Interface between RTOS target and an Industrial System: Developing USB Digital I/O Module

Benjamin O’Brien

Embry-Riddle Aeronautical University

Graduate Research Project

3/22/13

Revision 0.1

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Signature** | **Date** |
| **Student** | Benjamin O’Brien |  |  |
| **GRP Advisor** | Dr. Andrew Kornecki |  |  |
| **MSE Coordinator / Department Chair** |  |  |  |
| **Reviewer** | Dr. Shou Pang |  |  |

**Revision History**

|  |  |  |
| --- | --- | --- |
| **Date** | **Revision** | **Description** |
| 3/22/13 | 0.1 | Initial Report Layout |
| 3/31/13 | 0.2 | Initial Content added |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Table of Contents

[Table of Figures iv](#_Toc353017705)

[1 Introduction 1](#_Toc353017706)

[2 Objective 1](#_Toc353017707)

[3 Background 1](#_Toc353017708)

[3.1 Literary Exploration 2](#_Toc353017709)

[3.2 Existing Solutions 3](#_Toc353017710)

[3.2.1 Festo-Didatic EasyVeep 3](#_Toc353017711)

[3.2.2 Other Solutions (More on this later) 3](#_Toc353017712)

[4 Methodology 3](#_Toc353017713)

[4.1 Using EasyVeep 3](#_Toc353017714)

[4.1.1 Reversing the EasyPort Protocol 3](#_Toc353017715)

[4.1.2 Implementing an EasyPort 5](#_Toc353017716)

[4.1.3 Implementing a Custom EasyVeep 7](#_Toc353017717)

[5 Budget and Schedule 7](#_Toc353017718)

[6 Resources 8](#_Toc353017719)

[7 Appendices 9](#_Toc353017720)

[7.1 Reversing EasyVeep 9](#_Toc353017721)

[7.2 Arduino EasyPort Code 14](#_Toc353017722)

[7.3 Arduino Flow Control Failures 15](#_Toc353017723)

[7.4 Desktop EasyPort Emulator 18](#_Toc353017724)

[7.5 EasyVeep Model Design and Interaction 23](#_Toc353017725)

[7.5.1 SWF Dump Results 24](#_Toc353017726)

# Table of Figures

[Figure 1: Proposed System Configuration 1](#_Toc353017734)

[Figure 2 : EasyPort Protocol Sequence Diagram 5](file:///C:\Users\zagi\SkyDrive\GRP\docs\worddoc\GRP_Report.docx#_Toc353017735)

[Figure 3 : A screenshot of the desktop application implementing the EasyPort protocol. 6](#_Toc353017736)

[Figure 4 : EasyPort COM Port Discovery Code 10](#_Toc353017737)

[Figure 5 : EasyPort DCB Parameters 11](#_Toc353017738)

[Figure 6 : EasyPort.OCX Snippet Returning EasyPort Inputs 12](#_Toc353017739)

[Figure 7: EasyPort.OCX Snippet Storing Input Values 13](#_Toc353017740)

**Table of Tables**

[Table 1 : EasyPort Protocol Listing 4](#_Toc353017746)

[Table 2 : Important SWF Variable Names 23](#_Toc353017747)

[Table 3 : Process Information for Boat Lock Process 24](#_Toc353017748)

[Table 4 : Process Information for Parking Lot Barrier Process 25](#_Toc353017749)

[Table 5 : Process Information for Hot Water Tank Process 26](#_Toc353017750)

[Table 6 : Process Information for Eleavtor Process 27](#_Toc353017751)

[Table 7 : Process Information for Seven Segment Display Process 28](#_Toc353017752)

[Table 8 : Process Information for Wine Bottling Process 29](#_Toc353017753)

[Table 9 : Process Information for Ball Sorting System 30](#_Toc353017754)

[Table 10: Process Information for Train Crossing Process 31](#_Toc353017755)

# Introduction

This Graduate Research Project (GRP) was conducted in Spring 2013 at Embry Riddle Aeronautical University, under the supervision of Dr. Andrew Kornecki. The proposed area of research involves examining freely available real time industrial system simulation software and creating hardware device and supporting software necessary to allow students to control and interact with the simulated industrial systems. Such hardware and software would increase the number of systems with which the RTOS hardware targets could interact, allowing the Real-Time Systems (RTS) course students to gain more experience with controlling systems in real time.

# Objective

The main objective of this research is to construct a low-cost, working solution that will allow students taking the Real-Time Systems course to interact with and control system simulations provided by the EasyVeep software or a similar system. Currently only Festo’s proprietary EasyPort hardware is capable of interacting with the targeted software, but the EasyPort is an inadequate solution for various reasons detailed elsewhere in this report. Through various techniques including the analysis of available EasyPort specifications, inspection of system operation, and reverse engineering techniques it is possible to create a viable hardware and software solution to allow the RTOS target to interact with various system simulations.



Figure 1: Proposed System Configuration

# Background

The following sections include detailed descriptions of research that was performed during the proposal and initial development stages of the GRP. This section primarily documents the state of existing work in this field of study. This section also establishes the selected scope of work which is detailed in the Methodology section.

## Literary Exploration

The main motivation for this project was the discovery of the EasyVeep software. A major hurdle in achieving the objective of real time industrial simulation is creating accurate models. EasyVeep provides many already constructed simulations for free. The cost of the required EasyPort is a setback, however. Therefore before beginning a literary review was performed to determine if there was a more appropriate solution or a reasonably priced alternative to the EasyPort.

Initial research began using the IEEE Xplore database, which returned marginal results when searching for Festo and no results when searching for EasyVeep or EasyPort. The results found when searching for just Festo were more directed towards the use of Festo PLCs and other, non-free, simulation tools for use as training aids. The typical papers found that the tools were in fact quite useful for instruction, but published research into cheaper and more open source hardware options that work with the EasyVeep software was not found (Li, Lu and ZhiPing).

The search was expanded using Google Scholar. A conference paper was found discussing various software suites used for virtual training in control and automation. The author found that while there are various solutions, “All these tools require licensing whose price various according to the tool” and that EasyVeep was only one of a very few “final applications for practicing” control and automation (Izaguirre).This further reinforced the importance of basing the solution off of EasyVeep.

Since no moderately priced commercial solution was found, a search was performed to find similar academic projects. Limited information was found from a group called Fast Forward Technologies (Wolowicz and Knoll). The group was created as a result of a senior level design project at the University of Victoria to replicate an EasyPort. Their approach, however, was not to replicate the EasyPort exactly, but to create their own implementation that simply fit in the same form factor and used a USB interface instead of an RS232 serial interface (Wolowicz and Knoll). Their selected hardware and protocol were quite different from the EasyPort and most documents listed on their website were marked as proprietary. No further attempts to reverse engineer or replicate the EasyPort were discovered.

Most other literature found focused on simulating the control device instead of or in addition to simulating the processes. A well-documented design for a Virtual Plant Generator was found, but the paper didn’t provide any details on creating a hardware interface (Ngalamou). The Virtual Plant Generator was designed to use QT for simulation graphics, but it was mentioned that Flash, like EasyVeep uses, was a viable alternative. Another paper discussed extending PLC simulation software to interact with real world processes. Though the concept presented in the paper is exactly opposite of objective, some concepts could be considered useful. The solution used a parallel port but states clearly that the solution could be adapted to use serial or USB (ROMANIA). Some of the issues that could arise with such a system were documented as well.

The search continued for EasyPort like devices that could be modified to work with EasyVeep. More proprietary implementations of EasyPort like systems were found. Sealevel systems provides a wide range of Digital I/O USB solutions with up to 16 inputs and outputs (Sealevel). The units are industrial grade and use a proprietary SeaMAX protocol. The protocol used by the Sealevel units prevents them from being a viable solution to link the Real Time target to the simulated industrial systems, but the Sealevel website provides other valuable resources that can be applied to the propose project. Sealevel freely provides select chapters of its publication The Digital I/O Handbook which includes descriptions and design considerations for Digital I/O implementations (Sealevel).

## Existing Solutions

There are a number of existing simulation solutions for real time industrial control, but few come close to meeting the goal of this research project. Most available hardware and software solutions examined focus on training users to program and operate PLCs.

### Festo-Didatic EasyVeep

EasyVeep is advertised as a “graphical 2D process simulator with numerous attractive examples on PLC training.” EasyVeep is distributed freely and can be acquired easily. However, the software requires an external hardware interface, which is also produced by Festo-Didatic, called an EasyPort.

### Other Solutions (More on this later)

# Methodology

This section details the methods used and steps taken to complete the GRP. It is broken down into sections that coincide with the main work items that were experimented with during the research process. The items are presented in the chronological order in which they were researched.

## Using EasyVeep

As a software solution EasyVeep appears capable of meeting all the required functionality as discussed in Existing Solutions. In order to be used, a custom hardware device which replicates the behavior of Festo’s EasyPort is need. During in initial inspection of the manufacturer’s website, the documentation on the EasyPort was found to be inadequate in allowing complete replication of the EasyPort’s behavior. Eventually a document containing a protocol explanation was found after some broken links were fixed on Festo’s website. While waiting for the links to be repaired, a binary analysis of EasyVeep and its supporting binaries was performed.

### Reversing the EasyPort Protocol

The process of reversing the EasyPort protocol was a challenge. The extensive details on how the protocol was partially reversed are included in the appendix section Reversing EasyVeep. Primarily, the process consisted of static binary analysis using the freeware version of the Hex-Rays IDA interactive disassembler. Through examination of the files included with the EasyVeep installation, the library that facilitates serial communication with the EasyPort was located and analyzed both statically and dynamically.

To facilitated dynamic analysis, the com0com loopback driver was used to redirect, examine, and send information between various processes running on the local machine. This allowed for easy logging of the communication protocol as well as allowing fuzz testing of the library when attached to a debugger. Fuzzing was used when there was some ambiguity as to the content or context of the protocol. By attaching a small Python script to the other end of com0com loopback, simple EasyPort emulation was possible, and the data being sent back to the EasyPort library could be fuzzed until the desired debugging breakpoint was hit. A summary of the discovered protocol is shown in the table and figure below. This is an incomplete version of the entire protocol but includes all functionality necessary to meet the project objective. Commands are given in the format of a regular expression that would match a valid command. Each command is terminated with a carriage return.

Table 1 : EasyPort Protocol Listing

|  |  |
| --- | --- |
| Command Format | Description |
| setup0 | Request made by PC to initialize connected EasyPorts. |
| setup[1-4] | EasyPort response to setup0 command requesting a module number 1-4. |
| DV | Request by PC to get EasyPort version |
| V=\d.\d{2} | Response to version request containing the version number of the EasyPort. Must exceed (1.20). |
| MAW=[1-4].[0248]=[0-F]{4} | Modify the 16 bit output value of EasyPort module [1-4], channel log2([0248]) to be the value represented by hex number [0-F]{4}. |
| DEW[1-4].[0248] | Request the current input values from module [1-4], channel log2([0248]) |
| EW[1-4].[0248]=[0-F]{4} | Response to input value request. The current value of the inputs is represented by a four digit hex number. DAW command is sent in response requesting the current out values the EasyPort has. A response to DAW is not required. |
| DAW[1-4].[0248] | Request the current output values from module [1-4], channel log2([0248]) . A response is not required in this application. |

Figure : EasyPort Protocol Sequence Diagram

EasyPort

Computer

COM Setup

Module Identification

Update Output

Update Input

setup0

setup1

DV

V=1.21

DEW1.8

EW1.8=0000

MAW1.0=0001

DEW1.8

EW1.8=0000

DAW1.0

### Implementing an EasyPort

After determining the protocol that would be used, an attempt was made at implementing the EasyPort protocol on an Arduino Uno R3. The Arduino was selected for simplicity, and while it lacked the number of pins necessary to provide two full 16bit channels, it provided enough I/O to interact with a decent subset of the models provided by EasyVeep.

Problems were experienced when implementing the Arduino Uno EasyPort that were not easily surmountable. The main issues stemmed from the Uno’s lack of native support for Flow Control as detailed in the appendix section Arduino Flow Control Failures. The code, which only attempts to implement the startup cadence, is included in Arduino EasyPort Code along with a short description of the codes architecture.

With the Arduino eliminated as a valid platform, **com0com** was used to set up a loopback serial connection that supported flow control. A C# Windows Form Application was developed that provided a serial log window along with checkboxes to simulate changing inputs and a field to hold the current output value. Shortly after adding the protocol implementation to the form application end to end communication with EasyVeep was achieved. Implementation details for the desktop EasyPort emulator are shown in the appendix section Desktop EasyPort Emulator.

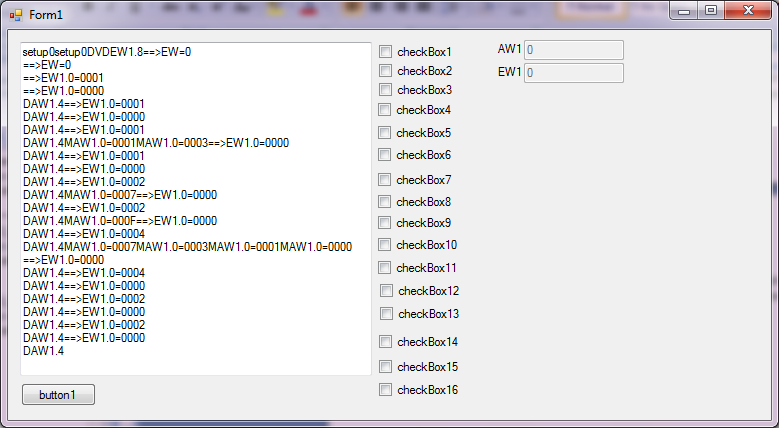


Figure : A screenshot of the desktop application implementing the EasyPort protocol.

Focus was once again shifted to creating a hardware implementation of the protocol. This time, the Texas Instruments Stellaris Launchpad was chosen. The Launchpad was selected primarily due to its immediate availability and low cost. In addition to providing more general purpose I/O pins and a more powerful processor, the Launchpad also included a debugging interface that the Arduino lacked. Initial implementation and testing began with the Launchpad performing only the startup cadence. Unlike the Arduino, the Launchpad had full support for hardware flow control and the cadence was easily implemented.

The implementation continued and the protocol commands for communicating the state of the actuator inputs was added. This allowed the Launchpad to interact with the EasyVeep models and begin receiving the simulated sensor values. With this accomplished, a simple breadboard circuit consisting of switches and LEDs was created to allow easy to visualize interaction. While the Launchpad was able to relay the state of the switches back to EasyVeep it was only receiving updated sensor values from EasyVeep immediately after the position of the switches changed, no when the sensor values in EasyVeep.

The issue was examined using the Launchpad’s debugger, but the debugging information and the EasyVeep protocol shared the same USB connection. Therefore, as debugger commands were issued and results received, the data was mixed and led to EasyVeep being unable to communicate with the Launchpad when the debugger was attached. Additional attempts to solve the issue using other means of debugging eventually led to the constant soft-locking of EasyVeep. Due to the closed source nature of the EasyVeep executable and the difficulty of debugging asynchronous serial communication using disassembled code the issue remained unresolved.

### Implementing a Custom EasyVeep

After the failures in interacting with EasyVeep Stellaris Launchpad, the decision was made to extract the models from the EasyVeep software and build a new application around them. This allowed for easier debugging and finer control of the environment. For details on use and implementation of existing EasyVeep processes in the custom implementation, reference appendix section EasyVeep Model Design and Interaction.

# Budget and Schedule

# Resources

**There are no sources in the current document.**

Ngalamou

http://paper.ijcsns.org/07\_book/201009/20100908.pdf

Romanian

http://www.aece.ro/abstractplus.php?year=2010&number=1&article=15

Izaguirre

http://www.indiana.edu/~ciec2011/ETD/ETD-351/ETD-351\_Izaguirre.pdf

# Appendices

## Reversing EasyVeep

In order to begin the reversing process, information regarding the format and structure of the EasyVeep executable needed to be obtained. The analysis began by simply launching EasyVeep. The user is greeted with a “Made with Macromedia” splash screen and visiting the Setup pane in EasyVeep confirms that the software relies heavily on what was previously known as Macromedia Flash Player (now an Adobe product). This was helpful information, and Flash, from previous experience, was known to be easy to decompile and reverse engineer in comparison to binaries produced with languages such as C and C++. However, pure Flash based programs cannot directly communicate with low level system devices such as serial ports; there are no documented functions in the official documentation for doing so. This indicates that the functionality of interest resided elsewhere. Analysis indicates that EasyVeep uses Flash only for running and displaying the simulated processes. Communication and protocol implementation is handled elsewhere.

The next step in the analysis was to examine the EasyVeep executable more closely. The assembly information was examined by viewing the executable properties. It was found to be an Authorware Runtime application. Authorware is primarily designed for developing learning management systems (LMS) in a visual manner (INSERT SOURCE). Given the description of the Authorware software, it did not appear that the Authorware application itself would have access to the serial port but further analysis was needed to eliminate the main executable as the targeted binary. To do so the Import Address Table (IAT) of the EasyVeep module was inspected.

The IAT serves as a table of virtual function pointers to various functions encapsulated in external libraries (<http://sandsprite.com/CodeStuff/Understanding_imports.html>). If the EasyVeep application were to perform the serial communication itself, it would likely be importing serial control functions from the Windows API. After dumping the IAT for EasyVeep.exe the 457 entries were checked and none appeared to have anything to do with serial communication. Additionally, the strings list was checked as well for anything related to serial communication. The strings table holds a list of all the ASCII strings used by a program and in this case, nothing related to the windows COM port file descriptor (\\COM%d) used when opening a serial port was found.

After the search for serial port related functions failed to turn up anything significant, the IAT was examined again for clues on where to look direct attention to next. Most imports were from the GDI32 library (Graphics Device Interface), USER32, and KERNEL32 libraries. The USER32 and GDI32 imports all were directed towards drawing and presenting the user interface while the KERNEL32 library functions were reading and traversing file directories and processes. The imports the proved to be of interest were those from the ole32 library. The ole32 library implements Object Linking and Embedding (OLE) which is primarily used for embedding and linking disparate data items, such as embedding a Flash movie into a Microsoft Word document. The OLE allows developers to create their own OLE Control Extensions (OCX) to extend functionality of existing user interfaces.

Inspection continued of the OLE functions. Using the freeware version of IDA interactive disassembler, breakpoints were set on all functions that loaded OLE objects. However, when launching the application through the debugger, exceptions were thrown and the application refused to load and hit the breakpoints. Instead of attempting to debug further, as bypassing anti-debugging measures is not within the extent of this project, a different approach was taken. EasyVeep was launched without a debugger attached and then once fully loaded IDA was used to attach the debugger. IDA provides a list of loaded modules and within that list a module named EasyPort.ocx was found. This appeared to be a promising target as it was named after the device that was being reverse engineered.

The EasyPort.ocx file was loaded into IDA for disassembly and debugging. Immediately upon checking the IAT, functions GetCommState, BuildCommDCBA, SetCommState, and PurgeComm appeared at the top of the list. A quick Google search showed that these were all part of the Windows Communication Functions API (<http://msdn.microsoft.com/en-us/library/windows/desktop/aa363194(v=vs.85).aspx)>.

The COM functions were break pointed and the debugger was attached to a running EasyVeep process. By stepping through the process it was found the EasyPort.ocx begins iterating over a range of integers, attempts to open a COM port of that integer, and then establish communications with an EasyPort device. The COM initialization code is shown below.



Figure 4 : EasyPort COM Port Discovery Code

This code first builds the file description string for the COM port and then builds the DCB control settings for the serial device. It then tries to open the COM using CreateFileA and the commands immediately following that function call attempt to detect and report errors. A breakpoint was set after the third sprint call to examine the string that was used setting the DCB for the serial device. It was found that for each COM port, the above routine gets called twice. The first time a baud rate of 19200 is used and the second time a baud rate of 115200 is used. If communication fails on both baud rates, the next COM port is tried until no more remain. If communication success, the process continues and all ports found to have a COM device are reported to EasyVeep.



Figure 5 : EasyPort DCB Parameters

To determine if the COM port is attached to an EasyPort, the EasyPort.ocx sends the ascii string “setup0\r” and waits for a response. If a response is not sent in a valid time frame the setup string is sent again. In return, the software expects a string in the format of “setup%1d\r” to be returned where the value represented by %1d is between 1 and 4. If this succeeds, the string “DV\r” is sent. The expected string in return is in the format of “V=%1d.%2d\r”. Once this exchange is complete, the validity of the EasyPort is confirmed.

Upon getting a list of valid COM ports, EasyVeep will attempt to open each in order to determine what variation of EasyPort they are. It does this by sending the message in the form of “DEW%d.8” where %d corresponds to the number received from the EasyPort during the setup process. It appears that the length of time and manner in which the EasyPort responds to these requests effects the model type returned. Further analysis has not been made, as it was found during implementation that simply responding as fast as possible would cause the emulated EasyPort to be returned as the correct type.

Further analysis was performed on the OCX object to determine what occurs after setup. It was recognized that the OCX was in the same format as a typical windows com object, so the virtual method table (vtable) was located in the .rdata section of the EasyPort.ocx disassembly (<http://blogs.msdn.com/b/oldnewthing/archive/2004/02/05/68017.aspx>). The vtable holds the name of the methods that can be called from a process using the OCX along with pointers to the associated functions. This allows plain text, expressive function names to be used in deciphering the cryptic disassembly.

The vtable contained many more methods than were necessary for meeting the project objective, so only those that were of concern were examined. The first function examined was “SetOutput”. The trace for set output eventually led to the sending of a command “MAW%d.%1d=%04X”, where MAW seemed to stand for Modify Output Word. The %d.%1d was similar to the “DEW%d.8” command found earlier, where the first digit corresponded to the module number, thus the format is Module#.Channel# = HexValue.

Determining how the input values were obtained was not as straight forward. The “GetInputWord” function did not directly lead to a certain serial command, but instead a small move instruction. The move instruction references an address based around the ecx register, which in C++ typically holds the address of the “this” variable. Therefore, at [eax\*4+0x24], the value of the most recent input is stored, where eax is assumed to be the module number.

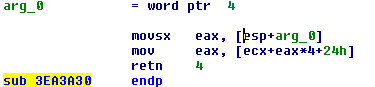


Figure 6 : EasyPort.OCX Snippet Returning EasyPort Inputs

Examining the code the is run when a serial data stream is received eventually led to a case where an almost identical, but reversed move was found. The same offset is being used to load the effective address of the field into edx and then the value in eax is moved into the address pointed to by edx. This code was found by tracing the execution for the case when the serial input is of the form “EW%1d.%1d=%4X\r”. Here EW denotes the value of the input word is the included four digit hex number.

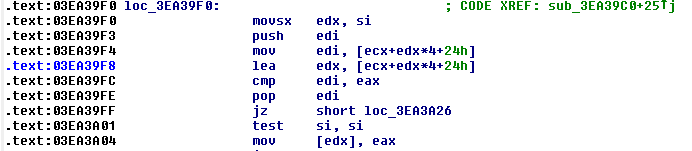


Figure 7: EasyPort.OCX Snippet Storing Input Values

With this knowledge, all the commands necessary to establish connection and interact with the base functions of EasyVeep are known. The cadance to initialize setup, along with the commands used to modify outputs, and query and update inputs are well documented. None of the other commands that a production EasyPort can interpret are necessary to perform the project objective.

## Arduino EasyPort Code

This code was developed during the initial testing of the reversed serial protocol. It attempts to complete the startup cadence but provides no further functionality. The code was found to respond correctly when manually issuing commands to the Arduino but failed to work with EasyVeep due to issues discussed in Arduino Flow Control Failures. The serial communication is event driven, with the SerialEvent function automatically firing whenever new serial data is received.

**String** commandString = "";

**void** setup() {

//EasyPort runs at either 19200 or 115200

//19200 gets tried first so just start with that

Serial.begin(19200);

//Commands shouldn't exceed 16 chars

commandString.reserve(16);

}

**void** loop() {

//We're just going to use SerialEvent

}

//

// This will process the current command string and send

// back the correct response.

////////////////////////////////////////////////////////

**void** ProcessCommand(){

**if** ( commandString.equals("setup0")){

Serial.print("setup1\r");

//Pretend to be module #1

} **else** **if** ( commandString.equals("DV")){

Serial.print("V=1.21\r");

//Version has to be greater than 1.2

} **else** {

**if** ( commandString.equals("DEW1.8") ){

Serial.print("EW1.0=0000\r");

}

}

commandString = "";

}

//

// This fires when new data shows up on the serial buffer

/////////////////////////////////////////////////////////

**void** serialEvent(){

**char** inchar;

**while**(Serial.available()){

//Read available bytes until we get a new command

// all commands end in a carriage return '\r'

inchar = (**char**)Serial.read();

**if** ( inchar != '\r'){

commandString += inchar;

}**else**{

ProcessCommand();

}

}

}

## Arduino Flow Control Failures

The startup cadence was implemented on the Arduino and it was tested using the serial monitor included with the Arduino IDE. When typing commands into the serial monitor that coincided with the commands the Arduino would be receiving from EasyVeep the behavior was as expected. However, when EasyVeep attempted to talk to the Arduino the connection was not established.

The Arduino was behaving as if it was not receiving valid serial data from EasyVeep. At the time, access to a logic analyzer was not available, and after multiple re-writes of the serial processing code, the code was modified to log all bits received to the Arduino’s EEPROM and report them over the serial line the next time the Arduino reset. The EEPROM writes were required because the Arduino resets every time a new device connects to it, so it was the only way to keep the data persistent.

With the new code, the Arduino was connected to EasyVeep and sent two “setup0\r” commands. EasyVeep was then closed and the Arduino serial monitor was opened. The EEPROM was dumped and the values were found to all be hexadecimal FF. This showed that the Arduino was receiving some sort of serial data but all the received bits were ones.

After many hours of debugging, it was found that the Arduino does not support the Request To Send (RTS) flow control operation. EasyVeep was attempting to use proper serial flow control and as a result, it appears that the Arduino was receiving corrupted data or control bits instead of the expected serial protocol. Since the Arduino uses a FTDI UART to USB IC and the IC only provides a physical interface to the TX and RX pins there was not trivial way to enable flow control (<http://arduino.cc/forum/index.php/topic,37368.0.html>).

The following log segment was produced using a trail version of Agg Software Advanced Serial Port Monitor. The areas of interest are bolded. The use of the flow control signals Request To Send (RTS) and Data Terminal Ready (DTR) are shown as well as the attempt to initialize a flow control handshake. The handshake fails and port is purged three times before the process aborts, sending zero data characters. This occurs for both baud rates when attempting to communicate with the Arduino and explains why the EasyVeep software was unable to communicate with the Arduino Uno.

<20130406092605.647 SYS>

COM is open

<20130406092605.647 SYS>

Set timeouts: ReadInterval=1000, ReadTotalTimeoutMultiplier=1000, ReadTotalTimeoutConstant=1000, WriteTotalTimeoutMultiplier=1000, WriteTotalTimeoutConstant=1000

**<20130406092605.657 SYS>**

**Baud rate 19200**

**<20130406092605.657 SYS>**

**RTS on**

**<20130406092605.667 SYS>**

**DTR off**

<20130406092605.667 SYS>

Data bits=8, Stop bits=1, Parity=None

**<20130406092605.667 SYS>**

**Set chars: Eof=0x00, Error=0x00, Break=0x00, Event=0x00, Xon=0x00, Xoff=0x00**

**<20130406092605.667 SYS>**

**Handflow: ControlHandShake=(), FlowReplace=(TRANSMIT\_TOGGLE, RTS\_CONTROL), XonLimit=0, XoffLimit=0**

<20130406092605.667 SYS>

Purge the serial port: RXABORT, RXCLEAR, TXABORT, TXCLEAR

<20130406092605.667 SYS>

Purge the serial port: RXABORT, RXCLEAR, TXABORT, TXCLEAR

<20130406092605.667 SYS>

Purge the serial port: RXABORT, RXCLEAR, TXABORT, TXCLEAR

**<20130406092605.676 TX>**

**[len=0]**

<20130406092606.133 SYS>

COM is closed

<20130406092606.133 SYS>

COM is open

<20130406092606.133 SYS>

Set timeouts: ReadInterval=1000, ReadTotalTimeoutMultiplier=1000, ReadTotalTimeoutConstant=1000, WriteTotalTimeoutMultiplier=1000, WriteTotalTimeoutConstant=1000

**<20130406092606.153 SYS>**

**Baud rate 115200**

**<20130406092606.153 SYS>**

**RTS on**

**<20130406092606.153 SYS>**

**DTR off**

<20130406092606.162 SYS>

Data bits=8, Stop bits=1, Parity=None

**<20130406092606.162 SYS>**

**Set chars: Eof=0x00, Error=0x00, Break=0x00, Event=0x00, Xon=0x00, Xoff=0x00**

**<20130406092606.162 SYS>**

**Handflow: ControlHandShake=(), FlowReplace=(TRANSMIT\_TOGGLE, RTS\_CONTROL), XonLimit=0, XoffLimit=0**

<20130406092606.162 SYS>

Purge the serial port: RXABORT, RXCLEAR, TXABORT, TXCLEAR

<20130406092606.162 SYS>

Purge the serial port: RXABORT, RXCLEAR, TXABORT, TXCLEAR

<20130406092606.162 SYS>

Purge the serial port: RXABORT, RXCLEAR, TXABORT, TXCLEAR

**<20130406092606.162 TX>**

**[len=0]**

<20130406092606.620 SYS>

COM is closed

## Desktop EasyPort Emulator

The Desktop EasyPort Emulator’s primary purpose was to allow the protocol to be debugged and further understood in an environment which was easy to debug. By utilizing the com0com interface the desktop application appeared to be communicating through a COM port to EasyVeep, allowing for connection and full manipulation of the EasyVeep models. The emulator included a log window showing all transmitted and received serial commands along with checkboxes to allow the user to toggle the actuator values. The form code is include below for reference. The application uses regular expressions to matched expected commands.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using System.Text.RegularExpressions;

namespace EasyPort

{

public partial class val : Form

{

private string currentCommand = "";

private int Ch1Out = 0;

private int Ch1In = 0;

private Timer timer1 = new Timer();

public val()

{

InitializeComponent();

serialPort1.Open();

}

private void serialPort1\_DataReceived(object sender, System.IO.Ports.SerialDataReceivedEventArgs e)

{

while (serialPort1.BytesToRead > 0)

{

byte[] indata = new byte[serialPort1.BytesToRead];

serialPort1.Read(indata,0,serialPort1.BytesToRead);

var comString = Encoding.UTF8.GetString(indata);

AppendSerialData(comString);

if (comString.IndexOf('\r') > -1 && comString.Length > 1)

{

//We gots a new command

currentCommand += comString.Substring(0,comString.IndexOf('\r'));

currentCommand = ProcessCommand(currentCommand,comString.Substring(comString.IndexOf('\r')+1));

}

}

}

private string ProcessCommand(string completedCommand, string residualCommand)

{

//Module should always be 1...

//Setup Command

if (completedCommand.Contains("setup0"))

{

serialPort1.Write("setup1\r");

}

//Version Info Command

else if (completedCommand.Contains("DV"))

{

serialPort1.Write("V=1.21\r");

}

else if (completedCommand.Contains("MT1"))

{

var ms = completedCommand.Substring(completedCommand.IndexOf("=")+1);

var msInts = int.Parse(ms);

initMyTimer(msInts);

}

//Digital Ouput Command

else if (completedCommand.Contains("MA"))

{

// MA Module.Word.Bit = Val (0 or 1)

// MAB Module.Word.Byte = Val (0-F)

// MAW Module.Word = Val (00-FF)

var maREG = new Regex(@"MA([A-Z]?)(\d)\.(\d)\.?(\[0-9A-F])?=([0-9A-F]+)");

var matches = maREG.Match(completedCommand);

if (matches != null && matches.Groups.Count > 0)

{

var action = matches.Groups[1].Value;

var module = matches.Groups[2].Value;

var word = matches.Groups[3].Value;

var address = matches.Groups[4].Value;

var value = matches.Groups[5].Value;

switch (action)

{

case "":

break;

case "B":

break;

case "W":

if (word == "0")

Ch1In = Int16.Parse(value, System.Globalization.NumberStyles.HexNumber);

break;

}

updateAW1();

}

else

{

//LOG BAD COMMAND

}

}

//Digital Input/Output Query Commands

else if (completedCommand.Contains("DE"))

{

//// Display Input Values ////

//DE Module.Word.Bit

//DEB Module.Word.Byte

//DEW Module.Word

var deREG = new Regex(@"DE([WB]?)(\d)\.(\d)\.?([0-9A-F])?");

var matches = deREG.Match(completedCommand);

if (matches != null && matches.Groups.Count > 0)

{

var datsize = matches.Groups[1].Value;

var module = matches.Groups[2].Value;

var word = matches.Groups[3].Value;

var bitbyte = matches.Groups[4].Value;

var valOut = "00";

switch (datsize)

{

case "":

//displaying a bit

break;

case "B":

//displaying a byte

if (word == "0")

valOut = (bitbyte == "0" ? String.Format("{0}:X", Ch1Out & 0x0F) : String.Format("{0}:X", (Ch1Out >> 8) & 0x0F));

break;

case "W":

//displaying the whole word

valOut = Ch1In.ToString();

break;

}

var transmission = String.Format("E{0}={1}\r", datsize, valOut);

sendDataBack(transmission);

}

else

{

//LOG BAD COMMAND

}

}

else if (completedCommand.Contains("DA"))

{

//// Display Output Values ////

//DA Module.Word.Bit

//DAB Module.Word.Byte

//DAW Module.Word

}

return "";

}

private void sendDataBack(string response)

{

if (InvokeRequired)

{

this.Invoke(new Action<string>(sendDataBack), new object[] { response });

}

AppendSerialData("==>" + response + "\r\n");

serialPort1.Write(response);

}

private void button1\_Click(object sender, EventArgs e)

{

}

private void textBox1\_TextChanged(object sender, EventArgs e)

{

}

private void updateAW1()

{

if (InvokeRequired)

{

this.Invoke(new Action(updateAW1));

return;

}

valAW1.Text = this.Ch1In.ToString();

}

private void updateEW1()

{

if (InvokeRequired)

{

this.Invoke(new Action(updateEW1));

return;

}

valEW1.Text = this.Ch1Out.ToString();

}

private void AppendSerialData(string b)

{

if (InvokeRequired)

{

this.Invoke(new Action<string>(AppendSerialData), new object[] { b });

return;

}

b.Replace("\r", Environment.NewLine);

textBox1.Text += b;

}

private void chkA1\_CheckedChanged(object sender, EventArgs e)

{

var newOut = CaluculateOutput();

Ch1Out = newOut;

updateEW1();

sendDataBack(String.Format("EW1.0={0:X4}\r", newOut));

}

private void initMyTimer(int ms)

{

if (InvokeRequired)

{

Invoke(new Action<int>(initMyTimer), new object[] { ms });

return;

}

timer1.Tick += new EventHandler(timer1\_Tick);

timer1.Interval = ms;

timer1.Start();

}

private void timer1\_Tick(object sender, EventArgs e)

{

timer1.Enabled = true;

}

private int CaluculateOutput()

{

return (checkBox1.Checked ? 1 : 0) |

(checkBox2.Checked ? 1 << 1 : 0) |

(checkBox3.Checked ? 1 << 2 : 0) |

(checkBox4.Checked ? 1 << 3 : 0) |

(checkBox5.Checked ? 1 << 4 : 0) |

(checkBox6.Checked ? 1 << 5 : 0) |

(checkBox7.Checked ? 1 << 6 : 0) |

(checkBox8.Checked ? 1 << 7 : 0) |

(checkBox9.Checked ? 1 << 8 : 0) |

(checkBox10.Checked ? 1 << 9 : 0) |

(checkBox11.Checked ? 1 << 10 : 0) |

(checkBox12.Checked ? 1 << 11 : 0) |

(checkBox13.Checked ? 1 << 12 : 0) |

(checkBox14.Checked ? 1 << 13 : 0) |

(checkBox15.Checked ? 1 << 14 : 0) |

(checkBox16.Checked ? 1 << 15 : 0);

}

}

}

## EasyVeep Model Design and Interaction

It was discovered that each of the EasyVeep process simulations were completely encapsulated in their own, independent swf files, which are compressed Adobe Flash movies. Adobe provides and ActiveX control that allows developers to easily load and embed Flash movies within an application. The swf files were still a black box with no fields publicly exposed in plain text. To allow interfacing with the process simulations, a trial version of the Sothink SWF Decompiler was downloaded and used to analyze the Actionscript code. This allowed for important variable names to be extracted so that they could be read and modified by the custom EasyVeep implementation. Some important variable names are as follows:

Table : Important SWF Variable Names

|  |  |
| --- | --- |
| **Variable Name** | **Purpose** |
| EprgName | English Program Name |
| EprgLeirasX | English Program Description ( X = 0..10) |
| EDigSensX | English Sensor Description ( X = 1..16) |
| EDigActX | English Actuator Description (X = 1..16) |
| DAX Digital | Actuator Value ( X =1..16) |
| DSX Digital | Sensor Value (X=1..16) |
| ASX Analog | Sensor Value ( X=1..16) |

These variables can be accessed in the using the GetVariable and SetVariable functions of the Flash ActiveX Control. For example, finding all the sensors associated with a process model can be performed as follows after the model is loaded.

**do**{

SensorDescription = axFlash.GetVariable(String.Format("EDigSens{0}", SensorDescriptionIndex));

**if** ( SensorDescription != "" )

movieInfo.Sensors[SensorDescriptionIndex-1] =

**new** **DigitalSensor**( SensorDescription, SensorDescriptionIndex);

SensorDescriptionIndex++;

} **while** (SensorDescription != "" && SensorDescriptionIndex <= 16);

Further analysis of all the models was performed using flasm, which unlike the Sothink decompiler, is free and unlimited in use. Flasm was combined with various command line utilities in order to produce dumps for each of the Flash movies. The following command was used to extract the number of sensors and actuators as well as their functions from each of the swf files.

**for** f in \*.swf; **do** printf "\n \_ $f \_ \n\n|Field|Value|\n|---|---|\n"; flasm -d $f | awk "/push \*'(\w+SensNum|\w+ActNum|EDig|EAnalog|fileName|EprgName)/" | sed "s/|//g" | sed "s/ \*push \*'\([^,]\*\)', '\([^']\*\)'/|\1|\2|/g" | awk '!/\|\|/' ; **done**;

It was found that the values of the EDigSens and EDigAct were constant (only one push every occurred to the variable) and that they could only hold the name of the sensor or actuator. The actual values (on or off, tripped or untripped) were stored elsewhere. Grepping through the output of flasm turned up variables matching the regular expression **D[A|S]\d+**. Here, DA1 - DA16 represented the digital values assigned to actuators 1 - 16. Similarly, DS1 - DS16 represented the values held by the digital sensors. Assuming that analog values would be handled the same way, the output was grepped for **A[A|S]\d**. No matches were found. After quickly searching through the results, it was shown that not a single process model leveraged the analog sensor abilities despite the inclusion of fields to specify them. This meant that **all the processes were completely digital** and that the MyEasyVeep application did not have to support analog values.

### SWF Dump Results

While there are 30 processes included with EasyVeep only some are relevant to the objective. The acquired dumps from those processes deemed to be of prime interests are shown below. The dumps for all 30 were produced in the same manner but are not included for brevity.

Table : Process Information for Boat Lock Process

|  |  |
| --- | --- |
| **Field** | **Value** |
| fileName | 13.swf |
| DigSensNum | 11 |
| DigActNum | 6 |
| AnalogSensNum | 0 |
| AnalogActNum | 0 |
| EprgName | Lock chamber |
| EDigSens1 | Gate on the left closed |
| EDigSens2 | Gate on the right closed |
| EDigSens3 | Water level low |
| EDigSens4 | Water level high |
| EDigSens5 | Ship from left |
| EDigSens6 | Ship at the left gate |
| EDigSens7 | Ship in the lock-gate |
| EDigSens8 | Gate on the left is open |
| EDigSens9 | Gate on the right is open |
| EDigSens10 | Ship at the right gate |
| EDigSens11 | Ship from right |
| EDigAct1 | Open the gate on left side |
| EDigAct2 | Open the gate on right side |
| EDigAct3 | Increase the water level |
| EDigAct4 | Decrease the water level |
| EDigAct5 | Free sign on left side |
| EDigAct6 | Free sign on right side |

Table : Process Information for Parking Lot Barrier Process

|  |  |
| --- | --- |
| **Field** | **Value** |
| fileName | 14.swf |
| DigSensNum | 6 |
| DigActNum | 6 |
| AnalogSensNum | 0 |
| AnalogActNum | 0 |
| EprgName | Parking Lot |
| EDigSens1 | IN barrier is down |
| EDigSens2 | IN barrier is up |
| EDigSens3 | OUT barrier is down |
| EDigSens4 | OUT barrier is up |
| EDigSens5 | Car is at the barrier IN |
| EDigSens6 | Car is at the barrier OUT |
| EDigAct1 | IN barrier up |
| EDigAct2 | OUT barrier up |
| EDigAct3 | Red signal IN |
| EDigAct4 | Green signal IN |
| EDigAct5 | Red signal OUT |
| EDigAct6 | Green signal OUT |

Table : Process Information for Hot Water Tank Process

|  |  |
| --- | --- |
| **Field** | **Value** |
| fileName | 1.swf |
| DigSensNum | 6 |
| DigActNum | 4 |
| AnalogSensNum | 0 |
| AnalogActNum | 0 |
| EprgName | Hot water tank |
| EDigSens1 | Min. water level |
| EDigSens2 | Lower water level |
| EDigSens3 | Upper water level |
| EDigSens4 | Max. water level |
| EDigSens5 | Min. temperature |
| EDigSens6 | Max. temperature |
| EDigAct1 | Inlet valve (fast) |
| EDigAct2 | Inlet valve (slow) |
| EDigAct3 | Outlet valve |
| EDigAct4 | Heating |

Table : Process Information for Eleavtor Process

|  |  |
| --- | --- |
| **Field** | **Value** |
| fileName | 3.swf |
| DigSensNum | 12 |
| DigActNum | 5 |
| AnalogSensNum | 0 |
| AnalogActNum | 0 |
| EprgName | Elevator |
| EDigSens1 | Call-button on the 0. floor |
| EDigSens2 | Call-button on the 1. floor |
| EDigSens3 | Call-button on the 2. floor |
| EDigSens4 | Button for 0. floor (in the elevator) |
| EDigSens5 | Button for 1. floor (in the elevator) |
| EDigSens6 | Button for 2. floor (in the elevator) |
| EDigSens7 | Elevator on 0. floor |
| EDigSens8 | Elevator on 1. floor |
| EDigSens9 | Elevator on 2. floor |
| EDigSens10 | Door is closed on 0. floor |
| EDigSens11 | Door is closed on 1. floor |
| EDigSens12 | Door is closed on 2. floor |
| EDigAct1 | Start the motor upwards |
| EDigAct2 | Start the motor downwards |
| EDigAct3 | Open the door on 0. floor |
| EDigAct4 | Open the door on 1. floor |
| EDigAct5 | Open the door on 2. floor |

Table : Process Information for Seven Segment Display Process

|  |  |
| --- | --- |
| **Field** | **Value** |
| fileName | 5.swf |
| DigSensNum | 12 |
| DigActNum | 8 |
| AnalogSensNum | 0 |
| AnalogActNum | 0 |
| EprgName | 7-segment display |
| EDigSens1 | No. 1 to display |
| EDigSens2 | No. 2 to display |
| EDigSens3 | No. 3 to display |
| EDigSens4 | No. 4 to display |
| EDigSens5 | No. 5 to display |
| EDigSens6 | No. 6 to display |
| EDigSens7 | No. 7 to display |
| EDigSens8 | No. 8 to display |
| EDigSens9 | No. 9 to display |
| EDigSens10 | No. 10 to display |
| EDigSens11 | No. 11 to display |
| EDigSens12 | No. 12 to display |
| EDigAct1 | 1. segment |
| EDigAct2 | 2. segment |
| EDigAct3 | 3. segment |
| EDigAct4 | 4. segment |
| EDigAct5 | 5. segment |
| EDigAct6 | 6. segment |
| EDigAct7 | 7. segment |
| EDigAct8 | 2.+3. segment (for number 1) |

Table : Process Information for Wine Bottling Process

|  |  |
| --- | --- |
| **Field** | **Value** |
| fileName | 6.swf |
| DigSensNum | 7 |
| DigActNum | 8 |
| AnalogSensNum | 0 |
| AnalogActNum | 0 |
| EprgName | Bottling wine |
| EDigSens1 | Bottle on the station: filling |
| EDigSens2 | Bottle on the station: cork |
| EDigSens3 | Bottle on the station: neck label |
| EDigSens4 | Bottle on the station: label |
| EDigSens5 | Sensor for dark bottle |
| EDigSens6 | Sensor for white bottle |
| EDigSens7 | Station is ready |
| EDigAct1 | Fill red wine |
| EDigAct2 | Fill white wine |
| EDigAct3 | Corking |
| EDigAct4 | Neck labeling red wine |
| EDigAct5 | Neck labeling white wine |
| EDigAct6 | Labeling red wine |
| EDigAct7 | Labeling white wine |
| EDigAct8 | Production line (motor) |

Table : Process Information for Ball Sorting System

|  |  |
| --- | --- |
| **Field** | **Value** |
| fileName | 7.swf |
| DigSensNum | 6 |
| DigActNum | 3 |
| AnalogSensNum | 0 |
| AnalogActNum | 0 |
| EprgName | Ball sorting system |
| EDigSens1 | Eject piston in position - |
| EDigSens2 | Eject piston in position + |
| EDigSens3 | Capacitive sensor |
| EDigSens4 | Optical sensor |
| EDigSens5 | Inductive sensor |
| EDigSens6 | Ball in gravity chute |
| EDigAct1 | Eject piston valve |
| EDigAct2 | Gate for 1. storage |
| EDigAct3 | Gate for 2. storage |

Table : Process Information for Train Crossing Process

|  |  |
| --- | --- |
| **Field** | **Value** |
| fileName | 8.swf |
| DigSensNum | 8 |
| DigActNum | 3 |
| AnalogSensNum | 0 |
| AnalogActNum | 0 |
| EprgName | Level crossing with barrier |
| EDigSens1 | Train is in the crossing from right |
| EDigSens2 | Train is in the crossing from left |
| EDigSens3 | Train coming from right |
| EDigSens4 | Train coming from left |
| EDigSens5 | Barrier is up (right side) |
| EDigSens6 | Barrier is down (right side) |
| EDigSens7 | Barrier is up (left side) |
| EDigSens8 | Barrier is down (left side) |
| EDigAct1 | Let barriers down |
| EDigAct2 | Switch the white signal (clear) |
| EDigAct3 | Switch the red signal |