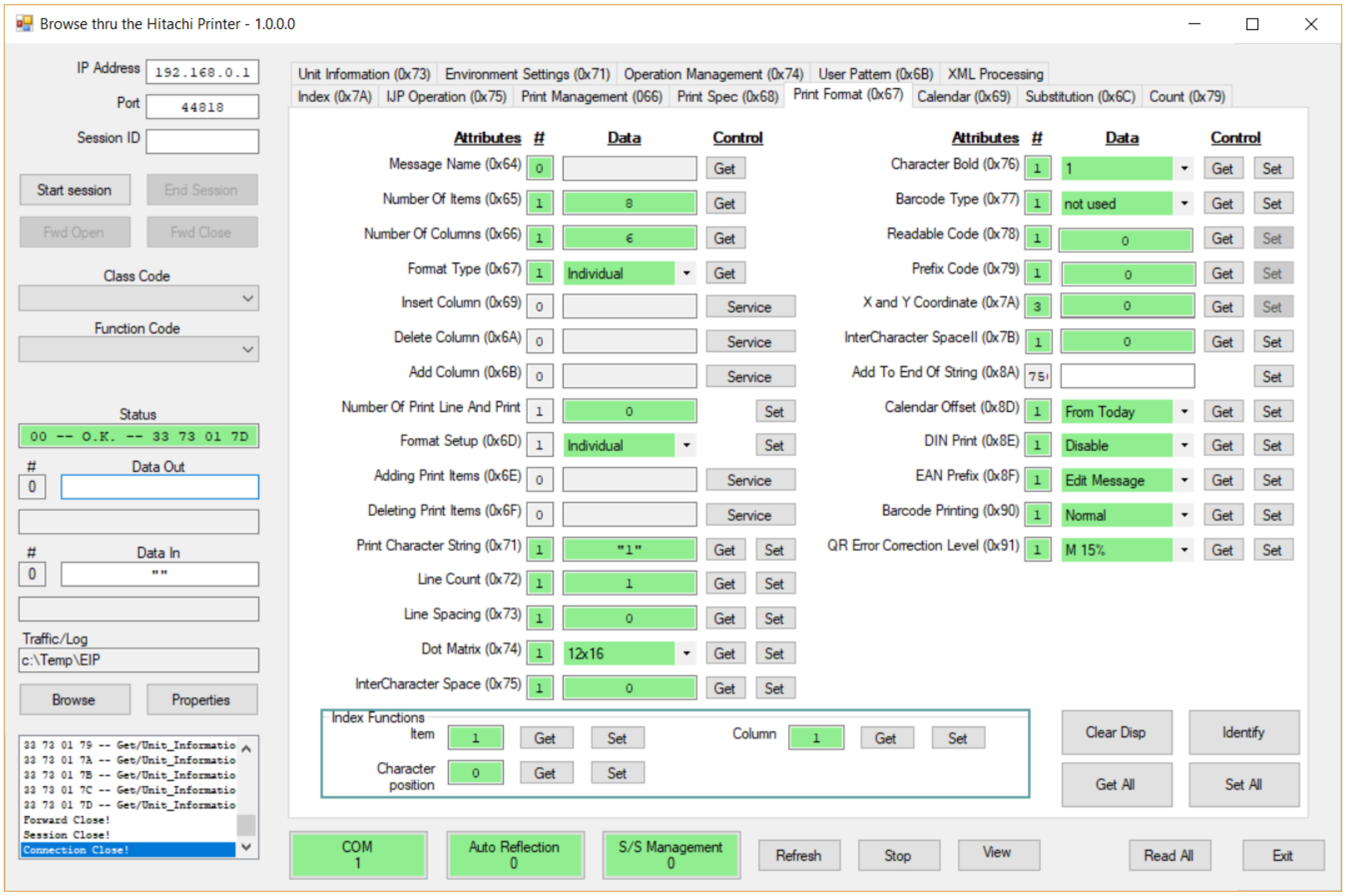
Sometimes there are things that just do not fit the mold. There is a need to build the byte array that will be sent as data to the printer. EIP Supports that scenario. For example

```
 AttrData attr = EIP.GetAttrData(ccPF.Print_Character_String);
 byte[] data1 = EIP.Encode.GetBytes("Hello World");                 // To UTF8 without a Null
 byte[] data2 = EIP.FormatOutput(attr.Set, " and Hello Dolly");    // To UTF8 with a Null
 EIP.SetAttribute(attr.Class, attr.Val, EIP.Merge(data1, data2)); // Merge the two arrays
```

This is an example of getting strings from different places to send to the printer in a single request. If the null character was at the end of data1, it would cause the printer to stop processing the message.

7. The Browser

Now that your test drive worked, it is time to take a serious look into the printer. Click the “Start Browser” button to look inside the printer


Browse thru the Hitachi Printer - 1.0.0.0

IP Address 192.168.0.1
Port 44818
Session ID

Status: 00 -- O.K. -- 33 73 01 7D

Data Out #: 0

Data In #: 0 ""

Traffic/Log: c:\Temp\EIP

```
33 73 01 79 -- Get/Unit_Informatio
33 73 01 7A -- Get/Unit_Informatio
33 73 01 7B -- Get/Unit_Informatio
33 73 01 7C -- Get/Unit_Informatio
33 73 01 7D -- Get/Unit_Informatio
Forward Close!
Session Close!
Connection Close!
```

Attributes	#	Data	Control
Message Name (0x64)	0		Get
Number Of Items (0x65)	1	8	Get
Number Of Columns (0x66)	1	6	Get
Format Type (0x67)	1	Individual	Get
Insert Column (0x69)	0		Service
Delete Column (0x6A)	0		Service
Add Column (0x6B)	0		Service
Number Of Print Line And Print	1	0	Set
Format Setup (0x6D)	1	Individual	Set
Adding Print Items (0x6E)	0		Service
Deleting Print Items (0x6F)	0		Service
Print Character String (0x71)	1	"1"	Get Set
Line Count (0x72)	1	1	Get Set
Line Spacing (0x73)	1	0	Get Set
Dot Matrix (0x74)	1	12x16	Get Set
InterCharacter Space (0x75)	1	0	Get Set

Attributes	#	Data	Control
Character Bold (0x76)	1	1	Get Set
Barcode Type (0x77)	1	not used	Get Set
Readable Code (0x78)	1	0	Get Set
Prefix Code (0x79)	1	0	Get Set
X and Y Coordinate (0x7A)	3	0	Get Set
InterCharacter SpaceII (0x7B)	1	0	Get Set
Add To End Of String (0x8A)	75		Set
Calendar Offset (0x8D)	1	From Today	Get Set
DIN Print (0x8E)	1	Disable	Get Set
EAN Prefix (0x8F)	1	Edit Message	Get Set
Barcode Printing (0x90)	1	Normal	Get Set
QR Error Correction Level (0x91)	1	M 15%	Get Set

Index Functions: Item 1, Column 1, Character position 0

COM 1 · Auto Reflection 0 · S/S Management 0

The index function is set to “1” so the Print Format display represents what you sent to Item 1 in the printer. The entries in Pink just say that what the printer returned did not match the EtherNet/IP Spec. Generally because they have no meaning. For example, the Layout is Individual so the X/Y coordinates are not used.

Play with it. You cannot hurt anything. There are 13 screens of Get/Set/Service buttons for experimenting. Just click a button and see how the printer changes.

8. XML (A work in progress)

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

The intent here is to describe a message that can be used to save a message from the printer as well as reload it into the printer. The goal is to be able to save a message in a format that is independent of printer model.

The XML document has two sections:

- Printer == Describes the settings in the printer needed to print the message
- Message == Describes the items in the printer

```
<Label Version="1">
  <Printer Make="Hitachi" Model="UX-D161W">
    <!--- Description of all printer wide parameters -->
  </Printer>
  <Message Layout="Individual">
    <!--- Description of all Items within the message -->
  </Message>
</Label>
```

8.1. Printer Section

The Printer section contains all parameters that can be set in the printer to control the printing of the message. It is broken into sub-sections along lines of printer capabilities.

```
<Printer Make="Hitachi" Model="UX-D161W">
  <PrintHead Orientation="Inverted/Forward" />
  <ContinuousPrinting RepeatInterval="0" PrintsPerTrigger="1" />
  <TargetSensor Filter="Until End of Print" SetupValue="50" Timer="0" />
  <CharacterSize Height="90" Width="10" />
  <PrintStartDelay Forward="96" Reverse="96" />
  <EncoderSettings HighSpeedPrinting="HM" Divisor="1" ExternalEncoder="None" />
  <InkStream InkDropUse="2" ChargeRule="Standard" />
  <Substitution Delimiter="/" StartYear="2019" Rule="1">
    <! Description of settings used for Time/Date Substitution used in message -->
  </Substitution>
  <Logos Folder="">
    <! Description of all user patterns used in the Message -->
  </Logos>
</Printer>
```

The Substitution Sub-Section contains the portions of the Substitution Rule file that are used in the message. The Logos Sub-Section describe the User Patterns used in the message.

8.2. Message Section

The message section describes the Rows and Columns of Items that make up the message. Thus, this is for Individual layout only for the moment.

```
<Message Layout="Individual">
  <Column InterLineSpacing="2">
    <Item>
      <! Description of an Item within a Column of a Message -->
    </Item>
  </Column>
</Message>
```

8.3. Item Section

The Item Section describes each item with a message. There are one or more Items within a Column. Since Column and Row numbers are assigned dynamically in the printer, the column and row number are not part of the specification.

```
<Item>
  <Font IncreasedWidth="1" InterCharacterSpace="1" DotMatrix="5x8(5x7)" />
  <BarCode />
  <Date Block="1">
    <! Description of all settings in a Calendar Block -->
  </Date>
  <Counter Block="1">
    <! Description of all settings in a Counter Block -->
  </Counter>
  <Text>Shift {{E}}</Text>
</Item>
```

The Item Section always contains a Font and Text Sub-Section. An optional Date Sub-Section or a Counter Sub-Section can also be provided if needed. However, Date and Counter Sub-Sections cannot appear in the same Item.

8.4. Date Sub-Section

The Date Sub-Section provides control of all settings in the printer Calendar Control. Since multiple Calendar Objects can be used in a message, multiple Date Sub-Sections can appear in an Item.

The Date Sub-Section has multiple parts. However, the printer limits which ones can be used together in a single item.

```
<Date Block="1" SubstitutionRule="1" RuleName="">
  <Offset Year="0" Month="0" Day="0" Hour="0" Minute="0" />
  <ZeroSuppress Year="Disable" Month="Disable" Day="Space Fill" />
  <Substitute Month="Enable" />
  <Shift ShiftNumber="1" StartHour="0" StartMinute="0" ShiftCode="D" />
  <TimeCount Interval="30 Minutes" Start="A1" End="X2" ResetTime="6" ResetValue="A1" />
</Date>
```

8.5. Counter Sub-Section

The Counter Sub-Section provides control of all settings in the printer Counter Control. Since multiple Counter Objects can be used in a message, multiple Counter Sub-Sections can appear in an Item.

```
<Counter Block="1">
  <Range Range1="000000" Range2="999999" JumpFrom="000199" JumpTo="000300" />
  <Count InitialValue="000001" Increment="2" Direction="Down" ZeroSuppression="Enable" />
  <Reset Type="Signal 1" Value="000001" />
  <Misc UpdateUnit="1" UpdateIP="0" Multiplier="" />
</Counter>
```

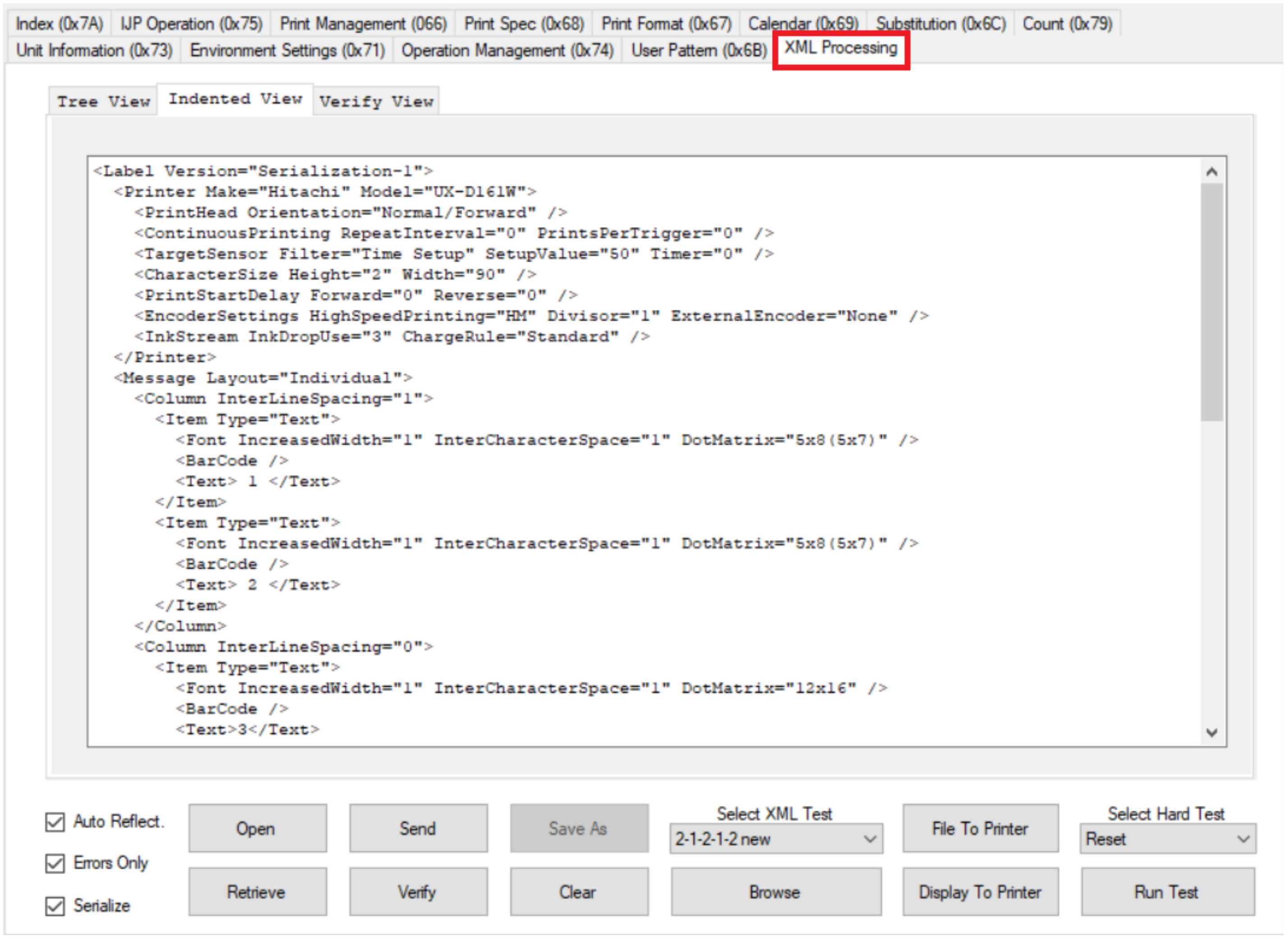
8.6. XML Example

The example shown in Section 2 would have the following representation in XML.

```xml
<Label Version="Serialization-1">
  <Printer Make="Hitachi" Model="UX-D161W">
    <PrintHead Orientation="Normal/Forward" />
    <ContinuousPrinting RepeatInterval="0" PrintsPerTrigger="0" />
    <TargetSensor Filter="Time Setup" SetupValue="50" Timer="0" />
    <CharacterSize Width="90" Height="2" />
    <PrintStartDelay Forward="0" Reverse="0" />
    <EncoderSettings HighSpeedPrinting="HM" Divisor="1" ExternalEncoder="None" />
    <InkStream InkDropUse="3" ChargeRule="Standard" />
  </Printer>
  <Message Layout="Individual">
    <Column InterLineSpacing="0">
      <Item>
        <Font InterCharacterSpace="1" IncreasedWidth="1" DotMatrix="12x16" />
        <BarCode />
        <Text>1</Text>
      </Item>
    </Column>
    <Column InterLineSpacing="0">
      <Item>
        <Font InterCharacterSpace="1" IncreasedWidth="1" DotMatrix="5x8(5x7)" />
        <BarCode />
        <Text> 2 </Text>
      </Item>
      <Item>
        <Font InterCharacterSpace="0" IncreasedWidth="1" DotMatrix="5x8(5x7)" />
        <BarCode />
        <Text> 3 </Text>
      </Item>
    </Column>
    <Column InterLineSpacing="0">
      <Item>
        <Font InterCharacterSpace="1" IncreasedWidth="1" DotMatrix="12x16" />
        <BarCode />
        <Text>4</Text>
      </Item>
    </Column>
    <Column InterLineSpacing="0">
      <Item>
        <Font InterCharacterSpace="1" IncreasedWidth="1" DotMatrix="5x8(5x7)" />
        <BarCode />
        <Text> 5 </Text>
      </Item>
      <Item>
        <Font InterCharacterSpace="0" IncreasedWidth="1" DotMatrix="5x8(5x7)" />
        <BarCode />
        <Text> 6 </Text>
      </Item>
    </Column>
    <Column InterLineSpacing="0">
      <Item>
        <Font InterCharacterSpace="1" IncreasedWidth="1" DotMatrix="12x16" />
        <BarCode />
        <Text>7</Text>
      </Item>
    </Column>
  </Message>
</Label>
```

9. XML Browsing

This section of the Hitachi Browser exposes the automation of the XML processing.



```
<Label Version="Serialization-1">
  <Printer Make="Hitachi" Model="UX-D161W">
    <PrintHead Orientation="Normal/Forward" />
    <ContinuousPrinting RepeatInterval="0" PrintsPerTrigger="0" />
    <TargetSensor Filter="Time Setup" SetupValue="50" Timer="0" />
    <CharacterSize Height="2" Width="90" />
    <PrintStartDelay Forward="0" Reverse="0" />
    <EncoderSettings HighSpeedPrinting="HM" Divisor="1" ExternalEncoder="None" />
    <InkStream InkDropUse="3" ChargeRule="Standard" />
  </Printer>
  <Message Layout="Individual">
    <Column InterLineSpacing="1">
      <Item Type="Text">
        <Font IncreasedWidth="1" InterCharacterSpace="1" DotMatrix="5x8(5x7)" />
        <BarCode />
        <Text> 1 </Text>
      </Item>
      <Item Type="Text">
        <Font IncreasedWidth="1" InterCharacterSpace="1" DotMatrix="5x8(5x7)" />
        <BarCode />
        <Text> 2 </Text>
      </Item>
    </Column>
    <Column InterLineSpacing="0">
      <Item Type="Text">
        <Font IncreasedWidth="1" InterCharacterSpace="1" DotMatrix="12x16" />
        <BarCode />
        <Text>3</Text>
```

Controls:

- Auto Reflect == Turns on Automatic Auto-Reflection for processing SEND requests
- Errors Only == Tells Verify to only output errors encounter.
- Serialize == Use the Serialized Interface (vs the XML Interface) for Send, Retrieve, and Verify requests.
- Open == Use file browser to open an XML file
- Retrieve == Retrieve the current Message from the printer
- Send == Send the currently open XML file to the printer
- Verify == Verify the currently loaded XML file against the contents of the printer
- Save As == Save the currently loaded XML file
- Clear == Clear the display
- Select XML Text == Pick an XML file from a list of existing XML files. Browse to select the folder.
- File to Printer == Same as Send
- Display to Printer == The Indented view can be modified. Send its contents to the printer
- Select Hard Test == A set of hard coded tests exist as examples. Select and Run Test.

10. XML Processing

XML files can be sent to the printer, retrieved from the printer, or verified that the XML in the printer matched an XML file.

10.1. To Printer

The XML file can be passed to the printer in one of three ways: File Name, XML Text, of XML Document.

```
bool success = EIP.SendXmlToPrinter(XMLFileName);
bool success = EIP.SendXmlToPrinter(XMLText);


using System.Xml;
XmlDocument xmlDoc = new XmlDocument() { PreserveWhitespace = true };
xmlDoc.LoadXml(XMLText);
bool success = EIP.SendXmlToPrinter(XMLDoc);
```

10.2. From Printer

The XML can be retrieved from the printer:

```
string XMLText = EIP.RetrieveXML();
```

10.3. Verify Contents

To make sure the printer accepted all commands, a call can be made to check the contents of an XML file with the contents of the printer.

```
bool success = EIP.VerifyXmlVsPrinter(xml); // File Name, string, or XMLDoc
```

The verification process writes the results into the Excel Traffic Spreadsheet.

11. Serialization Processing

Serialization is a late addition to the printer interface. It provides all the same interfaces as the XML processing.

11.1. To Printer

```
bool success = EIP.SendXMLAsSerialization(XMLFileName);
bool success = EIP.SendXMLAsSerialization(XMLText);


using System.Xml;
using System.Xml.Serialization;
XmlSerializer serializer = new XmlSerializer(typeof(Lab));
using (TextReader reader = new StringReader(XMLText)) {
   Lab Lab = (Lab)serializer.Deserialize(reader);
   EIP.SendXMLAsSerialization(Lab);
}
```

11.2. From Printer

The XML can be retrieved from the printer: Under development.

12. Serialization (Full Definition)

12.1. Root Class

The root would be a “Label”.

```
[XmlRoot("Label", IsNullable = false)]
public class Lab {
   [XmlAttribute]
   public string Version;   // Keep track of version to allow for changes
   public Printer Printer;  // Information that pertains to the printer
   public Msg Message;      // Information that pertains to the message
}
```

12.2. Printer Class

The printer section covers the printer setup for printing a message. It is broken into subsections (just because I did it that way for cijConnect). They may need to be collapsed into the Printer Class.

```
public class Printer {
   [XmlAttribute]
   public string Make;
   [XmlAttribute]
   public string Model;
   public PrintHead PrintHead;
   public ContinuousPrinting ContinuousPrinting;
   public TargetSensor TargetSensor;
   public CharacterSize CharacterSize;
   public PrintStartDelay PrintStartDelay;
   public EncoderSettings EncoderSettings;
   public InkStream InkStream;
   public Substitution Substitution;
   public Logos Logos;
}
```

When moving from printer to printer, the print head orientation may change.

```
public class PrintHead {
   [XmlAttribute]
   public string Orientation;
}
```

No idea how often this is used.

```
public class ContinuousPrinting {
   [XmlAttribute]
   public string RepeatInterval;
   [XmlAttribute]
   public string PrintsPerTrigger;
}
```

No idea how often this is used.

```
public class TargetSensor {
   [XmlAttribute]
   public string Filter;
   [XmlAttribute]
   public string SetupValue;
   [XmlAttribute]
   public string Timer;
}
```

Width can only be sent for Time Based printing

```
public class CharacterSize {
   [XmlAttribute]
```

```
      public string Height;
      [XmlAttribute]
      public string Width;
   }
```

Do not know the rules for setting Reverse Delay.

```
   public class PrintStartDelay {
      [XmlAttribute]
      public string Forward;
      [XmlAttribute]
      public string Reverse;
   }
```

Divisor is only used for encoder based printing. Maybe width should be here

```
   public class EncoderSettings {
      [XmlAttribute]
      public string HighSpeedPrinting;
      [XmlAttribute]
      public string Divisor;
      [XmlAttribute]
      public string ExternalEncoder;
   }
```

I think there are more settings like Interlaced and Single Scan.

```
   public class InkStream {
      [XmlAttribute]
      public string InkDropUse;
      [XmlAttribute]
      public string ChargeRule;
   }
```

The intent here is to be aple to carry along all logos that are used by the message. Specifying the folder would allow the handled to get a fresh copy of the logo. There is nowhere in the printer to store the folder name.

```
   public class Logos {
      [XmlAttribute]
      public string Folder;
      [XmlElement("Logo")]
      public Logo[] Logo;
   }
```

Substitution and Substitution Rule are a work-in-progress

```
public class Substitution {
   [XmlAttribute]
   public string Delimiter;
   [XmlAttribute]
   public string StartYear;
   [XmlAttribute]
   public string Rule;

   [XmlElement("Rule")]
   public SubstitutionRule[] SubRule;

}

public class SubstitutionRule {
   [XmlAttribute]
   public string Type;
   [XmlAttribute]
   public string Base;

   public string Text;
}
```

The layout and location allow messages to be sent to the printer. If a messahe is retriened from the printer, the bitmaps would be stored here.

```
public class Logo {
   [XmlAttribute]
   public string Layout;
   [XmlAttribute]
   public string Location;
   [XmlAttribute]
   public string RawData;
   [XmlAttribute]
   public string FileName;
}
```

12.3. Message Class

The message class describes the message. This layout only describes the “Individual” lauout. It may have to be expanded for “Overall” layout. It definitely will need to be expanded for “Free” Layout.

The current implementation uses the “Add Column” to allocate a column and “Line Count” to set the number of items in the column. The Column Number and Item Number is assigned by the printer so it is not included in the specification of a column or item. There can be multiple columns in a message.

```
public class Msg {
   [XmlAttribute]
   public string Layout;
   [XmlElement("Column")]
   public Column[] Column;
}
```

A column is made up of multiple items.

```
public class Column {
   [XmlAttribute]
   public string InterLineSpacing;
   [XmlElement("Item")]
   public Item[] Item;
}
```

A item contains all the information stored in the printer to descibe one item. Date and Counter Classes are Arrays to handle the case where multiple calendar and count are included in a single text string. It might be a good idea to allow multiple Text Classes to handle very long strings. .The Location objects is used for internal processing but is not included in the XML since all the values are assigned by the printer and not by the use. The data and counter cannot appear in the same item.

```
public class Item {
   [XmlAttribute]
   public string Type;
   public FontDef Font;
   public BarCode BarCode;
   [XmlElement("Date")]
   public Date[] Date;
   [XmlElement("Counter")]
   public Counter[] Counter;
   public string Text;
   [XmlIgnore]
   public Location Location;
}
```

The Location objects is used for internal processing but is not included in the XML since all the values are assigned by the printer and not by the user.

```
public class Location {
   public int Row;     // 0-Origin
   public int Col;     // 0-Origin
   public int Index;   // 0-Origin
   public int X;       // 0-Origin
   public int Y;       // 0-Origin
   public int calStart = 0;
   public int calCount = 0;
   public int countStart = 0;
   public int countCount = 0;
}
```

The Font class describes the unique characteristics of the font.

```
public class FontDef {
   [XmlAttribute]
   public string IncreasedWidth;
   [XmlAttribute]
   public string InterCharacterSpace;
   [XmlAttribute]
   public string DotMatrix;
}
```

The Barcode Class is a work in progress. Maybe include it with Font Class?

```
public class BarCode {
   [XmlAttribute]
   public string HumanReadableFont;
   [XmlAttribute]
   public string EANPrefix;
   [XmlAttribute]
   public string DotMatrix;
}
```

12.4. Counter Class

The counter class is an object within an item. It was arbitrarily broken into sub-classes. All open for discussion.

```
public class Counter {
   [XmlAttribute]
   public int Block;

   public Range Range;
   public Count Count;
   public Reset Reset;
   public Misc Misc;
}

public class Range {
   [XmlAttribute]
   public string Range1;
   [XmlAttribute]
   public string Range2;
   [XmlAttribute]
   public string JumpFrom;
   [XmlAttribute]
   public string JumpTo;
}

public class Count {
   [XmlAttribute]
   public string InitialValue;
   [XmlAttribute]
   public string Increment;
   [XmlAttribute]
   public string Direction;
   [XmlAttribute]
   public string ZeroSuppression;
}
```

```
public class Reset {
   [XmlAttribute]
   public string Type;
   [XmlAttribute]
   public string Value;
}

public class Misc {
   [XmlAttribute]
   public string UpdateUnit;
   [XmlAttribute]
   public string UpdateIP;
   [XmlAttribute]
   public string Multiplier;
   [XmlAttribute]
   public string ExternalCount;
   [XmlAttribute]
   public string CountSkip;
}
```

12.5. Date Class

The Date class is an object within an item. It was arbitrarily broken into sun-classes. All open for discussion.

```
public class Date {
   [XmlAttribute]
   public int Block;
   [XmlAttribute]
   public string SubstitutionRule;
   [XmlAttribute]
   public string RuleName;
   public Offset Offset;
   public ZeroSuppress ZeroSuppress;
   public Substitute Substitute;
   public TimeCount TimeCount;
   [XmlElement("Shift")]
   public Shift[] Shift;
}

public class Offset {
   [XmlAttribute]
   public string Year;
   [XmlAttribute]
   public string Month;
   [XmlAttribute]
   public string Day;
   [XmlAttribute]
   public string Hour;
   [XmlAttribute]
   public string Minute;
}
```

```
public class ZeroSuppress {
   [XmlAttribute]
   public string Year;
   [XmlAttribute]
   public string Month;
   [XmlAttribute]
   public string Day;
   [XmlAttribute]
   public string Hour;
   [XmlAttribute]
   public string Minute;
   [XmlAttribute]
   public string Week;
   [XmlAttribute]
   public string DayOfWeek;
}

public class Substitute {
   [XmlAttribute]
   public string Year;
   [XmlAttribute]
   public string Month;
   [XmlAttribute]
   public string Day;
   [XmlAttribute]
   public string Hour;
   [XmlAttribute]
   public string Minute;
   [XmlAttribute]
   public string Week;
   [XmlAttribute]
   public string DayOfWeek;
}
```

Shift was included as a sub-section of the Date Class since it results in a Calendat Object being generated..

```
public class Shift {
   [XmlAttribute]
   public int ShiftNumber;
   [XmlAttribute]
   public string StartHour;
   [XmlAttribute]
   public string StartMinute;
   [XmlAttribute]
   public string EndHour;
   [XmlAttribute]
   public string EndMinute;
   [XmlAttribute]
   public string ShiftCode;
}
```

Time Count was included as a sub-section of the Date Class since it results in a Calendar Object being generated..

```
public class TimeCount {
   [XmlAttribute]
   public string Interval;
   [XmlAttribute]
   public string Start;
   [XmlAttribute]
   public string End;
   [XmlAttribute]
   public string ResetTime;
   [XmlAttribute]
   public string ResetValue;
}
```

13. Current Thoughts on Printer Composition

Sections I am wondering about:

13.1. Substitution Class

The intent here is to be able to reload the substitution files in the printer. Probably need to add the Name of the File.

The Delimiter cannot be stored and retrieved from the printer so may need to be fixed.

The Start Year was placed as an attribute. Need a rational place to put it.

The Rule Class layout is just to allow the something like Month to be subdivided.

13.2. Logo Class

Needs lots of thought here. Need to see how the printer responds to reading a free layout logo.

13.3. Printer Class

The Printer Class is optional. If it is missing, no settings will change in the printer. The sub-divisions are arbitrary. More thought required. Things like Time Based vs. Encoder based.

14. Current Thoughts on Message Composition

Sections I am wondering about:

14.1. Message Class

The Message Class is optional. This can be used just to load the Substitution Rules.

The Message Class only specifies Layout (Free, Overall, Individual). It contains the collection of Columns. If the Message Class is included, there must be at least one column within the message. Within the printer, there is always at least one Column and one Item.

14.2. Column Class

Column Class only specifies Inter-line Spacing. You cannot specify a Column Number since the printer assigns column numbers

14.3. Item Class.

Item Class currently specifies a Type (Text, Date, or Counter) but that is also assigned by the printer based on the text in the message. That should be removed.

The Item Class always contains a Font Class, A Barcode Class, and a Text Class. It might work to move the Font Information into the Item Class as Attributes.

The Barcode Class could stay as it is or be moved into the Item Class as attributes.

The Text Class needs to be defined as a collection so very long text messages could be loaded. Care must be taken on breaking the text into sections as that can cause messages text to be discarded in the printer.

14.4. Date Class

The Date Class is broken into five sub-classes. The Date Class is a collection since multiple Calendar Blocks can be assigned to a single Item Class.

The Block Number is a 1-origin index of the Calendar Blocks assigned to this Item Class. The Calendar Starting Block and the number of blocks is assigned by the printer.

14.5. Shift Class

The Shift Class has an entry for each shift. The Shift Class is currently a collection. Should it be moved into another Sub-class?

14.6. Counter Class

The Counter Class is was arbitrarily broken into sub-Classes. I do not use them enough to know what makes sense.