UNIVERSITY OF WATERLOO
Faculty of Engineering

DESIGN OF AN AUTOMATED ELECTRONIC SCHEMATIC VALIDATION SOFTWARE

Christie Digital Inc.
Kitchener, Ontario

Prepared by
Bilal Junaid
ID 20347427
2B Mechatronics Engineering
May 15, 2012

3519 Bala Drive
Mississauga, Ontario
L5M 7N3
May 15, 2012

Professor Sanjeev Bedi,
Director of Mechatronics Engineering
Department of Mechanical and Mechatronics Engineering
University of Waterloo
200 University Avenue West,
Waterloo, Ontario
N2L 3G1

Dear Professor Bedi,

This report, entitled "Design of an Automated Electronic Schematic Validation Software" was prepared as my 2B Work Report for Christie Digital Inc**.** This is my third work term report. The purpose of this report is to detail the design of software written by myself and a co-worker to automate a schematic validation task assigned to me.

Christie Digital Inc. is globally recognized as a leader in visual technologies. It manufacturers a variety of display technologies for settings such as large audience environments, control rooms, training facilities, cinema, 3D and virtual reality, media, government, simulation, education, and others. Their technology includes DLP Cinema projectors, DLP and LCD projectors, specialized wall projection, and top of the market projection technologies for virtual reality, simulation, and 3D.

I was employed as an Electrical Engineering Intern in their Visual Environments group assisting with the design, testing, and documentation of the electronic boards used in their second generation LED projector specialized for the visualization and simulation industry. The electrical team was led by Tom Pawelko and Stuart Nicholson, and I was supervised and guided by Tom Kochuta, an Advanced Electrical Product Developer.

This report was written entirely by me and has not received any previous academic credit at this or any other institution. It was written to aid in understanding the script for future developers. I would like to thank Stuart Nicholson for explaining the advantages of certain data structures and algorithms used in the software, as well as giving ideas on how to design modular software so it could be used by all Electrical teams at Christie. I received no further assistance.

Sincerely,

Bilal Junaid
ID 20347427

Table of Contents

List of Figures

List of Tables

Summary

This report concentrates on the design of a schematic verification software for printed circuit boards (PCBs) used in the Max projector. The PCBs transfer thousands of signals to each other and to external devices through connector and harness pairs. These signals are expected to be located on predetermined pins on connectors and at a certain voltage level. The purpose of this report is to detail the design of the schematic validation software including a model to develop algorithms for signal and voltage tracing through, selecting an appropriate data structure to store and access relevant data, determining a programming language to develop the software in, and creating a user-friendly method to input and output results of the analysis.

It was determined that elements contained in schematic netlist files could be represented as nodes which contained paths to each other inside a graphical model, thus simplifying the signal and voltage tracing problem into a graph traversal problem. Different data structures were analyzed for their ease of use, ease of implementation, time for storing and accessing data, and readability within code. Several programming languages were analyzed for ease of future code maintenance at Christie, modularity, and speed. Lastly, interfaces for inputting and outputting results were analyzed for ease of use, time of implementation, and modularity.

It was concluded that the graphical model developed to brainstorm signal and voltage tracing algorithm was very useful for future algorithm development. Another major conclusion was that Python and Hash Tables would allow for the fastest code development which could also be maintained by future developers at Christie. Lastly, Excel Spreadsheets were the best method for inputting and displaying data due to time restrictions on the software development.

The major recommendations are that the graphical model should continue to be used to develop future algorithms. The software should be programmed in Python employing Hash Tables to store data structures, but developers should consider C++ in the future. Excel spreadsheets should currently be used; however, the software should be developed in future to use a Graphical User Interface (GUI) to make it easier to use as a single package. The software should also add capability for Voltage Input/Output High/Low verification. Lastly, all Electrical teams at Christie should be trained to use this software and encouraged to analyze their electronic designs with it so further improvements and suggestions can be made.

1.0 Introduction

The crucial process of the transfer, manipulation, and output of video data happens through complex logic and electronic circuitry present at the heart of the projector. This circuitry is implemented at a very small scale on electronic boards, formally known as printed circuit boards (PCBs), mounted within the mechanical exterior. There is a lot of tedious and time consuming verification processes involved in electronic design which may make this profession undesirable to some [1]. This leads to the necessity to automate as much of these processes as possible.

Some verification processes simply cannot be automated and require the electronic designer's time. This includes running prototype PCBs through electromagnetic capability (EMC) and electromagnetic interference (EMI) tests, and determining any detrimental effects unintentional generation and reception of electromagnetic energy may have on the electronic boards. Ensuring the wiring of signals to various devices corresponds to the logic and functionality intended by the designer is also not possible to automate. However, other processes such as verifying certain signals are connected to certain pins of devices or identifying the various voltage levels associated with a signal can possibly be accomplished through software, depending on the design tool used for schematic creation and what aspects of these signals need to be analyzed.

The focus of this report is on the design of automation software which is used to analyze thousands of signals within multiple electronic schematics. These schematics are used to design the PCBs used in the second generation LED projector, referred to as Max for confidentiality purposes, under development at Christie Digital Inc. The software was developed due to the enormous amount of time spent doing such verification manually by electronic designers every time a new projector was designed. The report details the tasks possible to automate, presents an abstract model to simplify complicated electronic schematics, develops algorithms based upon this model, discusses data structures and programming languages to support these algorithms, and determines the best way the results of the automated analysis can be presented to the user.

2.0 Background

Before the software design can be explained, it is necessary to discuss how multiple PCBs may transfer data to each other through connectors and harnesses and the necessity of documenting these connector interfaces. The importance of why these interfaces need to be constantly verified through the schematic as well as the details of this verification process are discussed which formulate the software design problem. Lastly, files which serve as the source of schematic information are presented so it is known what resources were available and how the software design utilized them.

2.1 Connectors and Harnesses

Due to the complexity of the Max projector, it is not possible to encapsulate the entire electronic design on one PCB. Thus, the projector design is split up into five main PCBs, each serving a unique purpose. For example, the SEN2 board is a second generation color sensor board used to calibrate the LEDs within the projector, whereas the QDPC2 (Quad-DVI Projector Control 2) is the main CPU and video data receiver. These boards transfer data to each other through connector and harness pairs. For example, the GDWQ PCB transfers control signals to the SEN2 board through two identical 40-pin Molex connectors located on both boards, and a harness whose two ends contain connectors which mate with the headers on the PCBs. Figure 1 below shows the block diagram of this interface and the corresponding harness drawing:

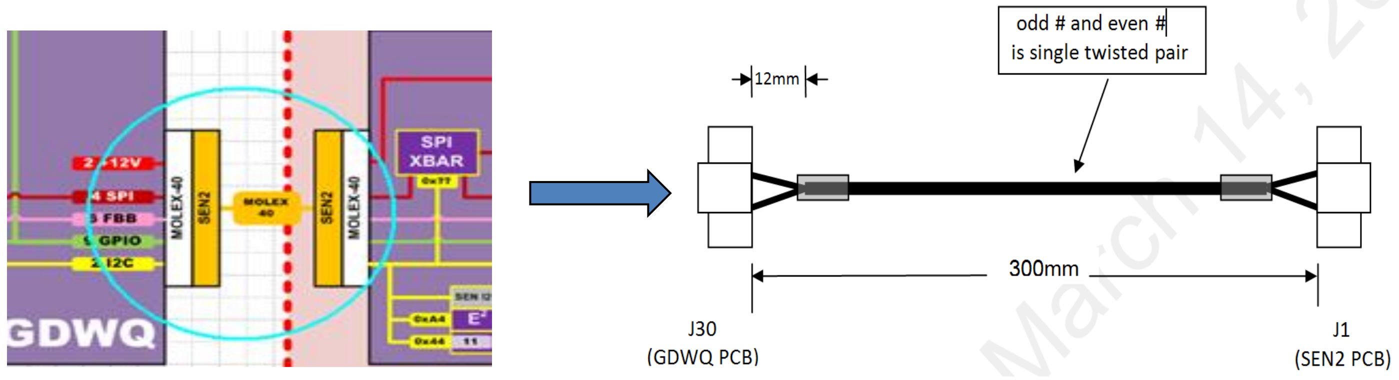
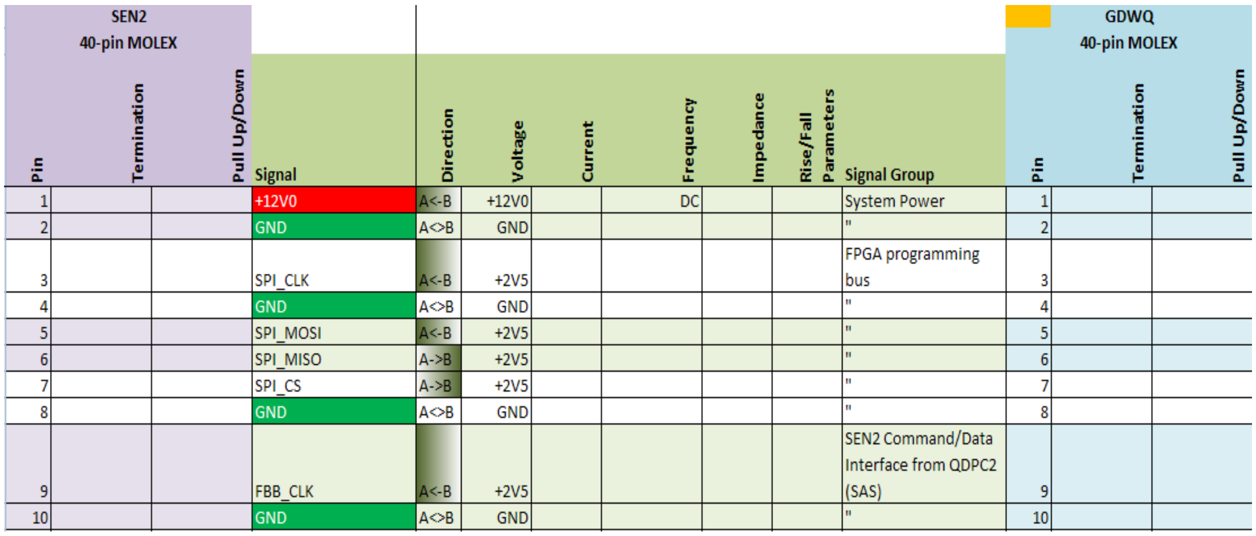


Figure 1: GDWQ and SEN2 Interface

There are also interfaces on PCBs which are used to communicate with external devices and applications. These could be JTAG interfaces which are used to program FPGAs on the PCB, or power connectors for external power supplies. Such interfaces would use connectors and harnesses which would be purchased from the manufacturer of the external devices.

2.2 Interface Control Documents (ICD)

All interfaces on every electronic board are documented in formatted excel spreadsheets called ICDs. The purpose of an ICD is to outline the pin and signal mapping for the connectors, any pull up/pull down resistors as well as the voltage level of the signal (see Appendix A for discussion on Voltage Levels), current, frequency, impedance, and other important parameters which may fully define a signal. Figure 2 shows a portion of an ICD outlining the SEN2 and GDWQ interface. The ICD also lists the connector type and manufacturer part number to be used.



SEN2 40-pin MOLEX											GDWQ 40-pin MOLEX		
Pin	Termination	Pull Up/Down	Signal	Direction	Voltage	Current	Frequency	Impedance	Rise/Fall Parameters	Signal Group	Pin	Termination	Pull Up/Down
1			+12V0	A<-B	+12V0		DC			System Power	1		
2			GND	A<>B	GND					"	2		
3			SPI_CLK	A<-B	+2V5					FPGA programming bus	3		
4			GND	A<>B	GND					"	4		
5			SPI_MOSI	A<-B	+2V5					"	5		
6			SPI_MISO	A->B	+2V5					"	6		
7			SPI_CS	A->B	+2V5					"	7		
8			GND	A<>B	GND					"	8		
9			FBB_CLK	A<-B	+2V5					SEN2 Command/Data Interface from QDPC2 (SAS)	9		
10			GND	A<>B	GND					"	10		

Figure 2: GDWQ and SEN2 ICD

These ICDs are required to be completed by the designers of the electronic boards before they develop the actual design. This is important because it forces them to communicate and make important design decisions about the transfer of signals before the schematic design actually takes place. Once it is known exactly which signals will be available from other boards and which signals must be provided to the interfaces for external devices, the designer can have a clear idea of the functionality of the signals which are used on their boards.

2.3 Validation Task: System Interface Verification

The design of the Max projectors was recently compressed to half its previous schedule. It was extremely crucial that all the interfaces used on every board conformed to the ICD standards electrically as well as mechanically, otherwise, harness and connector rewiring would need to be done for the prototype boards which would consume a lot of time. The verification task assigned was the following:

Mechanical Verification:

- Verify that the physical layout drawing of the connector on the PCB matched the pin orientation detailed in the mechanical data sheet of the connector.

Electrical Verification:

- If the connector is a debug header:
    - No need to check the signals
- If the connector is for external devices:
    - Ensure the pin/signal relationship corresponds to the datasheets of the external devices. Also ensure that voltage levels of the signals correspond to those outlined by the device datasheets.
- If the connector is between PCBs:
    - Verify that pin/signal relationships on the connectors on both PCBs match their respective ICD standards
    - Verify that voltage levels correspond to the values listed in ICDs. This would be done by finding all the devices or pull-up/pull-down resistors the signals may be connected to inside the entire schematic (see Appendix A for details on voltage levels)

The mechanical verification was not possible to automate and verification of connectors of external devices could only be automated partially. The rest of the tasks, which were the most time consuming, could be fully automated as will be described in later sections of the report.

2.4 DxDesigner Netlist File

In order to design a software which can automate the verification task detailed in Section 2.3, it was necessary to have a way of interpreting information in the schematics of the PCBs.

DxDesigner is an electronic schematic design tool created by Mentor Graphics and is used by the Electrical teams at Christie. It generates schematic ASCII netlist files which give details about all the parts and signals within a schematic.

Every device within a schematic, whether it is an integrated circuit (IC), a resistor, capacitor, inductor, or connector, all have reference designators assigned to them. An IC has reference designators starting with U such as U43, resistors with an R such as R21 and so on. Table 1 shows the appropriate reference designator letters for different devices.

Table 1: Reference Designator Letters for Various Devices

Device	Reference Designator Letter(s)
Integrated Circuit	U
Resistor	R
Capacitor	C
Inductor	L
Connector	P, J

Schematic symbols in DxDesigner also have a unique device type property among many other properties. It is not possible for two different devices within a schematic to have the same device type property; however multiple devices may have different reference designators but the same device type parameter. The generated ASCII netlist file contains all reference designators (Refs), their corresponding device types (Type), and pins on various refs (Ref.Pin) which signals are connected to. Figure 3 shows a portion of the netlist file where the resistor R106 is of Type "RUDC4K75" and the Signal "I2C_ID_SCL" is connected to R106.1, L25.1, and U57.3.

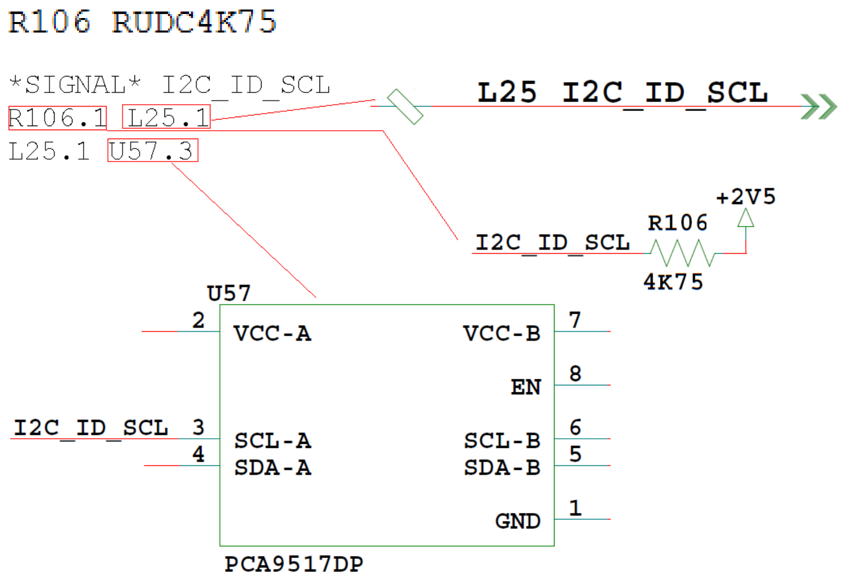


Figure 3: Netlist File Example

3.0 Graphical Model of Netlist Element Relationships

As discussed in Section 2.4, netlists have four main elements: Type, Ref, Ref.Pin, and signals. The relationship between these four elements can be modelled using a graphical modelling approach. When displayed in a graphical model, it is easier to develop algorithms to achieve various functionalities. The four elements discussed can be modelled as nodes, and the relationship between nodes is displayed through an undirected, unidirectional, or bidirectional line also called an edge. When the line is unidirectional or bidirectional, it specifies a path between nodes; specifically, it represents a way for the signal tracing and voltage algorithms discussed in Sections 4.1 and 4.2 to continue traversing nodes to discover results.

If the edge is undirected, it depicts a relationship between the two nodes but there is no path to be traversed. If the edge unidirectional, it represents a path which can be taken from the reference node to the node the edge is directed to, and depicts a relationship between the nodes. If it is bidirectional, then the path between the nodes is also bidirectional, along with a relationship existing between the nodes. Figure 4 below shows the graph model which can be established from a netlist file.

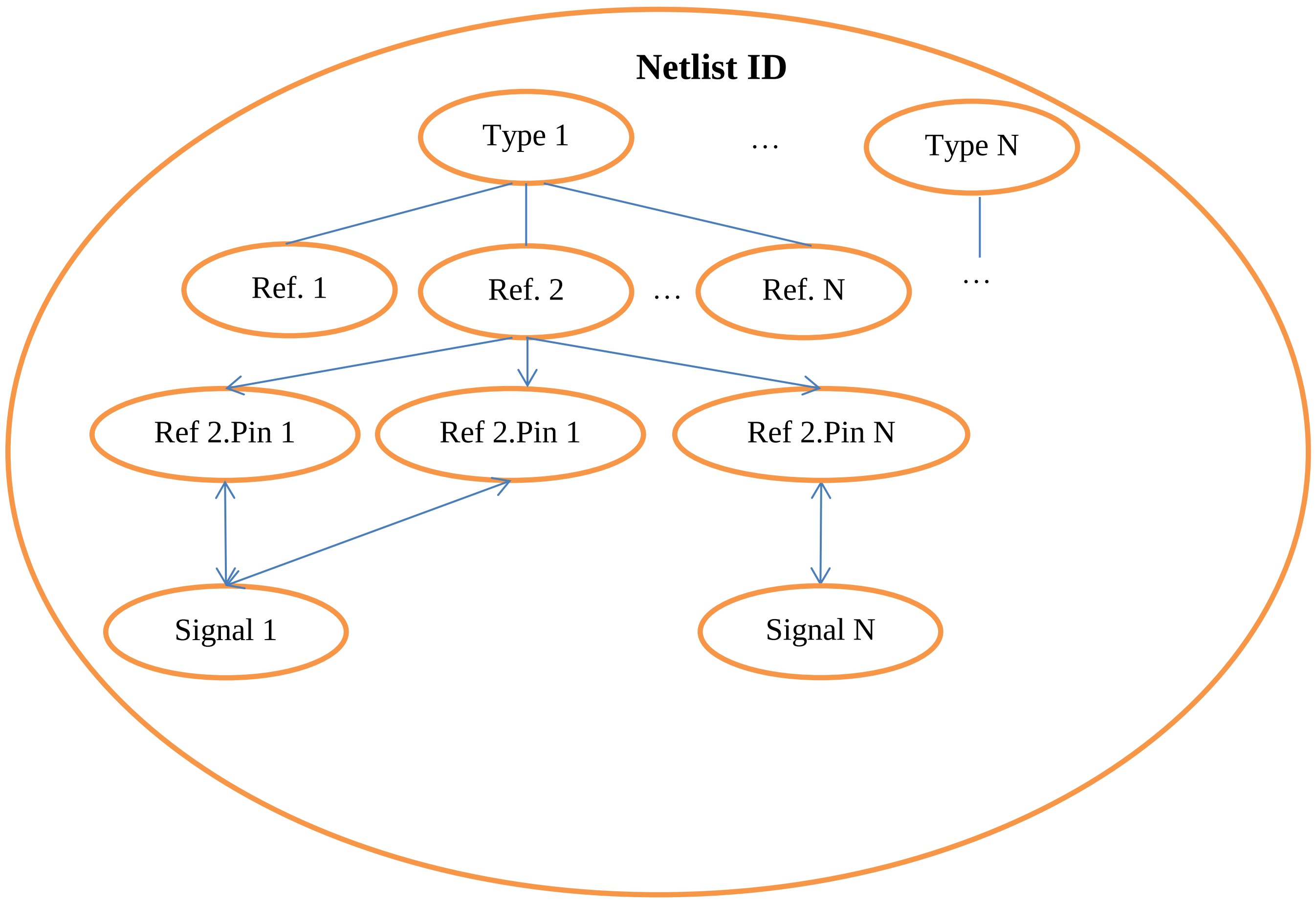


Figure 4: Graph Model of Netlist Elements

In Figure 4, a Netlist ID represents the name of the PCB whose netlist file is being used for example SEN2, or GDWQ. A Device Type can have multiple Refs associated with it. It can also be seen that Refs can have multiple pins associated with them, and there exists a unidirectional path from a Reference Designation to one of its Pins. Signals and Ref.Pins have a bidirectional path to each other. One signal may be attached to multiple Ref.Pin nodes however a Ref.Pin node can only have one signal connected to it.

4.0 Developing Algorithms Based on Graph Model

In Section 2.3, the Electrical verification results in two problems that need to be solved. The first problem is ensuring a set of signals on one board connect to a set of signals on another board. Additionally, it needs to be ensured that external interfaces have a specific pin/signal relationship. All this information can be found in the ICD of the interfaces. The second problem is to ensure all these signals are at a certain voltage level also defined within the ICD. This problem was solved using two similar but separate algorithms; one for tracing signals and another for determining voltage levels of signals. The actual Python implementations of the algorithms can be found in Appendix B. The following sections only discuss the obstacles present and solutions developed using the graph model.

4.1 Modified Graph Model for Signal Tracing

For signal tracing, the user should be able to specify that a signal on one board connects to another board or that a signal on one board connects to a certain pin of a connector on that same board. The current graph model in Figure 4 needs slight modifications in order to overcome obstacles which may occur when allowing this capability. These are listed below:

1) Certain signals may be connected to other signals through pass-through or filtering devices. Signals can have different names but represent the same signal when viewing them from a system perspective. For example, the set of signals shown in Figure 5 come from the connector P1 and are appended with "_C" at the end. These signals go through inductive filtering devices, and the signals on the other side do not have the "_C" anymore. However, when referring to "SPI_CLK", both "SPI_CLK" and "SPI_CLK_C" are encompassed.

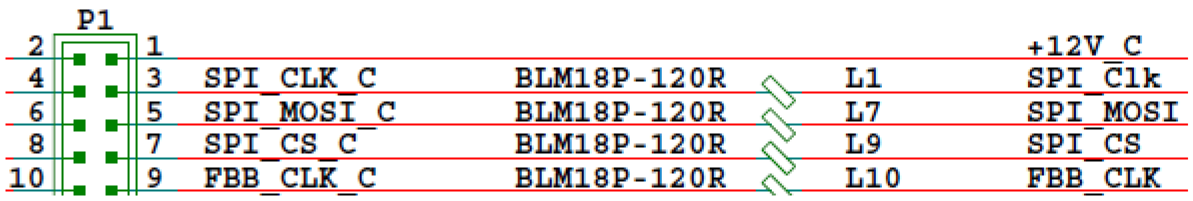


Figure 5: Pass-Through Device Example

The user may specify that the "SPI_CLK" signal needs to connect to P1.3 which is not the case. This scenario can be handled by allowing the user to specify a bidirectional path that may exist between one pin on a device to another pin on the same device. Figure 6 below shows that the "SPI_CLK" node is not connected to node P1.3. However, it is connected to L1.2 and can take a user-defined bidirectional path to node L1.1. This node has a path to "SPI_CLK_C" which eventually connects to node P1.3 and thus the signal tracing would be successful.

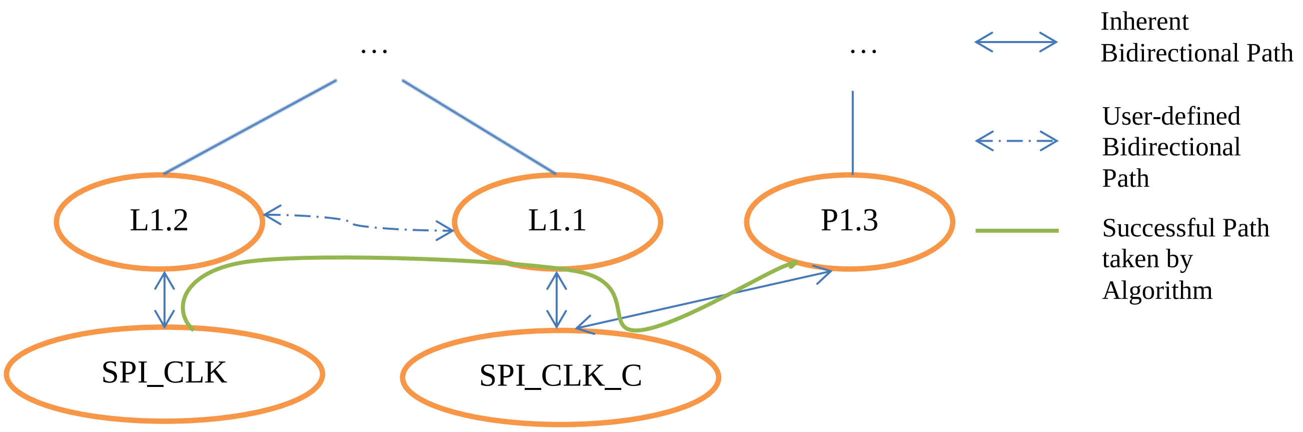

…
…
Inherent Bidirectional Path
User-defined Bidirectional Path
Successful Path taken by Algorithm
L1.2
L1.1
P1.3
SPI_CLK
SPI_CLK_C

Figure 6: Pass-Through Devices Accommodated Through User-Defined Bidirectional Path

Details of how a user may define this relationship is present in the Software User Manual in Appendix C: Section 3.12.

2) Signals on one board may be connected to signals on another board through a connector/harness pair. As an example, consider the case where the user wants to ensure

that the signal “QD0_I2C_SCL_C” on GDWQ connects to “I2C_ID_SCL_C” on SEN2 as shown in Figure 7.

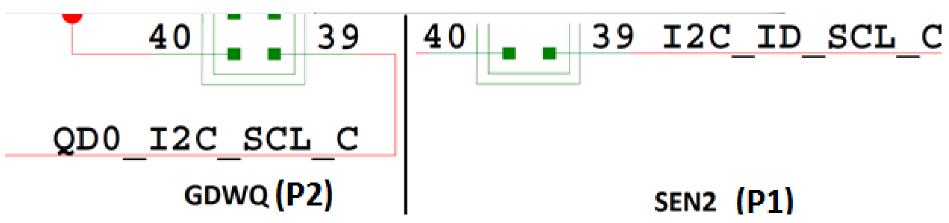
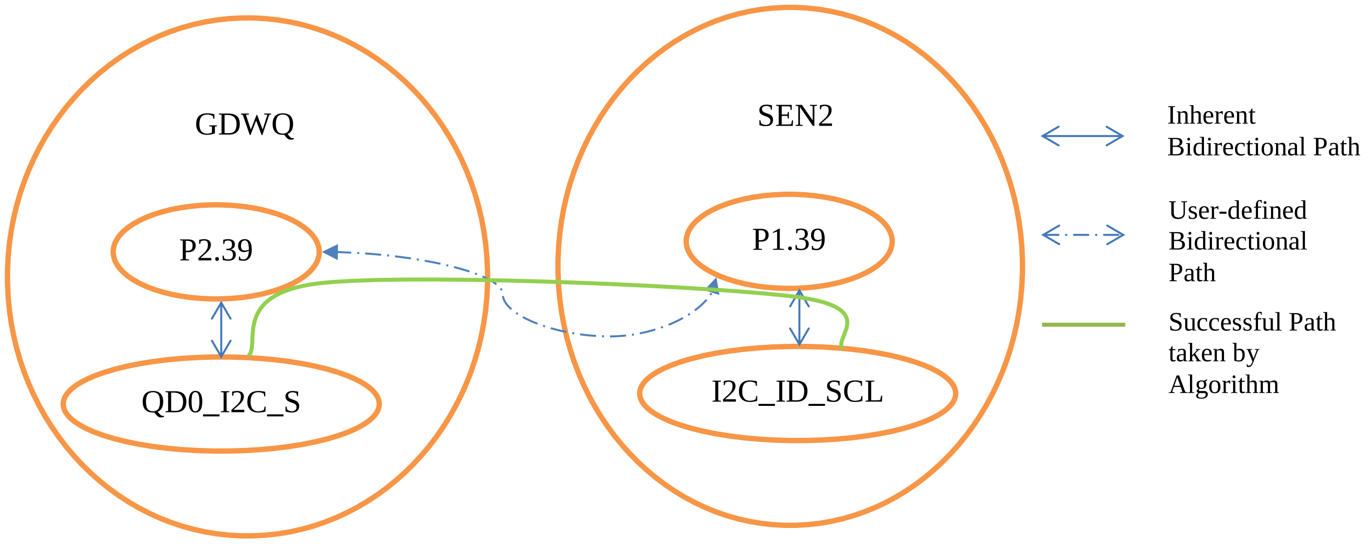


Figure 7: I2C Clock Signal Connection Between GDWQ and SEN2

The graph models of GDWQ and SEN2 are shown below in Figure 8 with no inherent path which the aforementioned signals can take to have a successful trace to each other. However, a user defined bidirectional path can be defined between the connector Ref.Pin nodes on the two different graphs resulting in a successful connection.



GDWQ
SEN2
P2.39
P1.39
QD0_I2C_S
I2C_ID_SCL
Inherent Bidirectional Path
User-defined Bidirectional Path
Successful Path taken by Algorithm

Figure 8: Connector/Harness Relationship Accommodated Through User-Defined Bidirectional Path

Details of how a user would define a relationship between connector pins are given in Appendix C: Sections 3.8 and 3.9.

4.2 Modified Graph Model to Support Voltage Tracing

A signal can have a voltage level associated with it if it is connected to a Rail signal. This connection can either happen through series resistors or through a device pulling signals connected on certain pins to Rail signals connected to its VCC pins, as discussed in Appendix A. A Rail signal in the Max electronic boards are any signals of the form “+nVn” or “PnVn” where n is an integer for example “+3V3” or “P5V0”. Figure 9 below shows the signal “I2C_ID_SCL” having a voltage level of 2.5V through the series resistor R106, as well as the device U57 pulling the same signal on Pin 3 to a Rail signal “+2V5” on its VCC pin.

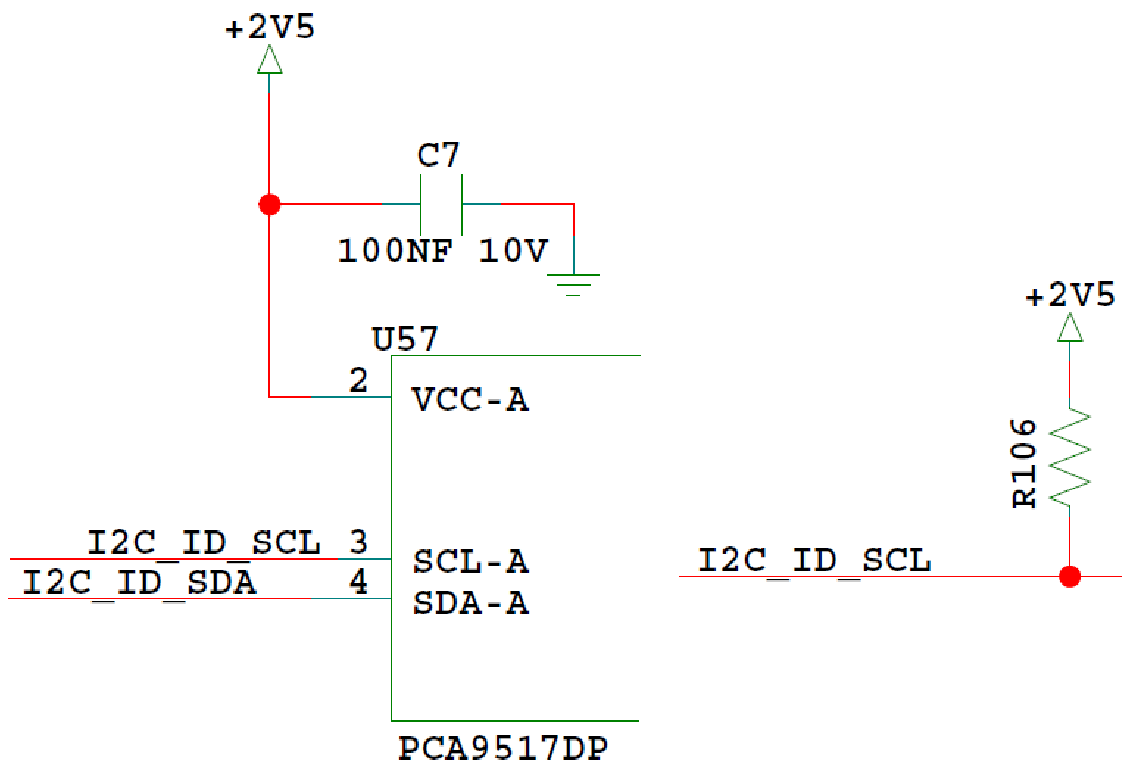


Figure 9: Two Methods of Signals Connected to Rail Signals

To detect voltage levels on signals, the voltage tracing algorithm can follow the user-defined paths which may exist for signal tracing. However, additional considerations need to be made to the graph model to successfully trace voltage levels:

1) To detect whether a signal is connected through a series resistor to a Rail, the software must create an inherent bidirectional path from between the two pins of resistors. This path can only be traversed by the voltage tracing algorithm. Figure 10 below shows a graph model where the Signal "I2C_ID_SCL" on SEN2 connects to R106.1. The software defined voltage path however allows the voltage tracing algorithm to continue onto pin 2 of the resistor, which is connected to a Rail signal "+2V5". Thus the Signal "I2C_ID_SCL" is assigned a voltage pull of +2V5 through R106.

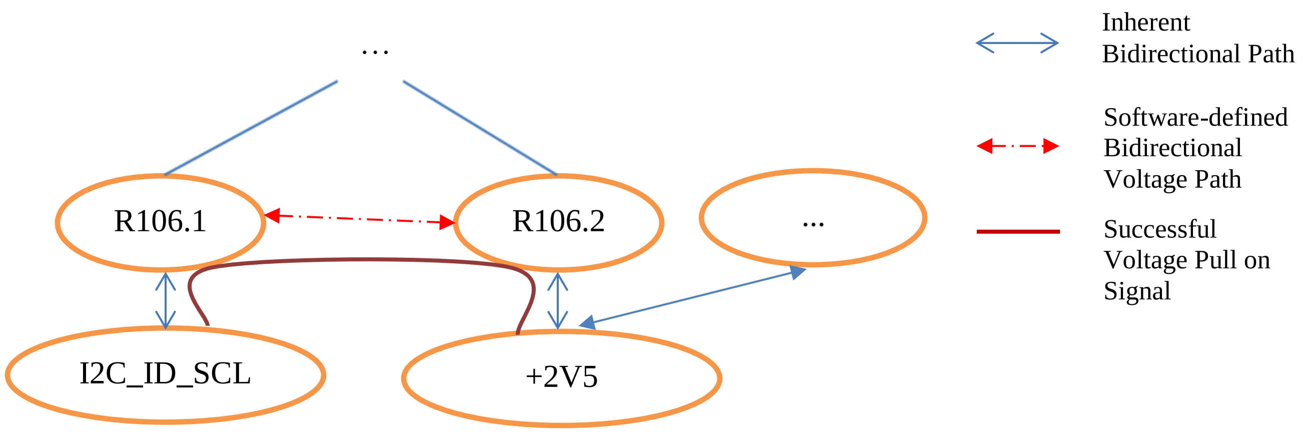

…
R106.1
R106.2
…
I2C_ID_SCL
+2V5
Inherent Bidirectional Path
Software-defined Bidirectional Voltage Path
Successful Voltage Pull on Signal

Figure 10: Pull-up Series Resistors Accommodated Through Software Defined Bidirectional Voltage Path

2) A device pulling signals connected on certain pins to Rail signals on its VCC pins can be accommodated by allowing the user to define a unidirectional path from certain pins on devices to that device's VCC pins. This path can only be traversed by the voltage tracing algorithm. Figure 11 below shows the signal "I2C_ID_SCL" connected to the node U57.3. This node has a unidirectional voltage path to U57.2, the VCC pin, which is

connected to the Rail signal “+2V5”. Thus the voltage tracing algorithm will find this voltage level, and append a voltage value of 2.5V to the signal “I2C_ID_SCL”. If this Rail Signal was “+3V0”, the software would raise a warning that there are conflicting voltages on “I2C_ID_SCL”: +2.5V through R106 and +3.0V through the device U57 on Pin 3.

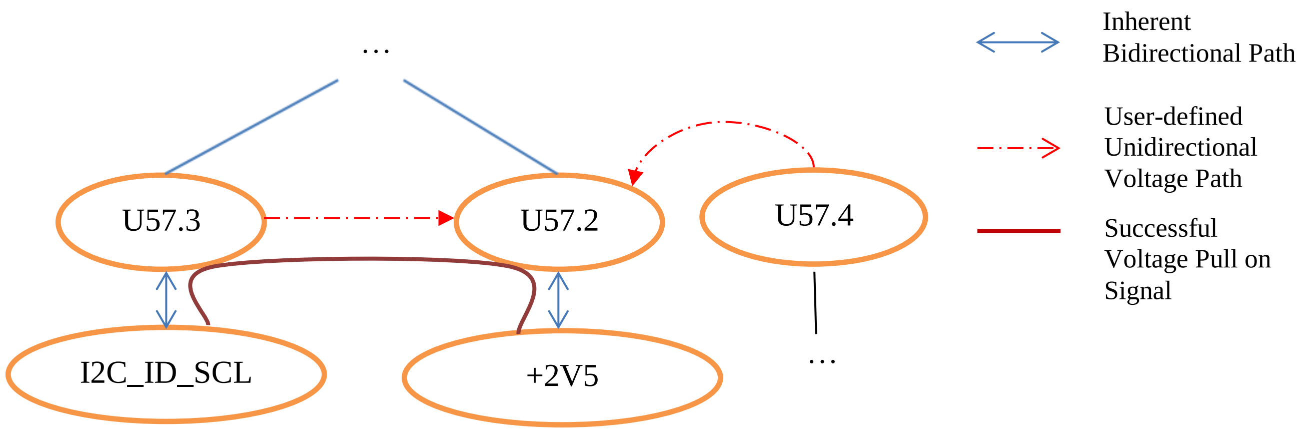

…
U57.3
U57.2
U57.4
I2C_ID_SCL
+2V5
…
Inherent Bidirectional Path
User-defined Unidirectional Voltage Path
Successful Voltage Pull on Signal

Figure 11: Device Pulling Signals to VCC Pin Accommodated Through User-defined Unidirectional Voltage Path

Details of how this relationship is established by the user can be found in Appendix C: Sections 3.15 and 3.16.

5.0 Data Structures

In order to perform any meaningful tasks, the software first needs a way to access all the netlist elements, the inherent relationship between these elements, and the user-defined relationship which may exist. There were two options considered to store this information. The first option was using Hash Tables to store all the relationships globally and access the tables inside functions. The second option was to use an Object Oriented approach where Device Types, Refs, Ref.Pins, and signals would all be Node objects and would store their values as well as relationships with other Nodes as member variables and functions.

5.1 Hash Tables

Hash Tables are also known as associative arrays. Unlike sequenced arrays, which can be indexed by numbers, Hash Tables are accessed through keys, which can either be strings or numbers. These keys correspond to values, and thus Hash Tables contain an unordered set of key and value pairs. Keys must be unique and can be assigned multiple values [2]. An empty Hash Table can be denoted as `{}`. The following table was proposed to store the netlist elements and their relationships:

```
{

"PART": { Ref: Type, … }
"CONNECTION": { Signal: [Ref.Pin, … ], … }
"REF.PIN": { Ref.Pin: Signal, … }
"RAIL": { Signal: Voltage }
"REF": { Ref: [Pin, …], … }

}
```

It can be seen from above that the key "CONNECTION" has a sub-table as a value. This sub-table has signal names as keys, and an array of Ref.Pins as values. This matches the graph model of netlist elements discussed in Section 3.0 which states that a signal can be connected to multiple Ref.Pin nodes. An example of accessing all the pins the signal “I2C_ID_SCL” is connected to would be done as:

```
array_of_pins = netlist_table[“CONNECTION”][“I2C_ID_SCL”]
```

User defined relationship or paths would be contained in other dictionaries. For example to define devices which pull signals on certain pins to their VCC pins, the following hash table structure would be created and populated:

```
{ Type: { Ref.Pin: Ref.Pin, … }, … }
```

Determining the ref.pin node the device Type UPCA9517 pulls Pin 3 on Ref U57 to would be done through the following:

```
vcc_ref_pin = devicepull_table[“UPCA9517”][“U57.3”] # value is U57.2
```

5.1.1 Pros

Hash Tables have a time complexity of O(1) for both storing and accessing elements on average [3]. O(1) corresponds to a constant or minimum runtime achievable by an algorithm. Since the voltage and signal tracing algorithms constantly access netlist elements and relationships, it is necessary that the runtime of accessing this information is very small in order to allow the software to run quickly.

This data structure is extremely easy to implement in Python since it is built into the language, and also very easy to implement in C++ which are the two languages of consideration as will be discussed in Section 6.0. The ease of implementation allows the code to be set up very quickly for data storage, and the rest of the time can be spent on refining the various algorithms.

5.1.2 Cons

The downside of using Hash Tables is they can make the code hard to read. Consider a section of code which loops through all the Ref.Pin nodes a signal `from_signal` is connected to on the Board `from_id`, and stores Ref and Pin into separate variables:

```
…
if from_id in self.syscon_dict["NETLIST"]:

   if from_signal in self.syscon_dict["NETLIST"][from_id]["CONNECTION"]:

         for ref_pin in
   self.syscon_dict["NETLIST"][from_id]["CONNECTION"][from_signal]:

               ref_token = ref_pin.split( '.' )
               ref = ref_token[0]
               pin = ref_token[1]
…
```

The main part of the code is accessing Ref and Pins and storing them in variables, however the conditional statements and the loops access various dictionaries in long lines of code making the main part of the code difficult to read.

5.2 Object Oriented Approach

The proposed objected oriented approach was to model Refs, Ref.Pin, and Signal as separate Classes which would Inherit common functions and variables from a parent node Class. There would also be a Netlist Class which would contain a list of Signal objects which would encompass all signals in the schematic. The node Class would have a member variable called `value`, a variable `node_type`, and member functions called `get_signal_paths()` and `get_voltage_paths()`. `value` would be the actual value of the node, such as “I2C_ID_SCL” or “R39”, and `node_type` would be either “Ref”, “Ref.Pin”, or “Signal”. The functions `get_signal_paths()` and `get_voltage_paths()` would return a list of nodes which can be traversed from the current node, and would be called by the signal and voltage tracing algorithms respectively. The Signal, Ref.Pin, and Reference Designator classes would then define additional variables or functions which are only relevant for themselves, for example the Signal class having a variable `list_ref_pins` which would be a list of Ref.Pin nodes which the Signal object is connected to. An example of how this would be defined in code is given below:

```
class Node():
   def __init__( self, input_value, input_node_type ):
         self.value = input_value
         self.node_type = input_node_type
         …

   def get_signal_paths( self ):
         code…
         return list of nodes

   def get_voltage_paths( self ):
         code…
         return list of nodes
   …

class Signal(Node):
   def __init__( self, name ):
         super(Signal, self).__init__( name, “Signal” )
         list_ref_pins = [] # empty array filled later

   def get_ref_pins():
         return list_ref_pins
   …
```

5.2.1 Pros

The advantage of an Object Oriented approach is that it is very clear and easy to read the code. Instead of having to reference dictionaries with ambiguous keys, Object Oriented programming makes it clear on what is being accessed and from which object. The following code below shows how all the Ref.Pin nodes associated with a certain signal would be accessed with Hash Tables and then in an Object Oriented way:

```
# HASH TABLE
if from_id in self.syscon_table["NETLIST"]:

   if from_signal in self.syscon_table["NETLIST"][from_id]["CONNECTION"]:
```

```
            ref_pins =
self.syscon_table["NETLIST"][from_id]["CONNECTION"][from_signal]


   # Object Oriented
   if from_id_index < self.netlist.size()-1 and from_signal_index <
self.netlist[from_id_index].list_signals.size() -1:

      signal = self.netlist[from_id_index].list_signals[from_signal_index]
      ref_pins = signal.get_ref_pins()
```

It can be seen that the object oriented method allows easier interpretation of the desired task. Accessing a signal from a variable `list_signals` is very obvious, and the function get_ref_pins() called on the accessed signal is self-explanatory.

The object oriented approach also allows more modular and easy to maintain code for the future. Furthermore, the graphical models created to solve the initial algorithms can easily be translated into their Object Oriented equivalent code.

5.2.2 Cons

The disadvantage with the Object Oriented approach was that it would take a long time to properly formulate a perfect class structure for Nodes, Netlists, Reference Designators, Ref.Pin, and signals. Since Inheritance would be used on the Node class, it would need to be ensured that this class supplied all the required functions for the classes which derive from it to function correctly. Object Oriented programming requires thorough planning otherwise it can make a problem more complicated [4].

Another problem was that another member of the team had already begun implementing this design with Hash Tables. If the development was to switch to the Object Oriented model, it would require discarding a large amount of previous code and then rewriting new code which would waste time.

Lastly, using member variables and lists of objects and nodes would not allow the O(1) access which Hash Tables allowed. For the most part, most of the fetching of data would either be O(1) or O(n), depending on how the classes were written. Since the algorithm would be run on thousands of signals, it was important that access to netlist elements and relationships be as fast as possible.

6.0 Programming Language Considerations

It was possible to either use Python or C++ when programming this software. All the developers of the software had equal amount of experience in both programming languages and it needed to be determined which one would be better for the purposes of this project.

6.1 Python

6.1.1 Pros

One of the main advantages of Python is that it has many built in functions which allow easy manipulation and parsing of strings. This project worked with netlist files, and stored all netlist and user-specified commands as strings, requiring intensive string manipulation. The ease of use of the string library would require less time to learn the string manipulation functions.

Another advantage of this language was that most of the automated testing software written within Christie Digital Inc. was in Python. Thus, if this code was to be maintained by other Christie employees in the future, it would be easier for them to understand and manipulate it as there is more exposure to Python within the company.

6.1.2 Cons

Python is generally slower than other mainstream programming languages such as Java or C++ when dealing with CPU bound problems [5]. This speed would be problematic if this software was used to analyze larger electronic projects.

5.2 C++

5.2.1 Pros

It is very easy to write Object Oriented code in C++ as it is a pure Object Oriented language [4]. It would be simple to implement the Graph Model of netlists discussed in Section 3.0 using Classes and Inheritence. The structure of C++ allows developers to write extremely modular code which can make it easy to maintain for someone who knows C++ well. Another advantage of this language was that it was faster in computing CPU bound algorithms allowing faster runtimes for larger projects [5].

5.2.2 Cons

Although the current team of developers working on this project knew C++ very well, the standard within the Electrical teams at Christie is mostly to write software in Python. Thus, writing the code in C++ could pose to be a challenge for certain developers who may work on this in the future.

7.0 User Interface for Input/Output

This software would require a manual process of entering all the desired connections which needed to exist between signals on different boards on the Max projector. It would also require the user to enter specific information about devices to accommodate challenges in the signal and voltage tracing algorithms as discussed in Sections 4.1 and 4.2. Once the analysis was completed on the input, it would also need to be displayed in a user friendly way. There were two approaches considered for Input/Outputs: a Graphical User Interface (GUI) developed specifically for this project or using Excel spreadsheets to enter and display information.

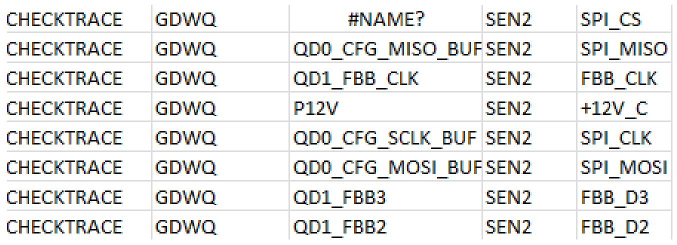
7.1 Excel Spreadsheets

Excel spreadsheets refer to both comma seperated files (.csv) as well as spreadsheets (.xls). Input from the user would be placed in comma separated files, and the output would be written to other .csv files after which they could be pasted in a conditionally formatted Excel spreadsheet

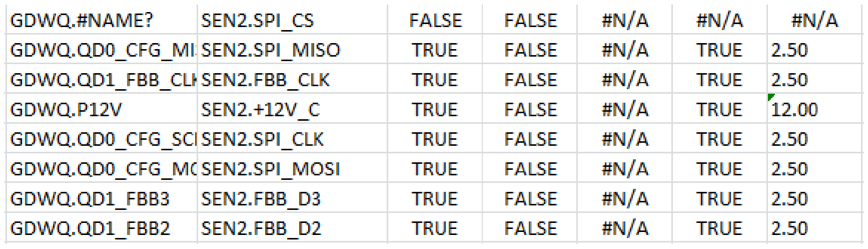
7.1.1 Pros

The software required a manual process of entering information about which signals needed to be connected on which boards, as well as specifying extra information about devices to accommodate the signal and voltage tracing algorithms discussed in Sections 4.1 and 4.2. One of the easiest ways to do this was to copy paste lists of signals already existing within Excel spreadsheets for the ICDs that were created for all the interfaces.

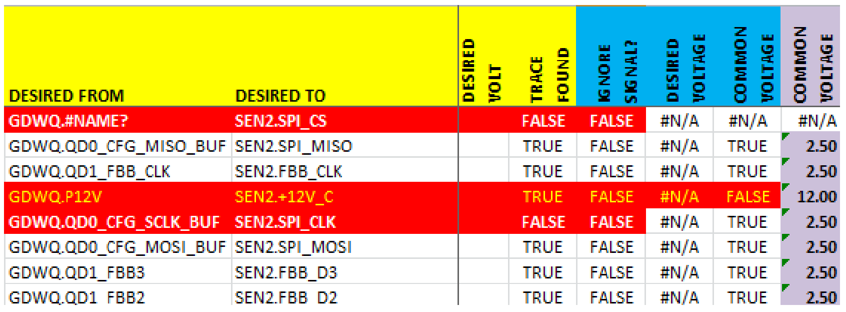
A great advantage of using comma separated files for input and output was they were very easy to read/write in software. These files could be parsed conveniently in code to extract all the relevant user input information, and the output could similarly be written by appending commas between data to organize the data in cells. Figure 12 below shows sample input and output files the software was tested with:


Input .csv file

CHECKTRACE	GDWQ	#NAME?	SEN2	SPI_CS
CHECKTRACE	GDWQ	QD0_CFG_MISO_BUF	SEN2	SPI_MISO
CHECKTRACE	GDWQ	QD1_FBB_CLK	SEN2	FBB_CLK
CHECKTRACE	GDWQ	P12V	SEN2	+12V_C
CHECKTRACE	GDWQ	QD0_CFG_SCLK_BUF	SEN2	SPI_CLK
CHECKTRACE	GDWQ	QD0_CFG_MOSI_BUF	SEN2	SPI_MOSI
CHECKTRACE	GDWQ	QD1_FBB3	SEN2	FBB_D3
CHECKTRACE	GDWQ	QD1_FBB2	SEN2	FBB_D2


Output .csv file

GDWQ.#NAME?	SEN2.SPI_CS	FALSE	FALSE	#N/A	#N/A	#N/A
GDWQ.QD0_CFG_MI	SEN2.SPI_MISO	TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.QD1_FBB_CLI	SEN2.FBB_CLK	TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.P12V	SEN2.+12V_C	TRUE	FALSE	#N/A	TRUE	12.00
GDWQ.QD0_CFG_SCI	SEN2.SPI_CLK	TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.QD0_CFG_MC	SEN2.SPI_MOSI	TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.QD1_FBB3	SEN2.FBB_D3	TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.QD1_FBB2	SEN2.FBB_D2	TRUE	FALSE	#N/A	TRUE	2.50


Output .csv file pasted
into Condtionally
Formatted Spreadsheet

DESIRED FROM	DESIRED TO	DESIRED VOLT	TRACE FOUND	IGNORE SIGNAL?	DESIRED VOLTAGE	COMMON VOLTAGE	COMMON VOLTAGE
GDWQ.#NAME?	SEN2.SPI_CS		FALSE	FALSE	#N/A	#N/A	#N/A
GDWQ.QD0_CFG_MISO_BUF	SEN2.SPI_MISO		TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.QD1_FBB_CLK	SEN2.FBB_CLK		TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.P12V	SEN2.+12V_C		TRUE	FALSE	#N/A	FALSE	12.00
GDWQ.QD0_CFG_SCLK_BUF	SEN2.SPI_CLK		FALSE	FALSE	#N/A	TRUE	2.50
GDWQ.QD0_CFG_MOSI_BUF	SEN2.SPI_MOSI		TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.QD1_FBB3	SEN2.FBB_D3		TRUE	FALSE	#N/A	TRUE	2.50
GDWQ.QD1_FBB2	SEN2.FBB_D2		TRUE	FALSE	#N/A	TRUE	2.50

Figure 12: Excel Spreadsheets Used for User IO

7.1.2 Cons

The main disadvantage with excel spreadsheets was that they required references to a lot of different files which needed to be maintained. The user would need to populate input files and then reference them within the software. The software would generate output .csv files which the user would need to open, copy, and then paste into Conditionally Formatted and Macro Enabled Excel spreadsheets.

7.2 Graphical User Interface (GUI)

Graphical User Interfaces could be made using the Qt Toolkit which integrates both into C++ and Python depending on whichever programming language was chosen. The desired Graphical User Interface included the ability to view all user-specified commands in an easy and readable manner. The results of the analysis would be displayed in a new window after the software would complete allowing the user to analyze the tracing and voltages of signals.

7.2.1 Pros

Having a graphical user interface would allow the software to be run in one single package without the need to deal with multiple files. This was desirable since the software became complicated as more user-input commands were introduced and the capabilities of the software were increased. Allowing everything to be incorporated into one GUI, and disturbing the software as an .exe file was very convenient compared to multiple excel files. Also, if this software was to become standardized within Christie, it would eventually require it to be interfaced with a GUI.

7.2.2 Cons

A GUI was very difficult to program, especially to allow input to be as convenient as excel files. A lot of the information about signals and devices needed to be specified, and it was not feasible to expect the user to manually type all of this. A GUI with insertion and selection capabilities to even remotely match the capabilities provided by excel would require a lot of intelligent programming.

Time was also a very large factor when considering a GUI. The main purpose of this project was to verify signals on the electronic boards as soon as possible, and designing a GUI would take away valuable time spent verifying signals.

8.0 Conclusions

The graph model is an efficient way to develop algorithms for this software.

This was determined due to the ease of identifying obstacles which existed for the signal and voltage tracing algorithms, and defining nodes and paths, either inside the software or through user-definition to overcome the obstacles. The graph model is very useful for developing algorithms for netlist analysis because all operations which can possibly be performed eventually reduce to traversing paths between nodes to reach a node with a certain value.

Hash Tables are the superior method of storing netlist elements and their relationships.

The main reason Hash Tables are superior in this project is because of the tight deadline. An Object Oriented approach would take too long to formulate correctly whereas Hash Tables can easily be set up and used. Another main reason is that using an Object Oriented model would require discarding previous code written by another co-worker who used Hash Tables to write part of the tracing algorithms. Hash Tables also guarantee a constant runtime on average allowing for fast execution, whereas the Object Oriented model could either be constant or linear runtime.

Python is the easier programming language to solve this task and maintain in the future.

Python contains a large variety of string manipulation functions built into the language which are very useful as a large portion of the programming involves string parsing and manipulation. Writing this software in Python also makes it easier to maintain as most of the software written by Electrical teams at Christie is in Python. C++ allows faster and modular code, but not enough developers know it well at Christie.

Excel Spreadsheets are the best way to receive user input and display the results of the analysis.

Excel spreadsheets allow the manual task of specifying signal connections to be a task of copy-pasting, whereas it would very difficult to implement such a smooth copy-pasting procedure in GUIs. Comma separated files are also extremely easy to parse requiring very little intelligent code. GUIs do have the advantage of allowing the software to be executed through one stand alone .exe file, however, coding the GUI would take a very long time.

9.0 Recommendations

Based on the analysis and conclusions in this report, the algorithms should be further developed using the Graph Modelling approach in order to make them easier to implement and understand in the future. The software should be written in Python employing Python Dictionaries which are built in Hash Table structures, due to their ease of use and maintenance in the future.

It is recommended that initially, Excel spreadsheets should be used within the software for handling input and displaying the results of analysis. However, once the analysis for the signals on the Max projector is complete, this software should be further developed in the future to incorporate a GUI. This would make the software easier to use by other Electrical teams at Christie, and can eventually grow into a standardized tool to be used for schematic analysis.

The software should incorporate Voltage Input and Output High/Low detection between signals connected on multiple devices. This concept is discussed in Appendix A. This would be another level of verification which can be done and will reduce the time spent verifying this manually.

Lastly, it is recommended that an effort is made to train other Electrical teams at Christie to use this software and encourage its use. It is an excellent tool which saves a lot of time, and the more it is used, the further it can be refined with new features and capabilities.

10.0 Glossary

PCB – Printed Circuit Board

LED - Light emitting diode

Max - Second generation solid state illuminated projector under development at Christie Digital

SEN2 - Sensor Board on Max Projector

GDWQ - Formatter Board on Max Projector

QDPC2 - CPU Board on Max Projector

JTAG - Joint Test Action Group for Electronic Testing

FPGA - Field Programmable Gate Array

ICD - Interface Control Document

IC - integrated Circuit

Ref - Reference Designator

Type - Device Type parameter of DxDesigner Symbols

Ref.Pin - Pin Number located on a Reference Designator

GUI - Graphical User Interface

11.0 References

[1] M. D. Brinbaum, *Essential Electronic Design Automation, 1st ed.* Columbus, OH: Prentice Hall, 2003, pg 100-120

[2] Python Software Foundation, “Dictionaries in Python.” [Online]. Available: http://docs.python.org/tutorial/datastructures.html#dictionaries. [Accessed: May 2, 2012].

[3] Python Software Foundation, “Time Complexity of Data Strucutres.” [Online]. Available: http://wiki.python.org/moin/TimeComplexity. [Accessed: May 1, 2012].

[4] D. E. Knuth. *The Art of Computer Programming, vol. 1, 3rd ed*. Boston, MA: Addison-Wesley, 1997, pg 101.

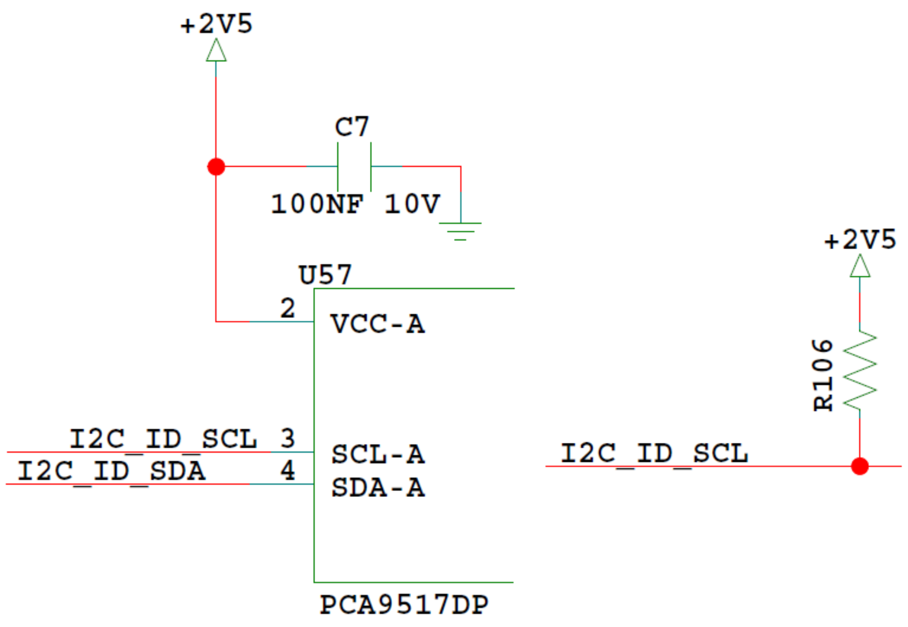
[5] S. Pigeon, “Python Speed for CPU Bound Problems.” [Online]. Available: http://hbfs.wordpress.com/2009/11/10/is-python-slow/. [Accessed: May 4, 2012].

Appendix A – Voltage Levels of Signals

When discussing voltage levels of signals in electronic design, the following elements need to be considered:

1) Signals being pulled up to certain voltage levels through pull-up resistors
2) Signals connected to pins on devices which may pull that signal to a certain voltage. This voltage would be the same as the supply voltage for that device
3) The voltage input/output high/low values between devices fall within the appropriate threshold

These two scenarios are shown in the Figure below where R106 pulls the signal "I2C_ID_SCL" to a 2.5V voltage rail and the device U57 pulls the signal "I2C_ID_SCL" connected to pin 3 to the its 2.5V voltage rail on the VCC pin:



Voltage levels are important to consider because devices have certain voltage input/output high and low levels in their data sheet, which define what they consider a low or high voltage value. For example, a voltage input low value on a device data sheet represents the voltage value which this device considers as a low or a “0” as an input. This would also have a min and max value for example a input low voltage could have a minimum value of 0.1V and a maximum input low voltage of 0.8V. Similarly, the device would have voltage input high min and max values. When a device has a signal connected to its output, and it is driving this signal into an input pin of another device, it must be ensured that these voltage values fall within each other’s threshold. For the most part, when two devices are powered by the same VCC voltage, it is usually sufficient to consider that their voltage input/output high/low values fall within each other’s range. However as an example, one device may output a maximum output low voltage of 0.8V, and it drives the input of a device which can detect a maximum input low voltage of 0.7V. Thus, it is possible that Device 1 may output 0.8V expecting Device 2 to recognize this voltage as low however Device 2 only detects low voltages to a maximum of 0.7V and it is possible that 0.8V may just be considered a high voltage. These considerations are important to take into account, and datasheets must be checked further to determine the validity of the voltage threshold of different devices.

Appendix B – Signal and Voltage Tracing Algorithm

Signal Tracing:

```
   def trace_netlist_signal( self, from_id_signal, to_id_signal, info_dict ):
   """
   trace_netlist_signal

   Locate all MAPs attached to a netlist signal and call
       trace_connection to see if they connect to the destination signal
   """

   ##print "trace_netlist_signal( %s, %s )" % ( from_id_signal, to_id_signal )
   ##print info_dict

   trace_success = False
   from_token = from_id_signal.split( '.' )
   to_token = to_id_signal.split( '.' )
   valid_params = True
   if len( from_token ) > 1:
       from_id = from_token[0]
       from_signal = from_token[1]
   else:
       valid_params = False

   if len( to_token ) > 1:
       to_id = to_token[0]
       to_signal = to_token[1]
   else:
       valid_params = False

   if valid_params and from_id_signal == to_id_signal:
       ##print "\n***\n\n*** SUCCESS\n\n"
       trace_success = True
   # do not attempt to trace GND signals since this function is only called on
ID.signal_name. If the CHECKTRACE specified in input
   # .csv file was true, the from_signal would == to_signal and the first condition would
be met. Otherwise, we end up traversing
   # thousands of ground connections.
   elif valid_params and from_signal != "GND" and to_signal != "GND":
       trace_success = False
       if from_id in self.syscon_dict["NETLIST"]:
           if from_signal in self.syscon_dict["NETLIST"][from_id]["CONNECTION"]:
               for ref_pin in
self.syscon_dict["NETLIST"][from_id]["CONNECTION"][from_signal]:
                   ref_token = ref_pin.split( '.' )
                   ref = ref_token[0]
                   pin = ref_token[1]
                   traced = False
                   # See if signal is attached to connection to harness
                   if from_id in self.syscon_dict["CONNECTION_REFS"] and not
trace_success:
                       if ref in self.syscon_dict["CONNECTION_REFS"][from_id]:
                           traced = True
                           path_id = "%s.%s" % ( from_id, ref_pin )
                           # Don't follow a path we've been down before
                           if path_id not in info_dict["PATH"]:
                               test_path = copy.copy( info_dict["PATH"] )
                               test_path.append( path_id )
                               test_info_dict = copy.copy( info_dict )
                               test_info_dict["PATH"] = test_path

                               ( id_signal, test_info_dict ) = self.trace_connection(
from_id, ref_pin, test_info_dict )
                               if len( id_signal ) > 0:
                                   ##print "Looking for path from %s to %s" % (
id_signal, to_id_signal )
                                  ( trace_success, test_info_dict ) =
self.trace_netlist_signal( id_signal, to_id_signal, test_info_dict )



                   # If signal not attached to connection to harness,
                   # see if signal goes through device
                   if not traced:
                       ref_type = self.syscon_dict["NETLIST"][from_id]["PART"][ref]
                       path_id = "%s.%s" % ( from_id, ref_pin )
                       # Don't follow a path we've been down before
                       if ref_type in self.syscon_dict["DEVICE"] and ref_type not in
self.syscon_dict["IGNORE"]["DEVICE"] and path_id not in info_dict["PATH"]:
                           traced = True
                           test_path = copy.copy( info_dict["PATH"] )
                           test_path.append( path_id )
                           test_info_dict = copy.copy( info_dict )
                           test_info_dict["PATH"] = test_path
                           ( id_signal, test_info_dict ) = self.trace_device( from_id,
ref_pin, ref_type, test_info_dict )
                           if len( id_signal ) > 0:
                               ##print "Looking for path from %s to %s" % ( id_signal,
to_id_signal )
                               ( trace_success, test_info_dict ) =
self.trace_netlist_signal( id_signal, to_id_signal, test_info_dict )

                   if trace_success:
                       info_dict = test_info_dict
                       break

   return ( trace_success, info_dict )
```

Voltage Tracing:

```
   def pull_netlist_signal( self, id, signal, info_dict, pull_path ):
   """
   pull_netlist_signal( id, signal, info_dict ):

   Check all connections to a signal to see if they are resistors to a rail
   Follow straight-through devices if needed
   """

   print "\npull_netlist_signal( %s, %s, %s, %s )" % ( id, signal, info_dict,
pull_path )
   ##print info_dict
   ##print pull_path
   ignore = False
   id_signal = "%s.%s" % (id, signal)
   rail = signal in self.syscon_dict["NETLIST"][id]["RAIL"]

   # if a signal is ignored, all the signals that trace to this ignored signal should
also be added to ignored list.
   # This is not done during tracing because the tracing algorithm does not traverse
all paths available to it. It only traverses
   # until the desired path is met
   if id_signal in self.syscon_dict["IGNORE"]["SIGNAL"]:
       ignore = True
       for temp_id_signal in info_dict["PULL_ID_SIGNAL"]:
           if temp_id_signal not in self.syscon_dict["IGNORE"]["SIGNAL"]:
               self.syscon_dict["IGNORE"]["SIGNAL"].append(temp_id_signal)
   elif signal[0:2] != "NC" and id_signal not in info_dict["PULL_ID_SIGNAL"] or rail:
       if rail:
           print "%s is rail" % signal
           if len( pull_path ) > 1:
               id_ref_pin = pull_path[-2]
               [ id, ref, pin ] = id_ref_pin.split( '.' )
               pull_part = self.syscon_dict["NETLIST"][id]["PART"][ref]
               pull_info = "%s.%s (%s.%s) to %s" % ( id, pull_part, ref, pin, signal )
           else:
               pull_info = "direct to %s" % signal
           info_dict["PULL"].append( pull_info )
           info_dict["VOLT"].append( self.syscon_dict["NETLIST"][id]["RAIL"][signal] )

       else:
           info_dict["PULL_ID_SIGNAL"].append(id_signal)
           test_id = ""
           test_signal = ""
           for ref_pin in self.syscon_dict["NETLIST"][id]["CONNECTION"][signal]:

               if ignore == True:
                   break

               [ ref, pin ] = ref_pin.split( '.' )
               id_ref_pin = "%s.%s" % ( id, ref_pin )
```

```
            # See if ref has specified voltage
                if id_ref_pin in self.syscon_dict["REFVOLT"]:
                    pull_volt = self.syscon_dict["REFVOLT"][id_ref_pin]
                    pull_info = "%s specified at %.2f" % ( id_ref_pin, pull_volt )
                    print "Device %s specifies %s is %.2f" % ( ref, signal, pull_volt
)
                    info_dict["PULL"].append( pull_info )
                    info_dict["VOLT"].append( pull_volt )

                # See if ref is resistor and not dnp resistor or a capacitor and not
dnp capacitor and not ignored
                if ref not in self.syscon_dict["IGNORE"]["DEVICE"] and ref[0] == "R"
and ref[1].isdigit() and ref in self.syscon_dict["NETLIST"][id]["PART"] and \
                   self.syscon_dict["NETLIST"][id]["PART"][ref].lower().find("dnp") ==
-1:
                    if pin == "1":
                        pull_pin = "2"
                    else:
                        pull_pin = "1"

                    pull_ref_pin = "%s.%s" % ( ref, pull_pin )
                    # See if opposite pin of resistor exists
                    if pull_ref_pin in self.syscon_dict["NETLIST"][id]["REF.PIN"]:
                        pull_signal =
self.syscon_dict["NETLIST"][id]["REF.PIN"][pull_ref_pin]
                        # See if resistor is connected to a rail but ignore pull downs
to GND as they cause unnecessary voltage conflicts
                        # If resistor not connected to rail, continue checking in case
it is series resistor
                        rail = pull_signal in self.syscon_dict["NETLIST"][id]["RAIL"]
                        if not rail:
                            ( info_dict, ignore ) = self.pull_netlist_signal( id,
pull_signal, info_dict, pull_path )
                        elif rail and not
self.syscon_dict["NETLIST"][id]["RAIL"][pull_signal] == 0.0:
                            pull_part = self.syscon_dict["NETLIST"][id]["PART"][ref]
                            pull_info = "%s.%s (%s) to %s" % ( id, pull_part,
pull_ref_pin, pull_signal )
                            info_dict["PULL"].append( pull_info )
                            info_dict["VOLT"].append(
self.syscon_dict["NETLIST"][id]["RAIL"][pull_signal] )
                            print "Resistor %s (%s) connects %s to rail %s" % (
pull_ref_pin, pull_part, signal, pull_signal )

                # See if signal goes through device
                ref_type = self.syscon_dict["NETLIST"][id]["PART"][ref]
                path_id = "%s.%s" % ( id, ref_pin )
                # Don't follow a path we've been down before
                if ref_type not in self.syscon_dict["IGNORE"]["DEVICE"] and ref_type
in self.syscon_dict["DEVICE"] and path_id not in pull_path:
                    pull_path.append( path_id )
                    # Check if device pin has fixed voltage expectation
                    if ref_type in self.syscon_dict["DEVICEVOLT"]:
                        if pin in self.syscon_dict["DEVICEVOLT"][ref_type]:
```

```
                                  info_dict["VOLT"].append( pull_volt )
                    ##print "Tracing device:", pull_path
                    ( id_signal, dummy ) = self.trace_device( id, ref_pin, ref_type, {
"PATH": pull_path } )
                    if len( id_signal ) > 0:
                        [ test_id, test_signal ] = id_signal.split( '.' )
                        ##print "Looking for path from %s to %s" % ( id_signal,
to_id_signal )
                        ( info_dict, ignore ) = self.pull_netlist_signal( test_id,
test_signal, info_dict, pull_path )

                # See if signal goes through device voltage linked pin
                # Don't follow a path we've been down before
                if ref_type not in self.syscon_dict["IGNORE"]["DEVICE"] and ref_type in
self.syscon_dict["DEVICEPULL"] and path_id not in pull_path:
                    pull_path.append( path_id )
                    # Check if device pin has fixed voltage expectation
                    if ref_type in self.syscon_dict["DEVICEVOLT"]:
                        if pin in self.syscon_dict["DEVICEVOLT"][ref_type]:
                            pull_volt = self.syscon_dict["DEVICEVOLT"][ref_type][pin]
                            pull_info = "%s.%s (%s) to %.2f" % ( id, ref_pin, ref_type,
pull_volt )
                            print "Device %s (%s) connects %s to %.2f" % ( ref,
ref_type, signal, pull_volt )
                            info_dict["PULL"].append( pull_info )
                            info_dict["VOLT"].append( pull_volt )
                    ##print "Tracing device:", pull_path
                    ( id_signal, dummy ) = self.trace_device( id, ref_pin, ref_type, {
"PATH": pull_path }, device_key="DEVICEPULL" )
                    if len( id_signal ) > 0:
                        [ test_id, test_signal ] = id_signal.split( '.' )
                        ##print "Looking for path from %s to %s" % ( id_signal,
to_id_signal )
                        ( info_dict, ignore ) = self.pull_netlist_signal( test_id,
test_signal, info_dict, pull_path )

                # See if signal is attached to connection to harness
                if id in self.syscon_dict["CONNECTION_REFS"]:
                    if ref in self.syscon_dict["CONNECTION_REFS"][id] and path_id not in
pull_path:
                            pull_path.append(path_id)
                            test_info_dict = copy.copy( info_dict )
                            test_info_dict["PATH"] = pull_path  # trace connection
function will continue adding to pull_path
                            ( id_signal, test_info_dict ) = self.trace_connection( id,
ref_pin, test_info_dict )
                            if len( id_signal ) > 0:
                                pull_path = test_info_dict["PATH"]
                                [ test_id, test_signal ] = id_signal.split( '.' )
                                ( info_dict, ignore ) = self.pull_netlist_signal(
test_id, test_signal, info_dict, pull_path )

    if ignore == True:
        info_dict["PULL"] = []
        info_dict["VOLT"] = []

    return ( info_dict, ignore )
```

Appendix C – Software User Manual

Note: This reference may be outdated as the script is further developed. The last edited date can be viewed in the footer of this document.

Tags

The following tags are used extensively to explain the parameters required for the script commands as well as other sections of this reference.

Tag	Explanation	Example(s)
ID	Refers to the electronic boards	SEN2, QDPC2
HARNESS_ID	Refers to arbitrary tag assigned to harnesses	SAS0
TYPE	Refers to the DEVICE parameter in DxDesigner symbols	EP3C16U256C7N
REF	Reference Designator	U32, R11, P1
PIN	Pin Number	12, A11
SIGNAL	Net name	I2C_SDA_ID, STEALTH_MODE

1.0 File Explanations

The source code files are system_connections.py and netlist.py. The script can be executed by browsing to the folder where the source code is located through command prompt and type the command (note this command could change in future):

```
system_connections.py –f “INPUT_FILE.csv” –o OUTPUT_FILE_TAG –v VALUE
```

For example:

```
system_connections.py –f “main.csv” –o Main –v 0
```

INPUT_FILE: name of the main input .csv file. It is usually main.csv however it can be any file the script should take as the input

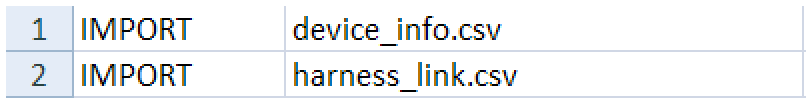
OUTPUT_FILE_TAG: The script outputs 3 .csv files of the form: OUTPUT_FILE_TAG_check.csv, OUTPUT_FILE_TAG_map.csv, and Volt_check.csv. For example, if the OUTPUT_FILE_TAG was “Main”, then the script would output Main_check.csv, Main_map.csv, Volt_check.csv.

VALUE: If this is 0, then the script does not output Volt_check.csv which is a voltage analysis of every single netname in every netlist file. If this is 1, then Volt_check.csv contains voltage analysis of all netnames.

For the remainder of this document, it is assumed that INPUT_FILE is “main” and OUTPUT_FILE_TAG is “Main” in order to give examples.

1.1 Input .csv File(s)

The input .csv file(s) contain the commands which tell the script what to do. These commands will be discussed in later sections. A typical setup is to have a main.csv file, in which the IMPORT command is used to import other .csv files, as shown below:



1	IMPORT	device_info.csv
2	IMPORT	harness_link.csv

This allows the user to categorize commands in separate files so there is not one large .csv file. The script parses commands regardless of order, so all commands can be scattered over multiple files, as long as these files are imported in the main.csv file specified in command line.

1.2 Output .csv Files

There are three output .csv files. Assuming that the OUTPUT_FILE_TAG is “Main”, then these would be Main_check.csv, Main_map.csv, and if VALUE is 1 for –v, then the third file is Volt_check.csv.

Main_check.csv contains the results of all CHECKTRACE and CHECKVOLT commands which were specified in the input.csv files, and shows whether the script was able to find the desired trace specified. It also contains information on voltage levels associated with the signal names that were specified to be traced.

Main_map.csv contains the pinouts of all the connectors as well as devices that the user specified to be outputted using the MAP and DEVICEPIN commands. The pinout orientation can be specified by the user to match how it actually looks on the devices in the schematic.

Lastly, Volt_check.csv contains information in the same format as Main_check.csv, but is instead a voltage analysis of all signals inside all the netlist files.

1.3 Additional Output Files

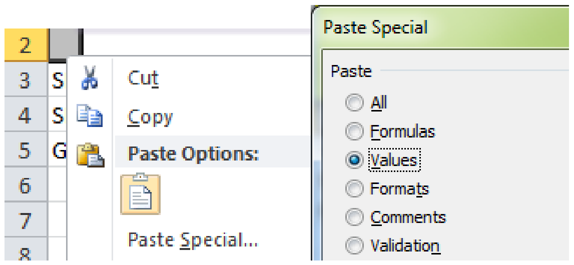
Currently, the script also outputs texts files which contain information on how signal names within a device on the schematic map to external signal names. The output is in text files and is specifically formatted if the DEVICEPARAM command has a DEVICETYPE category of XILINX_FPGA, ALTERA_FPGA, or LATTICE_CPLD. See Section 3.13 and 3.14 for more details.

1.4 System Analysis Spreadsheet

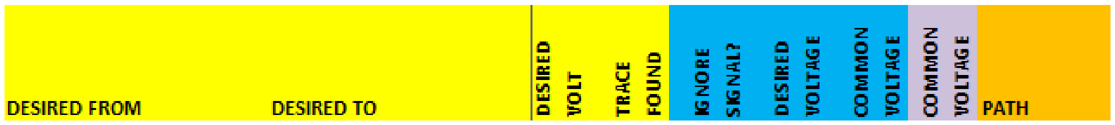
In order to view the output of the .csv files in a more readable manner, the conditionally formatted spreadsheet System_Analysis.xls was prepared. This spreadsheet contains three sheets within: Desired, All Nets, and Pinouts.

1.4.1 Desired

The user should paste the output of Main_check.csv to this sheet. This is done by copy pasting the entire Main_check.csv by pressing Ctrl+Shift+End and then doing a “Paste Special” and selecting Values in the top left column in the Desired Sheet as shown below:

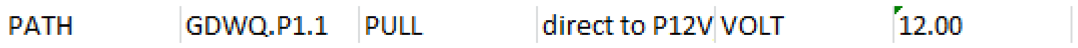


The Desired sheet has the following columns:



DESIRED FROM	DESIRED TO	DESIRED VOLT	TRACE FOUND	IGNORE SIGNAL?	DESIRED VOLTAGE	COMMON VOLTAGE	COMMON VOLTAGE	PATH

- Desired From/To are either ID.SIGNAL or ID.REF.PIN, for example Desired From being SEN2.SPI_CLK_C and Desired To being SEN2.P1.3.
- Desired Volt is a numerical voltage which is an optional parameter of the CHECKTRACE command
- Trace Found is a TRUE/FALSE flag depending on whether the specified connection in the CHECKTRACE command was found
- Ignore Signal is a TRUE/FALSE tag. It is FALSE by default. It is TRUE for signals associated with the IGNORE command. Also, any signals which are connected to the IGNORE signal through filtering devices or series resistors also have this flag as TRUE.
- Desired Voltage is a TRUE/FALSE flag. The flag is TRUE if the voltages found on the signals by the script are the same as Desired Volt, and FALSE otherwise. This is #NA if no Desired Voltage was specified.
- Common Voltage is a TRUE/FALSE flag. TRUE if the all the different voltage pulls found on a signal by the script are all equal, and FALSE otherwise. This is #NA if no voltage was found on the signals.
- Common Volt is a numerical voltage if all voltage pulls found on a signal are equal for a signal, and #NA otherwise.
- Path consists of three additional categories, and has an example shown below:
    - Path: This is a list of ID.REF.PIN which show the path the Desired From signal takes to get to Desired To
    - Pull: This is a list of where either the Desired From or Desired To signals are pulled and is of the format “direct to RAIL” or “ID.TYPE to RAIL” where RAIL is the rail name.
    - Volt: This is a list of the same size as the Pull list and corresponds to the numerical voltage associated each Pull list element respectively



PATH	GDWQ.P1.1	PULL	direct to P12V	VOLT	12.00

There is conditional formatting applied in this sheet to highlight the rows which have Trace Found and Common Voltage as False, and if Ignore Signal is True.

1.4.2 All Nets

If the VALUE of –v in Section 1.0 was 1, then the script will output a Volt_check.csv file. The output of this file will be pasted to the All Nets sheet similar to the Desired sheet.

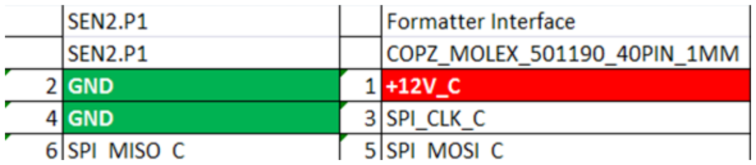
This sheet has the same columns as the Desired sheet. The only difference is that the Desired From and Desired To signals are the same and the Trace Found is #NA.

There is conditional formatting applied in this sheet to highlight the rows which have Trace Found and Common Voltage as False, and if Ignore Signal is True.

1.4.3 Pinouts

The output of Main_map.csv should be copy pasted to this sheet similar to how it was described in Section 1.4.1.

This sheet contains the pinouts of all devices and connectors specified from the MAP and DEVICEPIN command. There is conditional formatting applied to highlight certain signals such as GND, Voltage Rails, etc as shown below:



	SEN2.P1		Formatter Interface
	SEN2.P1		COPZ_MOLEX_501190_40PIN_1MM
2	GND	1	+12V_C
4	GND	3	SPI_CLK_C
6	SPI_MISO_C	5	SPI_MOSI_C

2.0 Setup

The user needs to have python installed on their computer. The following files should be contained within the same folder:

- system_connections.py and netlist.py
- Ascii netlist files for all schematics to be used in the script
- main.csv and all other .csv files that are used with the IMPORT command

It is recommended to keep System_Analysis.xls in the same folder as the rest of these files for convenience.

3.0 Commands

Commands are specified in any of the input .csv files. The commands are specified in the first column of the input .csv files and required parameters are contained the following cells.

3.1 COMMENT

COMMENT, VALUE

VALUE is any arbitrary string. When the COMMENT command is placed before the CHECKTRACE, MAP, and CHECKVOLT commands, it appears in the appropriate locations in the respective Main_check.csv and Main_map.csv files.

3.2 IMPORT

IMPORT, FILENAME

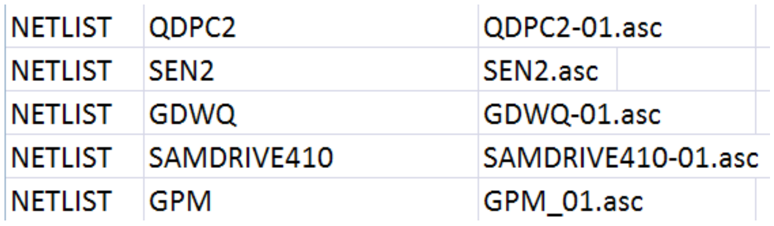
Imports other input .csv files for commands.

3.3 NETLIST

NETLIST, ID, FILENAME

Imports the netlist .asc file and associates the contents of that netlist file with the ID.

For example:



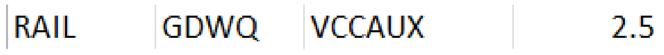
NETLIST	QDPC2	QDPC2-01.asc
NETLIST	SEN2	SEN2.asc
NETLIST	GDWQ	GDWQ-01.asc
NETLIST	SAMDRIVE410	SAMDRIVE410-01.asc
NETLIST	GPM	GPM_01.asc

3.4 RAIL

RAIL, ID, SIGNAL, VALUE

Specifies a signal name which represents a voltage rail in the specified board ID and the numerical value of that rail.

For example:



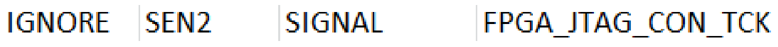
RAIL	GDWQ	VCCAUX	2.5

3.5 IGNORE

IGNORE, ID, SIGNAL/DEVICE, SIGNAL NAME/DEVICE TYPE

Specifies a signal to be ignored for voltage level checking or a device to be ignored when tracing signals for the CHECKTRACE command or passing through a device when tracing for voltage pulls.

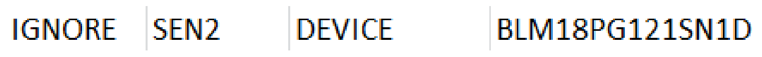
For example, if the command was specified for a signal as follows:



IGNORE	SEN2	SIGNAL	FPGA_JTAG_CON_TCK

The signal FPGA_JTAG_CON_TCK would not have any voltage level checks performed on it. Similarly, any signals which are connected to FPGA_JTAG_CON_TCK through connectors, series resistors, or pass through devices will also be ignored for volt checks as well.

As a device example, if the command was specified as:



IGNORE	SEN2	DEVICE	BLM18PG121SN1D

The device BLM18PG121SN1D would be ignored in the following scenarios:

- When the script tries to pass through the device if it is specified as a passthrough device through the DEVICELINK command
- When the device is a connector and links to a harness or other connectors
- When one of the pins on the device pulls the signal attached to it to another pin or voltage through the DEVICEPULL or DEVICEVOLT command

3.6 CHECKTRACE

CHECKTRACE, FROM ID, FROM SIGNAL/REF.PIN, TO ID, TO SIGNAL/REF.PIN [, GROUP [, VOLT]]

From/To ID are Netlist IDs eg. SEN2, QDPC2 etc.

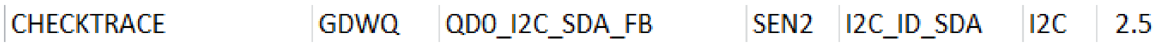
It can be checked whether a signal connects to a signal on the same board, different boards, or if a signal connects to a pin on a device on the same or different board, thus the From/To parameters are either SIGNAL such as I2C_SDA or REF.PIN such as J2.A11.

Group is an optional parameter which adds no purpose.

Volt is the desired voltage value which is evaluated as in the Desired Voltage flag in the System Analysis spreadsheet (Section 1.4.1).

The CHECKTRACE command generates output in the Main_check.csv file. See Section 1.4.1.

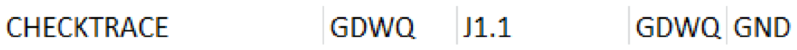
For example, if the I2C Data connection needs to be checked between the SEN2 and GDWQ interfaces, the following CHECKTRACE command is specified:



CHECKTRACE	GDWQ	QD0_I2C_SDA_FB	SEN2	I2C_ID_SDA	I2C	2.5

The From ID is GDWQ, the From Signal is QD0_I2C_SDA_FB, the To ID is SEN2, the To Signal is I2C_ID_SDA. For the optional parameters, the Group is I2C, and the Desired Voltage is 2.5V.

The following example specifies that there should be a GND connection on a certain pin on a connector.



CHECKTRACE	GDWQ	J1.1	GDWQ	GND

3.7 CHECKVOLT

CHECKVOLT, ID, SIGNAL [,GROUP [, VOLT]]

ID is a Netlist ID eg. SEN2.

Signal is the signal name in the schematic.

This command is when it is desired to perform a voltage analysis on a signal. Specifying the signal as part of the group, and specifying a desired voltage are optional parameters. This command generates output in the Main_check.csv file. See Section 1.4.1.

3.8 HARNESSLINK

HARNESSLINK, HARNESS_ID, REF1, PIN1, REF2, PIN2

HARNESS_ID is an arbitrary ID assigned to a harness.

REF1 is an arbitrary reference designator given to one end of a harness. PIN1 is a pin on this end of the harness. REF2 is an arbitrary reference designator assigned to the other end of the harness, and PIN2 is a pin on this end of the harness.

This command links two pins in a harness to specify the path a signal may take through two ends of a harness.

For example in the Mambo design, the SAS0 harness connected Pins A1-A18 on the QDPC2 side to Pins B1-B18 on the GDWQ side. Taking Pin A1/B1 as an example, and specifying the harness' reference designator on the QDPC2 end as P5 and the GDWQ end as P2, this would be specified through the HARNESSLINK command as follows:

HARNESSLINK	SAS0	P5	A1	P2	B1

Also see Section 3.9 for the related CONNECTION command.

3.9 CONNECTION

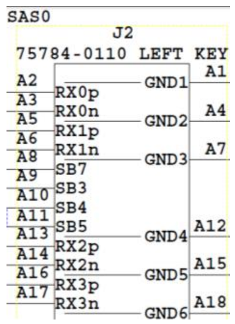
CONNECTION, FROM ID, FROM REF, TO ID, TO REF

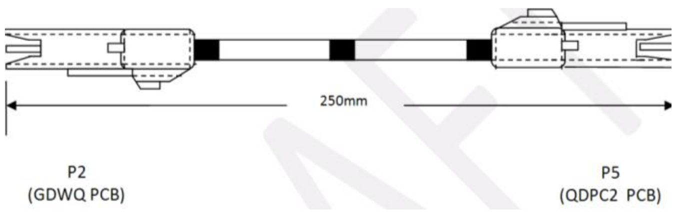
From/To ID are either Netlist IDs or Harness IDs. This allows connection between reference designator of a connector on a board to the reference designator on one end of a harness.

The SAS0 harness example will be continued from Section 3.8, where Pins A1-A18 on QDCP2 are connected to Pins B1-B18 on GDWQ through the SAS0 harness.

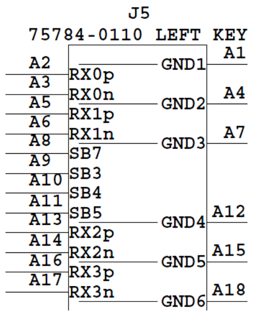
The HARNESSLINK command specifies the path a signal takes between two ends of a harness. The CONNECTION command links connectors to harnesses. Specifically for the SAS0 example, the connector J2 on GDWQ connects to Reference Designator P2 on the SAS0 harness. On the other end, the designator P5 on the SAS0 harness connects to connector J5 on the QDPC2 ID. This is shown in the images below:

GDWQ





QDPC2



The corresponding CONNECTION commands would be:

CONNECTION	GDWQ	J2	SAS0	P2
CONNECTION	SAS0	P5	QDPC2	J5

3.10 MAP

MAP, ID, REF, NAME

Specifies device pins to be mapped to Main_map.csv.

ID is a Netlist ID eg. SEN2. REF is a reference designator of a device. Name is an arbitrary string associated with the device map in the Main_map.csv file.

The following shows how to use the MAP command to display the pin out of connector P1 in the SEN2 board:

MAP	SEN2	P1	Formatter Interface

The following is an example of how one would cause all the pins of the SEN2 FPGA to appear in the Main_map.csv file:

MAP	SEN2	U32	SEN2 FPGA

Also see Section 3.11 for the DEVICEPIN command.

3.11 DEVICEPIN

1. DEVICEPIN, TYPE, RC, ROWS, COLS, DIR

2. DEVICEPIN, TYPE, ARB, ROW, COL, PIN

3. DEVICEPIN, TYPE, CATEGORY, NAME, PIN1, PIN2, …

The DEVICEPIN command allows the output of the pins of devices to be displayed in a certain orientation or manner. If a device is used with the MAP command, then the script checks to see whether there is additional orientation parameters provided for that device through DEVICEPIN. If there is not, the Main_map.csv file merely just lists all the pins and their corresponding signal names vertically, and if there are, then the output depends on what parameters are used for the DEVICEPIN command.

The two ways for specifying device pinouts for connectors are through the choice of parameters in 1 and 2. The 3rd set of parameters are for specifying pinouts for both connectors and devices.

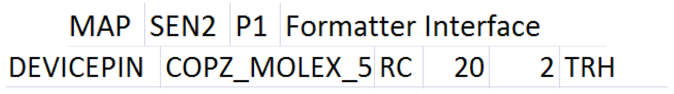
3.11.1 RC Method

This method is only for connectors. For 1, RC is chosen as the 3rd parameter, and ROWS and COLS are the number of desired rows and columns which should be displayed in the Main_map.csv file for that specific connector.

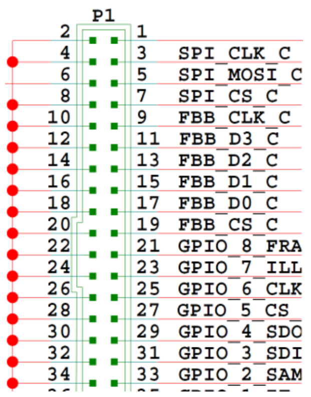
DIR is either TRH, TRV, TLH, TLF. This specifies where Pin 1 is located and whether the pins increase horizontally or vertically. For example, TRH means that Pin 1 starts in the top-right and increases horizontally such as:

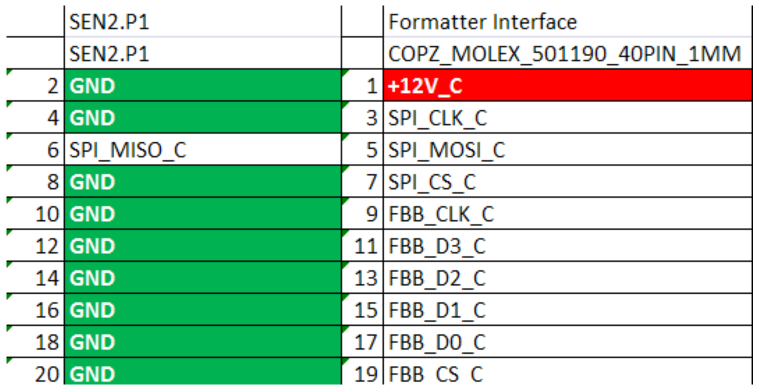
2.	1.
4.	3.

For example, for the SEN2 and GDWQ interface, the connector is 20 rows and 2 columns, and Pin 1 is TRH. Thus, the output of this connector would be displayed through the following commands:



MAP	SEN2	P1	Formatter Interface		
DEVICEPIN	COPZ_MOLEX_5	RC	20	2	TRH





	SEN2.P1		Formatter Interface
	SEN2.P1		COPZ_MOLEX_501190_40PIN_1MM
2	GND	1	+12V_C
4	GND	3	SPI_CLK_C
6	SPI_MISO_C	5	SPI_MOSI_C
8	GND	7	SPI_CS_C
10	GND	9	FBB_CLK_C
12	GND	11	FBB_D3_C
14	GND	13	FBB_D2_C
16	GND	15	FBB_D1_C
18	GND	17	FBB_D0_C
20	GND	19	FBB_CS_C

3.11.2 ARB Method

This method is only for connectors. For 2, ARB is chosen as the third parameter as specified which row and column a certain pin for the connector should be displayed in. For example, for the Ethernet connector in QDPC2, the pins are specified through the following DEVICEPIN and MAP command which results in an output of:

MAP	QDPC2	J8	Ethernet 1 (ArrayLOC)		
DEVICEPIN	RJ-45	ARB	1	1	1
DEVICEPIN	RJ-45	ARB	2	1	4
DEVICEPIN	RJ-45	ARB	3	1	2
DEVICEPIN	RJ-45	ARB	4	1	3
DEVICEPIN	RJ-45	ARB	5	1	5
DEVICEPIN	RJ-45	ARB	6	1	6
DEVICEPIN	RJ-45	ARB	7	1	9
DEVICEPIN	RJ-45	ARB	8	1	10
DEVICEPIN	RJ-45	ARB	9	1	12
DEVICEPIN	RJ-45	ARB	10	1	11
DEVICEPIN	RJ-45	ARB	11	1	8

QDPC2.J8	Ethernet 1 (ArrayLOC)		
QDPC2.J8	RJ-45		
		1	ETHN1_TX_P
		4	ENET1_VDD_RXTX
		2	ETHN1_TX_N
		3	ETHN1_RX_P
		5	ENET1_VDD_RXTX
		6	ETHN1_RX_N
		9	+3V3_SB
		10	ETHN1_LEDL_K
		12	+3V3_SB
		11	ETHN1_LEDR_K
		8	GND

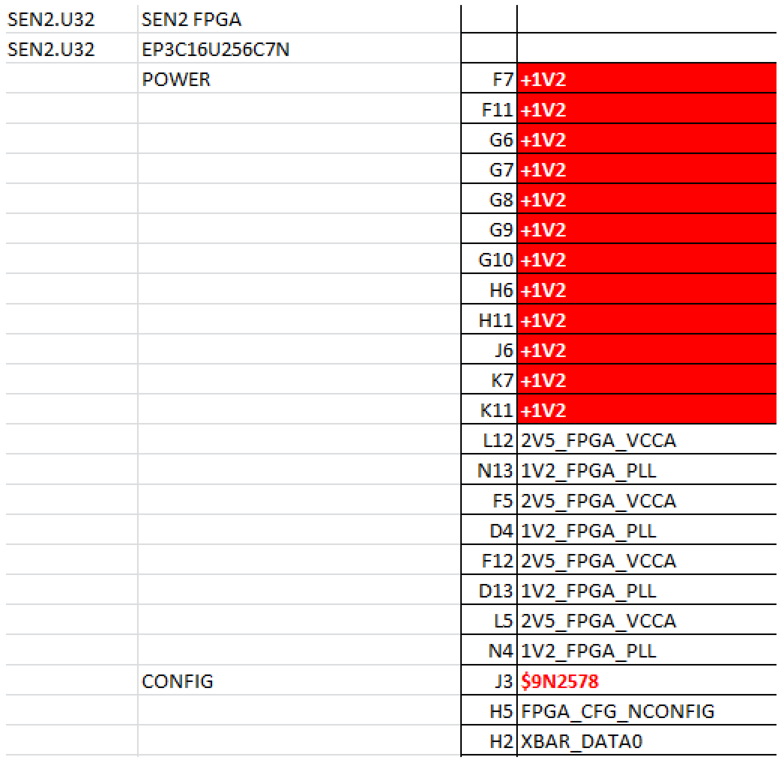
3.11.3 CATEGORY Method

This method is either for connectors or devices. If CATEGORY is the third parameter, then NAME is an arbitrary string which defines the category, followed by a set of pins which will appear under this category for the output of this connector in Main_map.csv. For example, if it is desired to categorize the SEN2 FPGA pins by banks, the following command would be specified:

DEVICEPIN	EP3C16U256C7N	CATEGORY	BANK8	C4	C7	D8	E7	A5	C6	A2
DEVICEPIN	EP3C16U256C7N	CATEGORY	BANK7	C10	C13	C11	A13	B9	A9	A16
DEVICEPIN	EP3C16U256C7N	CATEGORY	BANK1	E3	G3	G2	F2	D1	G5	C2
DEVICEPIN	EP3C16U256C7N	CATEGORY	BANK6	G14	F14	F13	E15	E16	E14	H16
DEVICEPIN	EP3C16U256C7N	CATEGORY	BANK2	K3	M3	L3	K2	M2	M1	K5
DEVICEPIN	EP3C16U256C7N	CATEGORY	BANK5	M14	L15	L14	M15	M16	N14	P15
DEVICEPIN	EP3C16U256C7N	CATEGORY	BANK3	P4	P7	T2	L7	L8	T1	P6
DEVICEPIN	EP3C16U256C7N	CATEGORY	BANK4	P13	T16	P11	N12	R9	T9	M10
DEVICEPIN	EP3C16U256C7N	CATEGORY	CONFIG	J3	H5	H2	G12	H12	H3	H13
DEVICEPIN	EP3C16U256C7N	CATEGORY	POWER	F7	F11	G6	G7	G8	G9	G10

MAP	SEN2	U32	SEN2 FPGA

This would result in the following output in the Main_map.csv file:



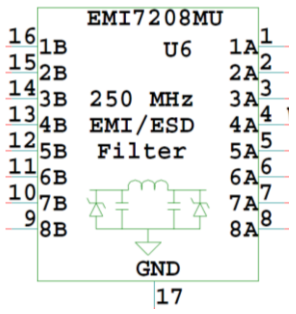
SEN2.U32	SEN2 FPGA		
SEN2.U32	EP3C16U256C7N		
	POWER	F7	+1V2
		F11	+1V2
		G6	+1V2
		G7	+1V2
		G8	+1V2
		G9	+1V2
		G10	+1V2
		H6	+1V2
		H11	+1V2
		J6	+1V2
		K7	+1V2
		K11	+1V2
		L12	2V5_FPGA_VCCA
		N13	1V2_FPGA_PLL
		F5	2V5_FPGA_VCCA
		D4	1V2_FPGA_PLL
		F12	2V5_FPGA_VCCA
		D13	1V2_FPGA_PLL
		L5	2V5_FPGA_VCCA
		N4	1V2_FPGA_PLL
	CONFIG	J3	$9N2578
		H5	FPGA_CFG_NCONFIG
		H2	XBAR_DATA0

3.12 DEVICELINK

DEVICELINK, TYPE, PIN, PIN [,BDIR]

This command specifies how a device links two pins. A common usage of this command is on filtering devices. It allows the tracing algorithm in the script to bypass such devices.

Any value other than an empty string for BDIR would count as a true flag, specifying a bidirectional connection between the two pins. For example, the filtering device shown would have one set of its pins linked to each other as shown below:



DEVICELINK	EMI7208MU	16	1	Yes

3.13 REFSIG

REFSIG, ID, REF, PIN, INT SIGNAL, EXT SIGNAL [,IOSTANDARD]

Specifies a signal name which should appear on a REF.PIN of a device in the specified board ID, and how that signal name translates to an external name. IOSTANDARD can optionally be specified.

INT SIGNAL, EXT SIGNAL, and IOSTANDARD are arbitrary strings.

This command needs to be used together with the DEVICEPARAM command specified in Section 3.14. If the DEVICETYPE in DEVICEPARAM is specified, then the REFSIG command generates an output text file with the name ID_REF_DEVICETYPE.txt.

The main purpose of this command is to generate output text files which serve as partial constraint files used for FPGA pin settings. For example, the signal V6_JTAG_TMS in located on Pin AF8 of the QDPC2 FPGA may be called FPGA_V6_JTAG_TMS inside the constraint file for the FPGA. Additionally, the constraint file may need to specify this signal as LVTTL. Thus, the REFSIG command would be specified as:

REFSIG	QDPC2	U43	AF8	V6_JTAG_TMS	FPGA_V6_JTAG_TMS	LVTTL

For this signal relation to appear in an output text file, the DEVICEPARAM, DEVICETYPE command would also need to be specified as:

DEVICEPARAM	UXC6VLX130T-3FFG1156C	DEVICETYPE	XILINX_FPGA

U43 would then be of DEVICETPYE XILINX_FPGA and the output file generated due to the REFSIG command would be called QDCP2_U43_XILINX_FPGA.txt. From the specified signal translation in the REFSIG command earlier, the output of this text file would include:

NET "FPGA_V6_JTAG_TMS" IOSTANDARD = LVTTL;
NET "FPGA_V6_JTAG_TMS" LOC = AF8;

Formatted according to Xilinx FPGA constraint files.

See DEVICEPARAM in Section 3.14 for supported parameters.

3.14 DEVICEPARAM

DEVICEPARAM, TYPE, DEVICETYPE, (ATERA_FPGA, XILINX_FPGA, LATTICE_CPLD) [, OTHER_PARAM, OTHER_PARAM_VALUE] , …

DEVICEPARAM is used by the developers of the ICD Script to be able to specify additional information about devices which cannot be obtained from netlist files. Currently, the only supported parameter in DEVICEPARAM is DEVICETYPE which can only have one of three values: ALTERA_FPGA, XILINX_FPGA, or LATTICE_CPLD.

In the future, other parameters and corresponding values may be allowed if needed.

When specifying if the DEVICETYPE is either XILINX_FPGA, ALTERA_FPGA, or LATTICE_CPLD, the REFSIG command knows how to format the output constraint text file. Currently, this is the only use of this command and future uses may be added.

As an example, the Xilinx FPGA on the QDPC2 and GDWQ may be specified as follows:

DEVICEPARAM	UXC6VLX130T-3FFG1156C	DEVICETYPE	XILINX_FPGA

3.15 DEVICEPULL

1. DEVICEPULL, TYPE, DIR, PIN A1, PIN B1, PIN B2, PIN B3, …

2. DEVICEPULL, TYPE, # PIN A, DIR, PIN A1,…,PIN A#, PIN B1, PIN B2, …

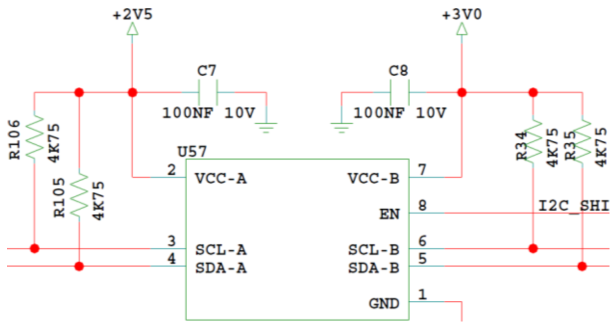
This command links the voltage between pins on a device.

The DIR parameter is either:

BA: PIN B1, B2, … all connect to PIN A1 [, A2, …] for voltage
ABBA: The set of PIN Bs and PIN As all have bidirectional connection

3.15.1 Single Pin A

For the first set of parameter, there is a single pin which a set of Pin Bs are related to. For example, Pins 3 and 4 on the following I2C translator device are connected to Pin 2 for voltage:



The command is:

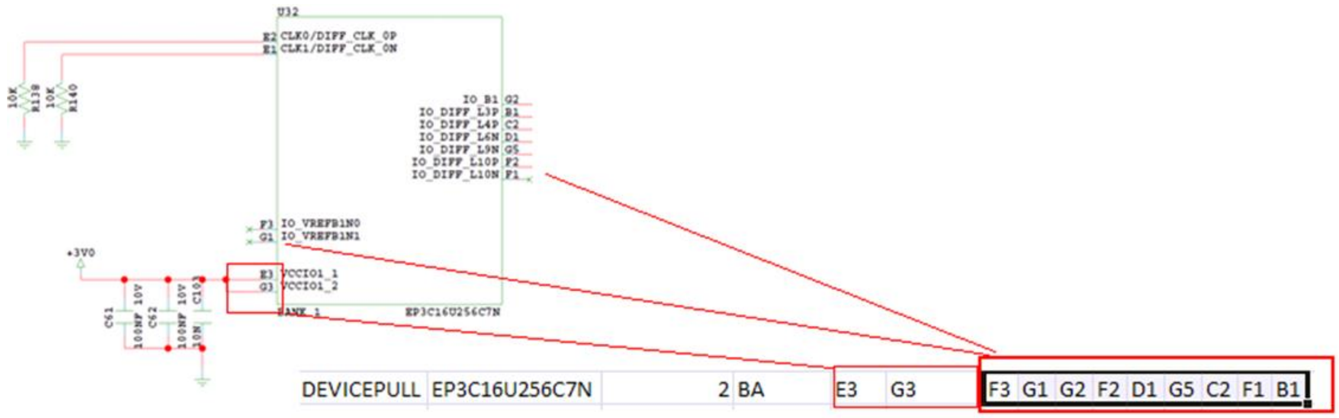
DEVICEPULL	UPCA9517	BA	2	3	4

3.15.2 Multiple Pin A

For the second set of parameters, there are a set of Pin As which a set of Pin Bs may be related to.

# PIN A is a numerical integer specifying the number of reference pins for voltage. After the DIR parameter, all the reference pins must be specified. Thereafter, a set of pin Bs need to be specified which are linked to the set of Pin As for voltage.

For example, on the following FPGA bank, the set of Pin A as reference voltage pins is [E3, G3], which are followed by a set of Pin B [F3, G1, G2, F2, D1, G5, C2, F1, B1] through the command:



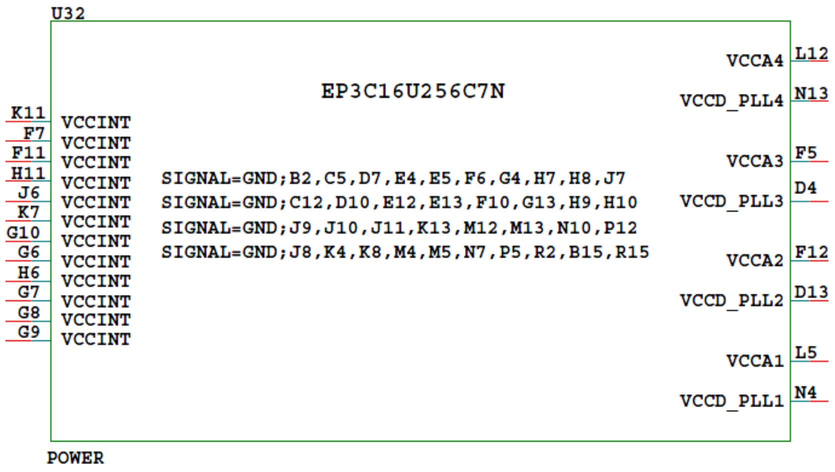
DEVICEPULL	EP3C16U256C7N	2	BA	E3	G3	F3	G1	G2	F2	D1	G5	C2	F1	B1

3.16 DEVICEVOLT

DEVICEVOLT, TYPE, VOLT, PIN1, PIN2, …

Specifies a fixed voltage on a set of pins on a device. For example, some of the GND pins on of the SEN2 FPGA as specified as:

DEVICEVOLT	EP3C16U256C7N	0	B2	C5	D7	E4	E5	F6	G4



U32
EP3C16U256C7N
K11 VCCINT
F7 VCCINT
F11 VCCINT
H11 VCCINT
J6 VCCINT
K7 VCCINT
G10 VCCINT
G6 VCCINT
H6 VCCINT
G7 VCCINT
G8 VCCINT
G9 VCCINT
SIGNAL=GND;B2,C5,D7,E4,E5,F6,G4,H7,H8,J7
SIGNAL=GND;C12,D10,E12,E13,F10,G13,H9,H10
SIGNAL=GND;J9,J10,J11,K13,M12,M13,N10,P12
SIGNAL=GND;J8,K4,K8,M4,M5,N7,P5,R2,B15,R15
VCCA4 L12
VCCD_PLL4 N13
VCCA3 F5
VCCD_PLL3 D4
VCCA2 F12
VCCD_PLL2 D13
VCCA1 L5
VCCD_PLL1 N4
POWER