

ExpressLink Design Documentation

|  |  |
| --- | --- |
|  |  |
| **Authors:** |  |
| **Project:** | ExpressLink |
| **Date Saved:** |  |

|  |  |
| --- | --- |
| **Reviewed by:** |  |
| **Approved by:** |  |

CONFIDENTIAL

The information contained herein is confidential and shall not be revealed to any third party without the expressed written consent of . The document and all copies shall be returned to Icron promptly upon request by Icron.

Table of Contents

[List of Tables and Figures ii](#_Toc245807325)

[1. Introduction 1](#_Toc245807326)

[1.1. Purpose 1](#_Toc245807327)

[1.2. Scope 1](#_Toc245807328)

[1.3. Terminology References 1](#_Toc245807329)

[1.4. Document Reference 1](#_Toc245807330)

[2. ExpressLink Design Overview 1](#_Toc245807331)

[2.1. Background Information 1](#_Toc245807332)

[2.2. System Evolution Description 2](#_Toc245807333)

[2.3. Current Process 2](#_Toc245807334)

[2.4. User Characteristics 2](#_Toc245807335)

[2.4.1. User Problem Statement 2](#_Toc245807336)

[2.4.2. User Objective 2](#_Toc245807337)

[2.5. Proposed Process 3](#_Toc245807338)

[2.6. Constraints 3](#_Toc245807339)

[2.7. Design Trade-offs 3](#_Toc245807340)

[3. Design Architecture 4](#_Toc245807341)

[4. External Module Interface 6](#_Toc245807342)

[4.1. Regular User Interface 6](#_Toc245807343)

[4.2. Developer Debugging User Interface 6](#_Toc245807344)

[5. Detailed Design 7](#_Toc245807345)

[5.1. ExpressLinkMain.cs 7](#_Toc245807346)

[5.1.1. Opening and Closing Serial COM Port 7](#_Toc245807347)

[5.1.2. Thread 8](#_Toc245807348)

[5.1.3. Generate Debug Log 8](#_Toc245807349)

[5.1.4. Check Firmware Version 9](#_Toc245807350)

[5.1.5. Serial Communication Port 9](#_Toc245807351)

[5.1.6. Information Logging 11](#_Toc245807352)

[5.1.7. Programming the Device 11](#_Toc245807353)

[5.2. StewieFileSender.cs 12](#_Toc245807354)

[5.3. StewieFile.cs 13](#_Toc245807355)

[6. Test Plan 13](#_Toc245807356)

# List of Tables and Figures

[Figure 1 - ExpressLink Project Structure 4](#_Toc245807357)

[Figure 2 - Embedded Resources Build Action 5](#_Toc245807358)

[Figure 3 - ExpressLink Normal User Interface 6](#_Toc245807359)

[Figure 4 - ExpressLink Debug Mode Interface 7](#_Toc245807360)

# Introduction

The ExpressLink program is used for remote programming and debug of LionsGate IC based products including Robson and the LionsGate Development Kit. It runs on a PC and offers a seamless interface for the customer.

## Purpose

The purpose of this document is to introduce the reader to ExpressLink aiding the reader in the future maintenance, use and addition of features of ExpressLink.

## Scope

This document will cover the design and implementation of ExpressLink, including design decisions, implementation details, as well as future maintenance and enhancement. It also covers the overall structure of the software application along with detailed explanation on a number of selected functions.

## Terminology References

For C# terminology and references please refer to the following link

<http://msdn.microsoft.com/en-us/library/618ayhy6%28VS.71%29.aspx>

or search for “C# Programmer’s Reference”.

## Document Reference

[R-1] ExpressLink Marketing Requirement.docx, Erin Martin-Serrano, August 2009

[R-2] Tigger Design Document, Alexandra Burgin, August 2009

# ExpressLink Design Overview

## Background Information

This document covers the design and implementation of ExpressLink, which is a C# program using Microsoft .NET Framework, that is used for remote programming and debug of LionsGate IC based products including Robson and the LionsGate Development Kit.

ExpressLink is a standalone executable without any associated helper files. As a result, it can be send to customers by email directly and the only requirement on the customer end is a PC running Windows operating system with .NET Framework 3.5 or higher installed.

The firmware file is embedded inside the executable; thus, for every update to the firmware, a new ExpressLink must be built with the updated firmware file and delivered to the customer.

ExpressLink is capable of logging all the messages coming from the host computer and the device; essentially, all serial port communications. It can generate a debug log with information about the host PC such as OS, Motherboard, BIOS and System information. This saves QA and Engineering time on debugging systems as well as save time and shipping costs when units have to be sent back to Icron for programming.

Multiple copies of ExpressLInk can run simultaneously without interfering each other.

## System Evolution Description

As mentioned before, ExpressLink has the firmware files built into the executable. Every time a new version of the firmware is released, the ICR files under the Resources folder must be updated. The “version” file generated by the LionsGate build script must also be copied over since it contains the software version tag which ExpressLink uses to check against the tag obtained from the device.

Future upgrade to ExpressLink could include a better way of checking firmware version instead of relying on a file that contains the version tag since this file could easily be left out in the update process. Also, a Mac version of ExpressLink is highly desired thus porting it to Mac is another possible task that can be taken up on in the future.

## Current Process

Currently there is no solution to the problem presented. The internal development tool Leo and Tigger are both complicate and display too much information that is only useful to the developers. Leo and Tigger do have the capability of downloading firmware image to the device but, at the same time, they are too powerful and are filled with features that may not be useful to the customer; thus, could lead to more confusion.

## User Characteristics

## User Problem Statement

The problem ExpressLink trying to solve is how to provide the user with an easy to understand UI, simple and convenient way to program the devices. It also addresses the issue that, when and if, a device crashes during or after the programming process, how to obtain the crash report as well as a complete and accurate system profile with the least amount of user interaction.

## User Objective

The design of ExpressLink is aimed at achieving the following goals:

* The customer/end user receives an application to update programming and no Internet connection is required
* If a new firmware version is released, the user receives a new application
* No driver is required to run the application
* Ability to have multiple instances of the application open at the same time
* Able to minimize the application
* Able to run on the Windows platform
* Communication is through a serial COM port
  + For Robson products, the current TTL-232R-3V3 USB to serial chip will be used.
  + For the Development Kit, the FT232 USB to serial chip is on board and thus any mini USB cable is ok to use.

## Proposed Process

The proposed process is a C# program utilizing Microsoft .NET Framework to provide a clean and easy to use UI to the user.

## Constraints

The first and foremost constraint of ExpressLink is that it is written in C# and requires Microsoft .NET Framework version 3.5 thus is only limited to Windows system and portability of the code base to Linux or Mac is rather difficult. Third party tools such as Mono can be used to help with the migration of existing .NET projects to other platforms such as Linux or Mac but the current Mono (2.4.2) only provides support for .NET Framework 2.0.

ExpressLink only supports the Stewie transfer protocol since currently the flash\_writer is not interrupt driven thus very slow if it is used in binary transfer. This requires an update to the flash\_writer before ExpressLink can be changed to use the binary protocol which is supposed to be much faster.

Another problem ExpressLink currently experiences is that when a serial port connector is pulled from the plug after a live connection is already established, upon reconnect ExpressLink is able to recover the lost serial port but it will throw an exception when exiting the program.

This problem is a bug of the underlying .NET Framework and a workaround has been implemented and tested in ExpressLink. Microsoft said this bug would be fixed in the next major release of .NET Framework, which is version 4.0 with a scheduled release date of March 22, 2010. Once .NET Framework 4.0 is released, this problem should be handled properly and, if proven to work, remove the workaround.

## Design Trade-offs

ExpressLink is aimed to be fast, error-proof, and easy to use. The project was analyzed by a code analysis tool, Microsoft FxCop, which was designed specifically for .NET based applications. The suggestions and problems discovered by FxCop were addressed where applicable. Some other issues raised were not addressed such as globalization, text handling when running on right-to-left systems.

# Design Architecture

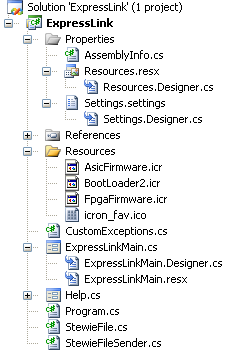


Figure - ExpressLink Project Structure

ExpressLink is composed of several different classes each handles different aspects of the program. The entire solution is under the namespace ExpressLink.

**ExpressLinkMain.cs**

This is the main form of ExpressLink. It is the user interface which the user uses to perform tasks and control other aspects of the software application.

**StewieFile.cs**

This class stores a Stewie file and provides functions that check if the file is in the correct format, retrieve records, header and footer of the file.

**StewieFileSender.cs**

This class manages the sending of a Stewie file to the target device, including synchronization with the target device before sending the data.

The firmware images are embedded into the final executable. They are stored under the Resources folder. These files are generated by the build script from LionsGate and should be updated to the most current stable version of LionsGate before building a new ExpressLink release.

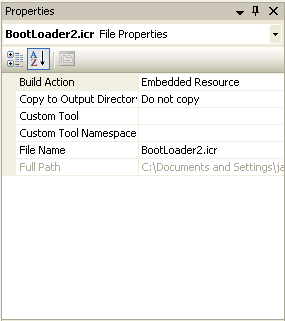


Figure - Embedded Resources Build Action

BootLoader2.icr:

This is the flash writer. It is send to the device to be programmed first and then the device uses it to receive the next firmware image and program the device itself.

AsicFirmware.icr:

This file is the actual firmware from the LionsGate repository. It must be built on a stable LionsGate release.

FpgaFirmware.icr:

This file is used as a debug tool since the current development board requires the FPGA firmware instead of the ASIC firmware. If debugging is done on the development board, this file must be send to the device instead of the AsicFirmware.icr.

NOTE: The Build Action of FpgaFirmware.icr should be set to “Embedded Resource” if FpgaFirmware.icr must be downloaded to the development board or else the program will fail. If the size of the executable becomes a concern in the future, set the Build Action to NONE for the final release that is going to the customer because ExpressLink is aimed at downloading firmware to ASIC devices instead of FPGA so the FpgaFirmware.icr is not needed when build an ExpressLink release for the customers.

# External Module Interface

## Regular User Interface

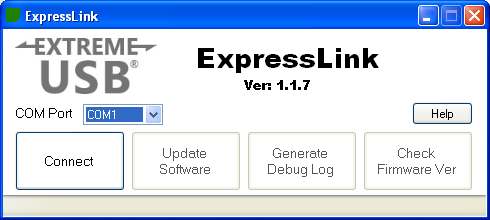


Figure - ExpressLink Normal User Interface

The design of ExpressLink is intended to be a standalone executable software application so there are no external APIs or library dependencies. The difference between the normal mode and debug mode interface is done using the #if DEBUG compiler directive.

## Developer Debugging User Interface

As a result, the build release seen by the user is always the one without the extra window that shows the message and communication between the device and the software application while, at the same time, allows the developer to see everything without the need of making changes in designer before each official release, which can be easily forgotten hence releasing a version that is intended for internal use to the customer.

/\*

\* Removes the need of changing the window size and the text box visibility

\* between debugging build and release build.

\*/

#if DEBUG

this.textBoxLogWindow.Visible = true;

this.Height = 350;

this.Text = "ExpressLink DEBUG VERSION";

#endif

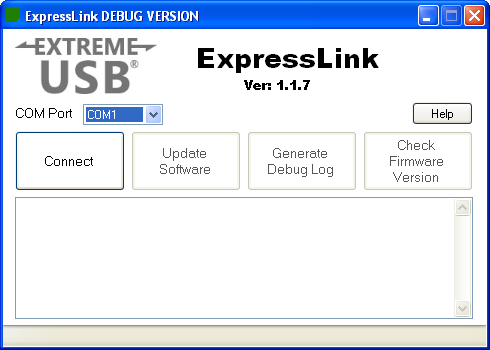


Figure - ExpressLink Debug Mode Interface

# Detailed Design

Because ExpressLink is targeted at a very specific user group and provides limited functionalities, it is designed with very limited freedom in terms of what the user can do and the availabilities of functions at different stage is strictly reinforced. Buttons becomes available at specific times and they are disabled during critical moments where they may interfere with other background works.

As a result, upon running the application, the user is limited to only one function, which is to connect to a COM port; nothing else is available until the application is connected to a COM port.

## ExpressLinkMain.cs

This class contains the core of the software application, including threading, event handling and UI component interactions.

## Opening and Closing Serial COM Port

A flag is used to determine the status of the Connect button. If it is already connected, then the function of the button is turned into disconnect from the device and vice versa. There is, however, a bug in the current .NET Framework.

If the USB cable is unplugged from the computer, resulting in the disappearance of the COM port when it is in use and connected to the device, the application would function normally but when the same COM port is re-opened or closed upon exiting the application, ExpressLink would crash due to an UnauthorizedAccessException. This is the result of a bug that still exists in the .NET Framework 3.5 SP1. This bug should be fixed, according to Microsoft, in the next major release of the .NET Framework.

Currently, a workaround is implemented to handle this. It involves saving the base stream of the COM port to a variable right after it is opened. When closing the port, try to close that base stream first. If the USB was unplugged from the computer, this would result in an UnauthorizedAccessException. This exception can then be caught and discarded so it would not crash the application.

The details of this bug and the numerous proposed workarounds can be found at the Microsoft Connect Web Site:

<https://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=140018>

## Thread

ExpressLink utilizes a worker thread that is responsible of two tasks and can be called by two separate functions.

private Thread threadSharedWorker;

threadSharedWorker = new Thread(new ThreadStart(CheckFirmwareVersion));

threadSharedWorker.IsBackground = true;

threadSharedWorker.Start();

threadSharedWorker = new Thread(new ThreadStart(GenerateDebugLog));

threadSharedWorker.IsBackground = true;

threadSharedWorker.Start();

As shown in the code snip above, this shared worker thread does two different things based on which function calls it. The two functions, Check Firmware Version and Generate Debug Log are mutually exclusive hence there is no sharing violation.

## Generate Debug Log

The GenerateDebugLog() function has its own thread because, though the software environment information gathering is fast, the hardware information retrieval is rather slow. Having it run on a separate thread from the main thread prevents the main UI from freezing while waiting for it to be completed.

The software environment information is obtained from the Environment Class provided in the .NET Framework.

The hardware information is retrieved by utilizing the ManagementObjectSearcher Class where specific hardware components of interests are searched. These hardware components are listed here:

private String[] SystemInfoKeys = { "Win32\_Processor" ,

"Win32\_BIOS" ,

"Win32\_USBController" ,

"Win32\_USBControllerDevice"};

The GetHardwareInfo() function handles the details of the parsing of the information retrieved from the ManagementObjectSearcher and record the parsed string into the log file.

The GenerateDebugLog() function is protected by a Monitor since while this function is writing to the log file, the UpdateLogWindow() function could also be writing logging information received from the Serial Port to the same log file. To prevent having the logging information and the system software, hardware information interleaved together in the log file, a Monitor is used to guarantee mutual exclusion on the file handler.

## Check Firmware Version

The LionsGate software provides the user with a list of UI commands that it accepts over the serial communication port. One of these UI commands is “showSwVer” which returns the version of the firmware currently resides in the device.

The CheckFirmwareVersion() checks to see if the firmware on the device has a different version tag from the one embedded in the ExpressLink executable file. However, because it takes some time for the device to respond to the UI command the function cannot get a result immediately after sending the command over to the device. Furthermore, the function needs to be able to timeout when the device is not responding at all.

The structure of the expected software tag string that the device report is of the following format:

SW Version Tag: LGSW\_vXX\_XX\_XX

SW Build Date: Oct 29 2009

SW Build Time: 13:29:40

Chip Revision: fpga\_build\_2008\_12\_02\_v01\_00\_00

This allows ExpressLink to parse the received characters on the serial communication port and look for strings that matches the patterns shown above. Before sending the command, the number of milliseconds since the system started is recorded and then later compared. If the full version tag strings are still missing after 2 seconds, ExpressLink considers it a failed attempt at retrieving the version from the device and reports an error to the user.

int TimeOutStart = Environment.TickCount;

CheckSWVer = true;

ReportedVersion.Clear();

comPort.WriteLine("showSwVer");

while (CheckSWVer && Environment.TickCount - TimeOutStart < 2000)

{

Thread.Sleep(20);

}

It also sets a flag which tells the UpdateLogWindow() function to look for the version tag strings and parse them accordingly. If the version string is successfully retrieved, a message box will pop up informing the user wheather the version of the firmware in the device matches the one ExpressLink has so the user does not have to upgrade the firmware if it is already up-to-date.

## Serial Communication Port

The StewieFile and StewieFileSender classes are based on the same two classes from the Tigger code base with modifications. Many unused function calls are removed and with some other function modified.

Most of the changes involved were to remove the code related to sending the files as binary as well as functions related to the fact that ExpressLink uses embedded firmware files whereas the old code base takes a packaged Tar file as the input.

Since ExpressLink connects to only one COM port at any given time, so the SerialPort object is initialized with a fixed setting

comPort.BaudRate = 115200;

comPort.Parity = Parity.None;

comPort.DataBits = 8;

comPort.StopBits = StopBits.One;

comPort.Handshake = Handshake.None;

comPort.ReadTimeout = 500;

comPort.WriteTimeout = SerialPort.InfiniteTimeout;

comPort.DataReceived += new SerialDataReceivedEventHandler(ReadCOMByte);

Note that the implementation of the SerialPort data handling in ExpressLink is event driven instead of being polled by a thread with an infinite loop.

Whenever there is data available the DataReceived event is fired and then handled by the ReadCOMByte function. The data received event threshold is the default value, 1 byte. However, this is not guaranteed, as a result, a while loop is implemented in the event handler to take care of it.

while (comPort != null && comPort.IsOpen && comPort.BytesToRead > 0)

{

if (!IsProgramming)

{

UpdateLogWindowDelegate(comPort.ReadLine());

}

else

{

byte ReceivedByte = (byte)comPort.ReadByte();

if (StewieBytes.Contains(ReceivedByte))

{

myStewieSender.SignalRecievedChar(ReceivedByte);

}

else

{

if (ReceivedByte == 0x0A)

{

UpdateLogWindowDelegate(ReceivedChars);

ReceivedChars = "";

}

else

{

ReceivedChars += (char)ReceivedByte;

}

}

}

}

When the application is not in programming mode, all the characters received from the COM port are printed as usual and are read by comPort.ReadLine(). But when it is in programming mode, the characters are read individually by comPort.ReadByte() and each byte is checked against the Stewie protocol to see if it is a synchronization character and handled it accordingly if it is. If it is not a synchronization character, then this character is appended to a string until a carriage return is received and only then is the entire string printed.

## Information Logging

Upon start up, ExpressLink opens up a log file with a unique name in the system temporary folder and all the information received by the application from the connected serial port or events passed to the main form by other classes are recorded and written to this file. As mentioned in the previous section, this function also utilizes a Monitor to protect the file handler.

private void UpdateLogWindow(String text)

The UpdateLogWindow() function is responsible for a variety of things depending on what the string is and the current status of the ExpressLink application.

It checks to see if the string contains "BtLdr:" because the UI command, showSwVer, is not available when the device is operating in the boot loader mode thus it does not make sense to have the CheckFirmwareVersion button available to the user when this string is detected. As a result, the CheckFirmwareVersion button is disabled when UpdateLogWindow() function detects that such string exists in the text it receives. On the other hand, it will enable the CheckFirmwareVersion button if it detects that the device is operating in “Application Mode”.

Furthermore, the textbox showing detailed information about all the communication between the application and the device is only available while the developer is debugging the software application, so it is unnecessary to send the text to the textbox if the textbox is invisible, which is what it would be when the customer is using it.

If the CheckSWVer flag is set to true, then this function will also look for the version tag strings and add them to a list when they are seen.

## Programming the Device

First of all, before start programming the device, it is assumed that the content of the log file is no longer important since it records everything about the previous device. Therefore, the log file handler is closed and then re-opened to clear the content.

Secondly, programming the device runs on a separate thread because the main UI should remain responsive even when the device is being programmed.

The device can be operating in two different modes. One of them is the normal Application Mode and the other is the Boot Loader mode. Currently, there is no way for the software to know which mode the device is in yet ExpressLink has to be able to program the device regardless of its current operating mode.

As a result, a UI command is first send to the device to set it in Boot Loader mode. This UI command, however, will not work if the device is already in the Boot Loader mode. It is assumed that the device is currently working in Application Mode and the "boot" command is always issued first.

comPort.WriteLine("boot");

Thread.Sleep(40);

Because the device will report back some error message if it is already in Boot Loader and cannot accept UI commands, ExpressLink has to wait a few seconds to consume the potential error messages before start sending the firmware files over. Before sending the firmware files, there is a handshake sequence ExpressLink has to perform with the device, without waiting to consume the potential error messages, it is possible for ExpressLink to incorrectly treating the error message as the expected handshake characters and reports an error as a result of it.

After that is done, the files are then being send over to the device one by one using the StewieFileSender class.

SendStewie("BootLoader2.icr", '@');

SendStewie("AsicFirmware.icr", '#');

SetStatusLabelText("Download Complete");

The SendStewie() function initializes a StewieFileSender object with the given file name and the synchronization character.

myStewieSender = new StewieFileSender(fileName, c);

SetStatusLabelText("Downloading " + fileName);

myStewieSender.UpdateMessage += UpdateLogWindowDelegate;

myStewieSender.SendBytes += SendData;

myStewieSender.SendByte += SendData;

myStewieSender.SendFile();

It also registers the StewieFileSender with the appropriate handlers to handle the different functions.

## StewieFileSender.cs

The StewieFileSender class is responsible of initializing a StewieFile object with the file name that is given to the StewieFileSender constructor and assign the StewieFile object with the proper synchronization character. It also handles the synchronization between the device and the ExpressLink application before each file transfer as well as the sending the actual data record over to the device.

As seen in the section above, the entire process of sending a firmware file to the device is started by the StewieFileSender.SendFile() function. It calls a private function PrepareStewieFile() which resets the receivedSyncChar and then start a new synchronization sequence.

private void PrepareStewieFile()

{

WaitSync.Reset();

this.SendSync.Reset();

this.recievedSyncChar = '0';

this.SendStewieFile(this.stewFile);

}

Inside the SendStewieFile() function, ExpressLink would attempt to first synchronize with the device by performing a handshaking sequence with a set of characters that will be sent to the device along with the expected result. If the result is not what it expects, then the device is out of sync and the entire programming operation is aborted.

If the synchronization completed successfully, then ExpressLink will send the Header of the StewieFile first followed by each of the records. For details on the Stewie file format, please check <http://linux.die.net/man/5/srec_stewie>. The TimeoutException, resulted from a non-responding device, and the TargetErrorException, due to device out of sync, are thrown directly to preserve stack integrity and handled externally by the calling function.

## StewieFile.cs

When the StewieFileSender object is first created, it initializes a StewieFile object. This class is responsible of reading the content of the embedded firmware file in its entirety and save that to a byte array. After that, the byte array is parsed and checked against the Stewie file format to confirm it is of the proper format before proceeding further.

This class also holds the static Header and Footer variables of the Stewie file since they are of fixed value for all Stewie files and are hence static.

It also has a few functions to get the length of the Stewie file, as well as the number of records it has.

# Test Plan

ExpressLink can be tested by using it to do what it is suppose to do, which is programming the ASIC units. Other functionalities can be tested by doing something that does not normally happen during regular use.