Design and Analysis of a Visual Programming Language for Microcontroller Systems

Jeng Howe Kong

Department of Electrical and Electronic Engineering
The University of Adelaide

Submitted for M.Eng.Sc on 22 December 2003

Supervisor
Michael Liebelt
Abstract

Conventional methods of programming microcontrollers using textual languages are hard to learn and are daunting to novice programmers seeking to learn microcontroller programming. Visual languages have always been regarded as a useful tool in helping non-programmers and novice-programmers to write programs. However there have been limited attempts at creating a visual language for microcontroller systems and there isn’t a visual language for microcontrollers that is flexible and easy to use. This thesis presents a way of addressing the issue by creating a low level visual programming language for microcontroller systems. The low level visual programming language aims to alleviate the problem by using a fine grained language to improved flexibility and providing an integrated visual language environment in which users can focus on writing programs and solving problems.

A visual language environment called CoreChart was developed for this purpose. CoreChart aims to simplify the process of programming microcontrollers by providing users with a tool to construct assembly programs visually. The visual language will utilize flow chart diagramming techniques to present users with a more meaningful view of the program. This allows users to focus on writing programs to solve problems, rather than on the rules and syntax of the language. The procedure of programming microcontrollers is further simplified by automating the task of compiling the program and downloading the program into the microcontroller. A survey was conducted on university and high school students to evaluate the effectiveness of CoreChart.
# Table of Contents

ABSTRACT .............................................................................................................. I

TABLE OF CONTENTS ........................................................................................ II

LIST OF FIGURES .................................................................................................. V

ACKNOWLEDGEMENTS ........................................................................................ VII

1 INTRODUCTION .................................................................................................. 1

1.1 BACKGROUND ................................................................................................. 1

1.2 PROGRAMMING MICROCONTROLLERS ....................................................... 1

1.3 PROGRAMMING LANGUAGES ........................................................................ 2

1.3.1 High Level Language Translators ............................................................... 2

1.3.1.1 Interpreters ............................................................................................ 2

1.3.1.2 Compilers ............................................................................................. 3

1.3.2 Types of Programming Languages ............................................................ 3

1.4 VISUAL LANGUAGE FOR EMBEDDED SYSTEMS ....................................... 5

1.5 TARGET MICROCONTROLLER SYSTEM ..................................................... 6

1.6 ARCHITECTURE OF THE PIC MICROCONTROLLER ................................. 7

1.6.1 Harvard Architecture ................................................................................. 7

1.6.2 Two-Stage Pipeline ................................................................................... 8

1.6.3 Reduced Instruction Set ............................................................................ 8

1.6.4 Limitations of the architecture ................................................................. 8

2 VISUAL PROGRAMMING LANGUAGES (VPL) .................................................. 11

2.1 DEFINITIONS AND INTRODUCTION ........................................................... 11

2.1.1 Problem Domain ....................................................................................... 11

2.1.2 Granularity ............................................................................................... 11

2.1.3 Language Levels .................................................................................... 12

2.2 THE BENEFITS OF VISUAL PROGRAMMING ........................................ 13

2.3 THE USES OF VISUAL LANGUAGES ........................................................ 14

2.4 PROBLEMS ASSOCIATED WITH VISUAL LANGUAGES ............................ 15
3 DESIGN CONSIDERATIONS ................................................................. 17
3.1 INTRODUCTION .............................................................................. 17
3.2 SYNTAX OF VPL ............................................................................. 18
3.3 USAGE OF AN APPROPRIATE METAPHOR ...................................... 19
3.4 IMPLEMENTATION OF VPL .............................................................. 20
3.5 VISUAL LANGUAGE ENVIRONMENT .............................................. 23
   3.5.1 Visual Language Editor ............................................................. 23
   3.5.2 Visual program execution ......................................................... 24
4 CORECHART IDE .............................................................................. 25
4.1 OVERVIEW ...................................................................................... 25
4.2 IMPLEMENTATION .......................................................................... 26
4.3 MAIN WINDOW ................................................................................ 28
   4.3.1 Program workspace ................................................................. 29
   4.3.2 Navigation tools ....................................................................... 29
   4.3.3 Editing tools ............................................................................. 29
   4.3.4 Icon palette .............................................................................. 29
   4.3.5 Icon properties ......................................................................... 30
   4.3.6 Main menu bar ........................................................................ 30
       4.3.6.1 File main menu bar item ...................................................... 31
       4.3.6.2 Edit main menu bar item ................................................... 31
       4.3.6.3 Options main menu bar item ............................................. 31
       4.3.6.4 Help main menu bar item ............................................... 32
4.4 DIALOG WINDOWS .......................................................................... 33
   4.4.1 Subroutines Dialog ................................................................. 33
   4.4.2 Variables Dialog ....................................................................... 34
   4.4.3 Constants Dialog ..................................................................... 35
   4.4.4 Labels Dialog .......................................................................... 35
   4.4.5 Import Subroutine Dialog ....................................................... 36
   4.4.6 Export Subroutine Dialog ....................................................... 37
List of Figures

1.1  Simplified block diagram of a PIC microcontroller
1.2  General format for 14-bit instructions
1.3  Addressing data memory using bank select register
1.4  Program memory organization
2.1  Language Levels
3.1  A typical flowchart diagram
3.2  Flowchart symbols and its relative meaning
3.3  Mid-Range PIC Family instruction set
4.1  CoreChart Main Window
4.2  Main menu bar with all its dropdown submenus
4.3  Subroutines Dialog
4.4  Variables Dialog
4.5  Constants Dialog
4.6  Labels Dialog
4.7  Import Subroutine Dialog
4.8  Export Subroutine Dialog
4.9  Chip Configuration Dialog
4.10  Hardware Settings Dialog
4.11  Program Preferences Dialog
4.12  Chip Model Selection Dialog
4.13  ASM Viewer and Assembly Status Dialog
4.14  CoreChart Icons
4.15  Comparison of a CoreChart Program vs. a flowchart diagram
4.16  Assembly program equivalent of the CoreChart Program
5.1  Time taken to complete practical
Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University or other tertiary institution and, to the best of the author’s knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

The author hereby consents to this thesis, when deposited in the University Library, being made available for loan and photocopying.

Kong Jeng Howe
Acknowledgements

I would like to thank my supervisor Mr. Michael Liebelt for his continued guidance and support throughout the course of this research. I must also thank the staff of the electrical and electronics engineering department in Adelaide University and the staff of eLabtronics for their support. In particular, Miroslav Kostecki and Peng Choo from eLabtronics for their help in technical details and in obtaining high school survey candidates.

Finally, I thank my family and friends for keeping my confidence and enthusiasm high during the time it took to complete this research.
1 Introduction

1.1 Background
Microcontrollers have been extensively used throughout the human society in an attempt to improve our way of life. Due to the decreasing price of electronic components, even simple consumer products can include a microcontroller to realize particular features. This has driven up the competition between equipment manufacturers to differentiate their products from others by providing enhanced functionality while maintaining or reducing cost. To address these requirements, manufacturers use integrated circuit-based embedded control systems. These embedded systems enable manufacturers to differentiate their products, replace less efficient electromechanical control devices, add product functionality, and significantly reduce product costs [1]. Each embedded system has a software part running on the microcontroller and a hardware part consisting of mechanical and electronic components. One of the common examples is the microwave oven with a digital display. In this example, a microcontroller is used to process user inputs from the touch buttons on the front panel of the microwave, then display a response on the LCD screen and execute the relevant operation. Other appliances with embedded microcontrollers include televisions, fridges, hi-fi systems, remote controls etc.

1.2 Programming microcontrollers
Microcontrollers in embedded systems are essentially a slimmed down computer with built in program and data memory, input/output ports, and some additional peripherals. As with all computer systems, they require software to operate. The software that the embedded system understands is machine code. The machine code is a low level programming language that is heavily influenced by the basic architecture of the microcontroller [2]. Machine codes are normally phrased in such a way that facilitates the translation and execution of the code by the microcontroller, which is very different from the languages human use. This makes it difficult for human to understand the code, and it is even more difficult to write a program using this language.

Assembly language is a rudimentary attempt to display the machine code in a more human readable form. Each assembly language statement corresponds to one machine
language statement. Nevertheless, programming microcontrollers using assembly language is still a time consuming task. Novice programmers who wish to learn microcontroller programming face a steep learning curve as they first need to familiarize themselves with the cryptic syntax of the assembly instruction set used by microcontrollers.

1.3 Programming Languages

Higher level programming languages which more closely resemble human spoken language were introduced in an attempt to simplify the task of programming. This attempt has proved successful and has enabled widespread proliferation of microcontrollers into our society.

A language translator is normally used to translate the higher level programming language into low level assembly language or machine code. High-level languages have a number of advantages over machine code and assembly language. Mainly, they allow the programmer to use a notation that is more in tune with problem solving; thus, allowing the programmer to concentrate more on solving problems. Programming languages can also be classed into textual programming languages and visual programming languages. Currently, textual programming languages have been extensively used in programming embedded systems through out the industry and at home by amateurs.

1.3.1 High Level Language Translators

These higher level languages rely on a translator to convert the language into machine readable code. There are generally two types of translators: interpreters and compilers [3].

1.3.1.1 Interpreters

An interpreter is a higher level language translator that converts high-level languages to machine code. The interpreter itself is a program that resides in the computer. It executes a higher level language program by reading each language statement one at a time, translating it and executing the resulting machine code. The advantage of using interpreters is that programs can be written interactively. Codes can be tested out immediately and be built upon step by step until the final product is achieved. The
downside to this is that a portion of the processor cycle is used for the real time translation of the language into machine code. There is also no optimization routine built into the interpreter. Every line is translated before executing, no matter how many times it has done this before. This degrades the performance of the system. Furthermore, an interpreter is a piece of software that resides in the program memory. This increases the total memory requirement of the embedded system.

1.3.1.2 Compilers
A compiler is a high-level language translator that relies on a host machine (like a desktop PC) to translate a high-level language to machine code. The machine code is then downloaded to the microcontroller program memory. This way, the microcontroller executes the program directly in machine code in full speed without a need for translation. Unlike interpreters, programs cannot be written interactively.

1.3.2 Types of Programming Languages
C and BASIC are two of the most common textual programming languages used to program microcontrollers [4]. They have the advantage of using a more human comprehensible code than assembly language or machine language, being easier to master, and allowing construction of more complex algorithms in programs more easily. BASIC, which stands for Beginners All Purpose Symbolic Code is typically used by amateurs and novice programmers due to its easy to learn syntax and its comprehensive set of function libraries. However, BASIC also has the reputation of being slow and inefficient. This may be the price for improved usability. C is more commonly used by the industry as it is more flexible and efficient, but C uses a more obscure syntax and many people find it difficult to learn and master. FORTH is a stack based programming language which is less commonly used as it is much harder to learn and master. However, it is a very efficient language and can be easily translated to machine code. This makes it a suitable language for interpreters as it requires less processor cycles to translate to machine code.

Nevertheless, every programming language still requires time to learn and may prove daunting to some beginners. Furthermore, the efficiency of the generated machine code heavily depends on the efficiency of the translator. This may be important when dealing with large and complex programs where the hardware limits of the microcontroller
dictate the maximum size of the program and its ability to process data at an acceptable speed. Finally, programmers will face difficulty when writing timing critical applications where one is required to get a precise count of program instruction cycles for some real time control applications.

Visual Programming Languages (VPL) express programs in a visual form using graphics and arrows. Visual Programming Languages often utilize some form of flow diagram to represent the instruction flow in a program, or the flow of data through a system. Visual Programming Languages are relatively new in embedded systems programming and there has not been any widely known visual programming language for embedded microcontroller systems. One limited application example of visual programming languages on embedded microcontroller systems is the Lego Programmable Brick [5], developed by the MIT Media Laboratory in conjunction with Lego. The Programmable Brick uses an 8-bit 68HC11 microcontroller embedded in a Lego brick [2], with six input ports for attachment to sensors and four output ports to drive devices like motors, buzzers or lights. A Visual Language called LogoBlocks is used to program the Programmable Brick. It is a graphical variant of the popular Logo, a beginners programming language. LogoBlocks allows children to construct a program by pushing blocks which represent pieces of a program around the screen [6]. This system has proven to be very robust and easy to learn for novice programmers and children as it eliminates syntax problems. However, LogoBlocks is a highly specialized visual programming language written specifically for the Programmable Brick and is targeted for use by children. This limits the usability of the visual language as a more professional programming tool.
1.4 Visual Language for Embedded Systems

Currently, there is a lack of a proper visual programming language for embedded microcontroller systems. Most of the programming tasks in microcontrollers are accomplished by textual programming languages. Nearly all textual languages have a complex set of rules and syntax that define the language itself. Novice programmers often find the rules and syntax of a language confusing and difficult to learn.

This thesis aims to produce a functional low level Visual Programming Language for microcontroller systems. The Visual Programming Language will allow the programmers to focus on the goal of problem solving, rather than meddling with the rules and syntax of the language itself. It accomplishes this goal by using graphical symbols or icons to replace the textual syntax used in assembly language.

The VPL will also need to address several issues. First, it must be easy to learn. This requires the language to have a user friendly visual language editor and the graphical symbols used to represent the program must be easily comprehensible. Second, the VPL has to be flexible enough to implement all of the functions of the microcontroller to allow for more advanced programming by the user. Third, the language needs to have some form of portability across different microcontroller models that share the same instruction set. This is essential because microcontroller manufacturers tend to release newer versions and have variations of their microcontrollers to enhance its functionality and cater for the diverse needs of the market. Most of the different variations of the microcontrollers use identical architectures and instruction sets, but have different memory sizes and peripherals. The VPL should be able to generate code for different microcontroller models that use the same architecture and instruction set. Fourth, the language must be efficient to be of practical use, as microcontrollers tend to have limited memory and processing power.

This thesis will investigate the practical application of the designed VPL by conducting an experiment to compare the effectiveness of the language against other methods of programming.
1.5 Target microcontroller system

A target microcontroller system needs to be chosen as microcontroller architectures vary from one manufacture to another and also between different microcontroller families from a manufacturer. Furthermore, different microcontrollers have diverse features and capabilities. It would be time consuming to create a generic visual programming language that caters for all types of microcontrollers since each microcontroller type would need a translator written specifically for it. Hence, for this thesis, a particular microcontroller family was chosen such that the focus of the research can be placed on design of the visual programming language. Selection of the target microcontroller was based on the following criteria: popularity, cost, availability, case of use, and features/functions.

Popularity of the microcontroller is determined by the market share. This is again divided into the market share between types of microcontrollers manufactured by different manufacturers and the different architecture types. Some of the major microcontroller manufacturers include Motorola, Microchip, NEC, Hitachi etc. and they produce microcontrollers with architectures ranging from 4-bit to 32-bit. 4-bit microcontrollers are inexpensive but lack the minimum performance and features required for many applications; 16 and 32-bit microcontrollers are more powerful but relatively more expensive.

The mid-range 8-bit PIC microcontroller manufactured by Microchip was selected for its wide availability, low cost and adequate performance. There are many different mid-range PIC microcontroller models available, each with different memory size and peripherals. However, the different models share the same architecture and use the same instruction set. Hence, switching from one microcontroller to another only involves change in memory size and peripherals. This enables the creation of a VPL that can target all mid-range PIC microcontrollers with no change to the language itself.
1.6 Architecture of the PIC microcontroller

A deeper understanding of the microcontroller is necessary as the VPL will be designed in such a way that the user is able to access all of the microcontroller’s features and functions. This section contains a brief introduction on the architecture of the mid-range PIC microcontroller.

1.6.1 Harvard Architecture

Figure 1.1: Simplified block diagram of a PIC microcontroller

The PIC microcontrollers use Harvard architecture in which the program memory and data memory are separate memories accessed from separate buses. This allows for program memory and data memory to be sized differently. To execute an instruction,

---

1 Information obtained from PICmicro MID-RANGE MCU FAMILY DataSheet [26]
the instruction and data is fetched in a single cycle. While the program memory is being accessed, the data memory is on an independent bus and can be read and written. These separated buses allow one instruction to execute while the next instruction is fetched. A simplified block diagram of the PIC microcontroller is shown in Figure 1.1. Harvard architecture allows for instructions and data to be sized differently, making it possible to have instructions encoded as single words. In the mid-range PIC microcontroller, single word instructions are 14-bits wide. The program memory access bus fetches instructions in a single cycle and these are executed in the next cycle. In a typical Von Neumann architecture, most instructions are multi-byte. In general, an instruction will need to be fetched from memory using multiple cycles.

1.6.2 Two-Stage Pipeline

The PIC microcontroller has a two-stage instruction pipeline which overlaps the fetch and execution of instructions. The fetch of the instruction takes one cycle, while the execution takes another cycle. However, due to the overlap of the fetch of current instruction and execution of the previous instruction, an instruction is fetched and another instruction is executed every single cycle. Hence most instructions are executed in a single cycle except for two branch instructions which each take two cycles.

1.6.3 Reduced Instruction Set

The PIC microcontroller uses a reduced instruction set architecture which only contains 35 instructions. This is achieved by using symmetric instructions where it is possible to carry out any operation on any register using any addressing mode. This symmetrical nature allows for simple programming.

1.6.4 Limitations of the architecture

The architecture used by mid-range PIC microcontrollers does have some limitations. This is due to the limited 14-bit wide instruction word. As indicated in Figure 1.2, the instruction opcode typically takes up 3-6 bits, this leaves 8-11 bits available for data and memory address. This sets an upper limit on the addressable data memory to 128 locations (indicated by the file register address) and program memory to 2048 locations (by the CALL and GOTO instructions). This issue is addressable by
organizing the data memory into different banks and using a bank select register to switch between active banks. The program memory is also organized into pages and a register (PCLATH) is used to address the upper 5-bits of the 13-bit program counter. This introduces additional complexity when writing programs for the microcontroller and must be considered when designing the visual language. Only higher end models of the mid-range PIC microcontroller family have more than 128 bytes data memory and more than 2048 lines of program memory.

<table>
<thead>
<tr>
<th>Byte-oriented file register operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
</tr>
<tr>
<td>d = 0 for destination W</td>
</tr>
<tr>
<td>d = 1 for destination f</td>
</tr>
<tr>
<td>f = 7-bit file register address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit-oriented file register operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
</tr>
<tr>
<td>b = 3-bit bit address</td>
</tr>
<tr>
<td>f = 7-bit file register address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Literal and control operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
</tr>
<tr>
<td>OPCODE</td>
</tr>
<tr>
<td>k = 8-bit immediate value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CALL and GOTO instructions only</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
</tr>
<tr>
<td>k = 11-bit immediate value</td>
</tr>
</tbody>
</table>

Figure 1.2: General format for 14-bit instructions [26]
Figure 1.3: Addressing data memory using bank select register [26]

Figure 1.4: Program memory organization [26]
2 Visual Programming Languages (VPL)

2.1 Definitions and Introduction

There have been many diverse definitions of visual programming languages as different people adopted the term to describe their own concepts. For example, in recent years, the term ‘Visual Language’ is popularly used to describe a conventional textual language which has a WYSIWYG (What You See Is What You Get) Graphical User Interface design tool attached. Examples of this kind of language would be Borland Delphi, Microsoft Visual Basic & Visual C++. A more proper definition of VPL is a language which uses some graphical representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional textual programming language [8]. This definition does not impose restrictions on the type of data or information. Rather, the only limitation is that the language itself must have some meaningful visual representations as a means of programming. Programs written using VPLs are constructed in the traditional sense of programming [9]. However, the language primitives are: icons, lines, arrows, boxes, etc. In the conventional sense, these languages also have well defined syntax and semantics.

2.1.1 Problem Domain

An important issue in the visual language area is that of the size of a visual language’s problem domain. Many visual languages have been written with a particular problem in mind, for example, many so-called “FSA (Finite State Automata) languages” have been written to target a specific problem. On the other hand some other visual languages have been created that try to be capable of solving much more general programming problems. One such example is Prograph and Prograph CPX [10]. The general consensus in the visual language community is that visual languages will be most successful in the case of special purpose languages [11] where the target audience and/or the target platform is well defined.

2.1.2 Granularity

In some ways related to the idea of problem domain is the idea of the granularity of a visual language. In a fine grained visual language, the elements of the language represent a small number of steps on the underlying computer or interpreter. In a coarse
grained visual language, on the other hand, a single element of the language can represent a large number of steps on the underlying computer or interpreter. In a fine grained language, programmers can have more control over the underlying hardware the code is supposed to run on and it provides more flexibility to the programmer. The downside is it requires more programming elements to accomplish the task. This increases the complexity of the program. For a coarse grained language, fewer program elements are needed for the same task. However, this also reduces the control the programmer has on the hardware and also the programming language is less flexible.

2.1.3 Language Levels
Just as textual programming languages can be classified according to their level of abstraction, visual languages can be classified as well. The concept of abstraction, in terms of computer languages can be defined as “the process by which the essential properties required for the solution of a problem are extracted while hiding the details of the implementation of the solution from the programmer” [12]. There is no single definition of the level of abstraction of a language; however Watt [13] tells us that a high-level programming language is “a language that is more or less machine independent”. Table 2.1 lists out some examples of visual languages with different level of abstraction. Currently, there have not been any known attempts at creating a low-level visual language. The LogoBlocks visual programming language mentioned in Chapter 1 which was used to program an embedded microcontroller system exhibits the characteristics of a very-high level visual language, as the user has no control over the creation of variables and all of the lower level functions are hidden from the user. However, LogoBlocks only supports a specific platform. This conflicts with the previous definition by Watt. One possible explanation to this contradiction is that the language (LogoBlocks) itself is flexible and can be implemented on other platforms, but currently only one platform exists. Other high level languages like C/C++ are portable across different platforms, provided that a compiler is available. The same can be applied to LogoBlocks.
Table 2.1 Language Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Instructions</th>
<th>Memory Handling</th>
<th>Textual</th>
<th>Visual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels</td>
<td>Examples</td>
<td>Examples</td>
<td>Examples</td>
<td>Examples</td>
</tr>
<tr>
<td>Low Level Languages</td>
<td>Machine-like</td>
<td>direct memory access and allocation</td>
<td>assembly</td>
<td>language</td>
</tr>
<tr>
<td></td>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High Level Language</td>
<td>expressions and</td>
<td>memory access and allocation through</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>explicit flow of</td>
<td>control through operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Very-High Level</td>
<td>fully abstract</td>
<td>hidden memory access and automatic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Languages</td>
<td>machine</td>
<td>allocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2 The benefits of visual programming

Conventional textual programming is a mature discipline with many advocates. Textual programming boasts both stable design tools as well as advanced programming environments which include various program and data visualization tools. The question which must therefore be asked is what advantage there is in using a visual programming language system.

There are a number of advantages to be gained when programming using a visual language as opposed to a textual one. Just as high-level languages can be more tuned toward problem solving; visual languages can be more expressive than textual ones. This means that the appearance of the program can more easily reflect the programmer’s understanding of the problem to be solved. Generally, visual languages are able to provide programmers or users with a better understanding of what the program can do and how it works [14] [15]. This is accomplished by presenting the program in a visual notation that is more easily understood by the programmer. The advantages of visual notations of programs over textual notations include [8]:

- Visual languages can convey more meaning in a more concise unit of expression
- Visual languages can help understanding and remembering
- Visual languages can make programming more interesting
- The human mind is visually oriented

Furthermore, visual languages can also eliminate the need to remember program syntax in textual languages, and better visualize the pathways that the program is following.
2.3 The uses of visual languages

Visual languages have been used for a wide variety of applications. These applications can be loosely organized into three categories, depending on the target audience of the visual language.

- Visual language can be used by people who have no experience in conventional (textual) programming.
- Visual language can be used to assist people in learning textual programming.
- Visual language can be used by experienced programmers.

The problem domain of the visual language dictates the possible users of that language. There are many visual languages that are accessible to both experienced programmers and non-programmers, but these languages usually have a small problem domain. On the other hand, it is quite difficult to design a language which has a large problem domain and yet is accessible to non-programmers.

Visual languages are often found to be used as a tool to assist people to learn textual programming or being learned by non-programmers as these groups of people will often find the Visual language more intuitive and easy to learn. In these (usually, but not necessarily) limited problem domain visual languages, there are usually fewer graphical symbols involved. Users will be able to recognize the different graphical elements more easily and the method of combining the visual elements to form a program can be easily explained.

Visual languages do not exclude the use of text in program definitions. Possible uses of text in a visual language are for naming objects, for expressing mathematical formulas, or for commenting programs. Therefore, another possible use for visual languages is as an add-on for a conventional textual language. By integrating a visual and a textual language, the particular advantages of each can be utilised to overcome some disadvantage of either. A detailed examination of this can be found in [16], but in general this technique consists of the visual program being converted into a textual equivalent and being integrated (or perhaps reintegrated) with a textual program which can then be dealt with normally. The ability to convert visual programs into its textual equivalent will also aid users in learning textual programming and provide
more experienced programmers a way to fine tune the program.

Visual programming languages provide new ways of expressing computation which can be of interest to experienced programmers. Also, attributing particular graphical characteristics such as texture, colour and shape to corresponding program structures can help novice programmers recognise different elements of a program.

2.4 Problems associated with visual languages.

One of the most common and widely publicized problems is the screen space problem. This problem is widely referred to in visual language literature, for example, [17]. Simply stated, it means, “The problem with visual languages is that you cannot have more than 50 visual primitives on the screen at one time”. This problem would be more evident when designing visual languages for low level programming languages as more icons would be needed to accomplish a task. One workaround for this problem is to use subroutines or macros to reduce the number of icons on a screen.

Another criticism directed towards visual languages by people with little experience in the area is that “naming is easier than drawing new icons”. The truth of this criticism often depends upon the programmer in question and the program being written. Although it is probably true that naming is almost always quicker than drawing new icons. The easy way for a visual language to overcome this criticism is for it to allow named icons to be used in the place of drawn ones if necessary.

There are also performance issues with visual programs, namely program input speed. Experienced programmers often find they can type (using a keyboard) a textual program faster than they can construct (using a mouse) a visual program. This problem can be offset by a number of factors, firstly, visual programs are easier to understand which makes it easier for a programmer to both understand his own code and to understand somebody else’s code. Visual programs can also be easier to debug because it can be more immediately clear to the programmer what the program is doing. Also, a good GUI (Graphical User Interface) for a visual programming language will minimize any slowdown caused by using a graphical editor.
Lastly, is the issue of portability in visual programming languages. In most cases, visual languages were designed with a single problem area in mind and are usually targeted at specific platforms or applications. The advantage of expressing the problem visually comes at the cost of less abstraction. This results in reduced portability of the program across different platforms.
3 Design Considerations

3.1 Introduction

This part of the thesis will focus on the design requirements of a low level visual programming language for mid-range PIC microcontrollers. Unlike high-level languages, low level languages almost always have specific target platforms where the hardware architecture of the microcontroller has a significant influence on the low level language. It is possible to have a more generic low level language that would be portable across different architectures. However, this would also mean that special features of a particular architecture or microcontroller model may not be accessible as the language would no longer be generic if it is to support certain special features found only on particular microcontroller models or architectures. The work on the current research will mainly focus on creating a visual language that is specific to the mid range PIC microcontroller which uses a 14-bit instruction set. However, some effort has been made to ensure the visual language can be adapted to other microcontrollers without significant changes to the structure and grammar of the visual language.

There are a number of issues to be examined when designing a VPL. These include the syntax of the visual language, the visual language editor, the interface with the microcontroller system, and the execution of the visual program. These issues make up different portions of a completed visual language environment and are interrelated and play an important role in the successful design of a visual programming language.
3.2 Syntax of VPL

According to Ben-ari [18], “the syntax of a programming language is a set of rules that define what sequences of symbols are considered to be valid expressions in the language.” The same definition applies to visual languages. There are a number of different ways of specifying the syntax of a visual language; the most common method is to use a syntax which is an extension of the syntax for conventional textual languages [19].

The syntax of the VPL is designed to bear close resemblance to its textual equivalent. This is accomplished by using a fine grained language where each symbol or icon in the visual language translates to a single instruction in assembly language. As a fine grained language, it has the advantage of straightforward translation into its textual equivalent, and also from a textual language into its visual equivalent. It also provides unprecedented flexibility to the user as it is essentially a textual language with a visual shell. However, as with all fine grained languages, it is less suitable to construct complex and large programs. This is well suited for its target platform of 8-bit microcontrollers which has limited memory and processing power.

Syntax checking in visual languages is usually implemented graphically and dynamically through the use of a structured visual language editor. The editor uses specific rules on the placement of graphical symbols and the graphical objects used within the editor have a set of properties that can be configured with minimum subjective input.
3.3 Usage of an appropriate metaphor

A metaphor is a familiar analogy for how the programming system works. When the metaphor is good, users can infer how the programming works by referring to their existing knowledge and expectations about how the modeled system works; otherwise they might be required to learn a collection of rules that seem arbitrary. In order to maximize this transfer of knowledge, the metaphor should be based on and conceptually close to a concrete real-world system that is widely known by the user audience [20]. Studies by Mayer have shown that using familiar metaphors allows users to apply their knowledge and experience on the metaphors to aid them in understanding the programming language [21]. Frequently a pictorial representation of how a system will work is easier to understand than a lengthy amount of text.

The flowchart diagramming technique that was often used to describe pseudo code is a logical metaphor to be used for the VPL. A flow chart is defined as a pictorial representation describing a process being studied or in this case it describes the behavior of a program. Flow charts tend to provide people with a common language or reference point when dealing with a process or a program. It represents a familiar model of visual notation where users can infer how the programming system works by referring to their existing knowledge and expectations on flowcharts. Work by Scanlan [22] indicated that flowcharts had a significant advantage for the time needed to comprehend an algorithm. These effects were observed in simple cases as well as in complex ones. Studies of programming students by Vessey and Weber [23] and by Cunniff and Taylor [24] showed similar results.
3.4 Implementation of VPL

The flowchart methodology was used as the basis for the visual programming language because it presents a familiar model for programmers to work on. Nonetheless, flowcharts are mainly suited towards less complex programs. As the program becomes big and complex, a flowchart becomes less effective. This is a problem that will be inherited by the VPL and will need to be addressed. Since the current scope of the research is to provide a tool for novices to program microcontrollers, the flowchart implementation is expected to be advantageous in this case.

The flowchart technique has a wide range of application in diverse fields. Each serves different purposes and is used for different applications. The one that this VPL will be based on is a flowchart technique that is commonly used to visualize assembly programs.

Figure 3.1 shows a typical flowchart diagram used to visualize an assembly program. As seen in the figure, each icon in the flowchart diagram uses a specific shape which was specified in the ANSI standards to represent its primary function. Oval shapes represent start and termination. Parallelograms are used to represent data (in this case, assignment of values to variables). Rectangle shapes represent processing of data, and diamond shapes are used to represent decision making process. Typically, each flowchart symbol can represent one or more assembly instructions. However, in order to use the flowchart as a visual language, each flowchart symbol must have a well defined function that can be easily translated into its assembly language equivalent. This begins by organizing the assembly instructions into groups according to their functions, and assigning an appropriate flowchart symbol to the grouped instructions.

Figure 3.1: A typical flowchart diagram
Table 3.3 lists the entire mid-rage PIC microcontroller instruction set, organized into groups according to the operations they perform. The first group of instructions performs data assignment operations and mainly deals with assigning values to memory locations. The second and third group performs data processing operations, while the fourth group contains decision operators. This is followed by branch operations, subroutine operations and special operations. The visual language will utilize additional symbols as the conventional flowchart symbols used to visualize assembly programs do not provide enough variety. The symbols were carefully selected such that they are easily distinguishable from each other. The distinction between the symbols is reinforced by using different colors for different symbols. It is important to have clear distinctions between different symbols to avoid confusion and misinterpretation. “If objects look alike, kids would expect them to behave alike” [25]. Table 3.2 shows all symbols used in the visual language.

Table 3.2: Flowchart symbols and its relative meaning

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Assignment Symbol]</td>
<td>Assignment – use to assign a value to a variable.</td>
</tr>
<tr>
<td>![Calculate Symbol]</td>
<td>Calculate – contains arithmetic and logical data processing operations on variables or values.</td>
</tr>
<tr>
<td>![TextBox Symbol]</td>
<td>TextBox – used to place comments or additional assembly instructions or assembler directives that were not supported in the visual language.</td>
</tr>
<tr>
<td>![Decision Symbol]</td>
<td>Decision – Skips/executes next symbol based on outcome of preset conditions.</td>
</tr>
<tr>
<td>![Decision with Counter Symbol]</td>
<td>Decision with counter – Increments/Decrements a counter, skips next symbol when counter is zero.</td>
</tr>
<tr>
<td>![Goto Symbol]</td>
<td>Goto – unconditional branch to a predefined position (indicated by label, or subroutine name)</td>
</tr>
<tr>
<td>![Label Symbol]</td>
<td>Label – used as a target destination for Goto operation.</td>
</tr>
<tr>
<td>![Call Symbol]</td>
<td>Call – calls a subroutine.</td>
</tr>
<tr>
<td>![Start/End Indicator]</td>
<td>Start/End indicator – used to indicate start and end of a subroutine</td>
</tr>
</tbody>
</table>
Table 3.3 Mid-Range PIC Family instruction set [26]

<table>
<thead>
<tr>
<th>Data Assignment Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRF f</td>
<td>Clear f</td>
</tr>
<tr>
<td>CLRW -</td>
<td>Clear W</td>
</tr>
<tr>
<td>MOV F f, d</td>
<td>Move f</td>
</tr>
<tr>
<td>MOVWF f</td>
<td>Move W to f</td>
</tr>
<tr>
<td>MOVLW k</td>
<td>Move literal to W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Processing Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDLW k</td>
<td>Add literal and W</td>
</tr>
<tr>
<td>ANDLW k</td>
<td>AND literal with W</td>
</tr>
<tr>
<td>ADDWF f, d</td>
<td>Add W and f</td>
</tr>
<tr>
<td>ANDWF f, d</td>
<td>AND W with f</td>
</tr>
<tr>
<td>COMF f, d</td>
<td>Complement f</td>
</tr>
<tr>
<td>DEC F f, d</td>
<td>Decrement f</td>
</tr>
<tr>
<td>INC F f, d</td>
<td>Increment f</td>
</tr>
<tr>
<td>IOR L W k</td>
<td>Inclusive OR literal with W</td>
</tr>
<tr>
<td>IORWF f, d</td>
<td>Inclusive OR W with f</td>
</tr>
<tr>
<td>RLF f, d</td>
<td>Rotate Left f through Carry</td>
</tr>
<tr>
<td>RRF f, d</td>
<td>Rotate Right f through Carry</td>
</tr>
<tr>
<td>SUBL W k</td>
<td>Subtract W from literal</td>
</tr>
<tr>
<td>SUBWF f, d</td>
<td>Subtract W from f</td>
</tr>
<tr>
<td>SWAPF f, d</td>
<td>Swap nibbles in f</td>
</tr>
<tr>
<td>XOR L RW k</td>
<td>Exclusive OR literal with W</td>
</tr>
<tr>
<td>XORWF f, d</td>
<td>Exclusive OR W with f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit-oriented data processing Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BCF f, b</td>
<td>Bit Clear f</td>
</tr>
<tr>
<td>BSF f, b</td>
<td>Bit Set f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decision Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DECF SZ f, d</td>
<td>Decrement f, Skip if 0</td>
</tr>
<tr>
<td>INCFSZ f, d</td>
<td>Increment f, Skip if 0</td>
</tr>
<tr>
<td>BTFS C f, b</td>
<td>Bit Test f, Skip if Clear</td>
</tr>
<tr>
<td>BTFS S f, b</td>
<td>Bit Test f, Skip if Set</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO k</td>
<td>Go to address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subroutine Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL k</td>
<td>Call subroutine</td>
</tr>
<tr>
<td>RETFIE -</td>
<td>Return from interrupt</td>
</tr>
<tr>
<td>RETL W k</td>
<td>Return with literal in W</td>
</tr>
<tr>
<td>RETURN -</td>
<td>Return from Subroutine</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR WDT -</td>
<td>Clear Watchdog Timer</td>
</tr>
<tr>
<td>NOP -</td>
<td>No Operation</td>
</tr>
<tr>
<td>SLEEP -</td>
<td>Go into standby mode</td>
</tr>
</tbody>
</table>

\[ \begin{align*} f &= 7\text{-bit register address} \\
 d &= 0 \text{ for destination } W, 1 \text{ for destination } f \\
 b &= 3\text{-bit bit address} \\
 k &= 8\text{-bit literal value (11-bit for CALL & GOTO)} \end{align*} \]
3.5 Visual Language Environment

A visual language environment is a collection of different components which allow the easy editing, syntax checking and, running of visual programs. In contrast with conventional textual language environments, the level of interconnection between the different parts of the environment is usually relatively high. In a textual language, a program can ordinarily be written in the editor for one environment, syntax checked and compiled using a compiler from a different environment and, perhaps executed or interpreted in a different environment. Visual languages however, are frequently environment specific, that is, a visual language editor can only edit a single visual language, the syntax checker and compiler can only deal with one language, and, the execution process is restricted to a single language.

There are a number of issues that must be addressed when designing a visual language environment for a low level VPL, notably, the user interface of the environment. The visual language environment must provide users with a comfortable and intuitive interface so that users spend less time learning to use the interface and more time on constructing programs and solving problems. Hence, the development of the visual language environment is mainly focused on the user interface. Due to the limited time available for development, the visual language environment will employ the use of a language translator (instead of a compiler) to translate the visual language into its textual equivalent (assembly language), then use a third party assembler to assemble it into a binary machine code file. This way, syntax checking and error detection is handled by the third party assembler and error messages, if any will be parsed in by the visual language environment and displayed to the user. By doing so, the development time for the visual language environment can be greatly reduced and yet the visual language environment is able to support a wide range of microcontroller models that share the same instruction set.

3.5.1 Visual Language Editor

A visual language editor is used to enter/edit visual language programs on a computer. Editors for visual languages are generally based on placing discrete graphical objects like lines, circles, arrows, icons, etc. on a drawing area. The syntax of the grammar for the visual language will operate on the properties of these objects. The editor should also provide copying protocols to produce duplicate or near duplicate elements in order
to lessen the time taken to reproduce similar program fragments. This is analogous to the operation of cutting and pasting in textual editors. The editor will also aim at minimizing the shift in user focus between keyboard and mouse. This is achieved by using dropdown menu selection in configuring the properties of the individual objects. The editor will also need to address the screen space issue, to maximize the display of graphical symbols on the screen and provide tools that facilitate navigation of the visual program.

3.5.2 Visual program execution

Visual programming languages often support execution of the visual program in the visual language environment to provide immediate feedback to the user. The ability to test partial solutions is an important feature as it aids problem solving. Users should be encouraged to test their code incrementally as it is a useful debugging strategy. Hence, it is important for the low level VPL to provide means of quickly programming the microcontroller to facilitate testing and debugging. This is accomplished by providing a macro function in the visual language environment that assembles the visual program to produce binary code, then downloads it into the microcontroller, in successive steps.
4 CoreChart IDE

4.1 Overview
This chapter will introduce CoreChart, the integrated development environment that has been developed to streamline the process of programming mid-range PIC microcontrollers. This is accomplished by providing a low level visual language for constructing assembly languages and automating the task of downloading the program into the microcontroller. First, users construct programs visually by manipulating the icons in the program work area. CoreChart will then translate the visual program into its textual equivalent (in assembly language). The textual program is then assembled using a conventional assembler and then downloaded to the microcontroller using third party programmer software. The entire process of translating, assembling, and downloading of the program is automated and is transparent to the user, allowing the user to focus on writing the program. An official assembler released by Microchip and a third party programmer software (IC-Prog) was used to reduce the development time, and to provide more comprehensive support for different mid-range PIC microcontroller models, and support for a wider range of hardware PIC programmers.

The development of CoreChart was an ongoing effort and is still in progress today. At the start of the development cycle, it is desirable to produce a working program as soon as possible so that it can be used and tested by users. The structure of the program must be carefully planned such that it will allow for additional functions to be implemented without much difficulty. Two months were spent on background research and planning of the program structure, followed by four months of coding. A working program was produced at the end but not all planned features were implemented. The working program was then beta tested by staff at eLabtronics and a limited number of eLabtronics clients to discover bugs and to provide feedback on the user interface of the program. During the beta testing period, the planned features were successively implemented together with the additional features suggested from feedback provided by the beta testers.
4.2 Implementation

CoreChart was written using Visual Basic 6, which was designed to be simple to use and allows for rapid application development. Visual Basic supports object oriented programming and contains a graphical user interface design tool. The GUI design tool allows graphical Windows programs to be constructed by placing pre-built graphical objects and widgets on screen, rather than writing numerous lines of code to describe the appearance and location of interface elements. Visual Basic also supports creation of event driven applications whereby code is executed in response to an event (i.e. user clicking on a button). However, using Visual Basic also limits CoreChart to Windows platforms and making code migration to other operating systems difficult.

There are three main types of source files in Visual Basic: Forms, Modules and Classes [27]. Forms are classes that define the windows/dialog object. They contain user configurable properties and support event driven procedures that are executed in response to events. Forms have a visual component attached. Users build the interface of the program by placing buttons, labels, textboxes etc. onto a form, and then write code for the individual events, i.e. when the form loads, or when a button is pressed. Modules contain user defined data types, functions and procedures that are used in Forms. Procedures in Modules are not associated to events and are only executed when called in other procedures. Lastly, Classes in Visual Basic are used to define objects. Objects can be viewed as a combination of custom data types and a set of procedures to access/modify the data. Data stored within an object can only be accessed via the set of procedures provided by the object.

The CoreChart source code is well structured and is distributed in 17 Forms, 14 Modules and 4 Classes. The 17 Forms are definitions of the main window and the numerous dialogs used in CoreChart. Forms have a graphical part where the user interface is defined and a textual part where code is written in response to an event. Forms are considered to be objects in Visual Basic and an instance of the form is created during runtime. The 14 modules contain commonly used procedures and functions. Modules provide a way to organize procedures with similar functions into groups. This facilitates maintenance, debugging, or code reuse as modules can be easily moved by importing the module into another project. The 4 Classes are used to define two data objects in CoreChart: the Icon Cluster and Variable Cluster object.
The former is used to store the visual program and the latter is used to store variables used in the visual program. A resource file was used to store the bitmap graphics used in the CoreChart user interface. Lastly, the code also employs extensive use of Windows APIs to access native Windows functions that are otherwise unavailable in Visual Basic.

CoreChart employs the use of shared libraries to implement support for different microcontroller models. The shared library was also written in Visual Basic (as an ActiveX DLL). It is essentially a compiled class module that provides public methods and properties for applications to interface to. All microcontroller model specific information is stored in the shared library and can be accessed by CoreChart via the interfaces provided by the shared library. A library for a new microcontroller is generated by modifying a template library with microcontroller specific information, and then recompiling the library.
4.3 Main Window

The main window of CoreChart is shown in figure 4.1. The name of the active program is shown on the title bar on top of the main window together with the microcontroller model. The main window is divided into six subsections, they are

1. Program workspace
2. Navigation tools
3. Editing tools
4. Icon palette
5. Icon properties
6. Main menu bar

Each subsection will be described in detail in the following pages.
4.3.1 Program workspace
This workspace area (indicated by \( \text{\ding{157}} \) in Figure 4.1) is where the visual program is displayed. It is also used by users to construct and edit the visual program. It occupies the largest portion of the main window. Each of the icons displayed in the program workspace is numbered. When an icon(s) is selected, it will be highlighted by drawing a black box over the icon and shading the number box on the icon. This provides a visual feedback to the user to indicate that the icon(s) is selected. The program workspace only displays one subroutine at a time to reduce cluttering and allow the user to be more focused on the current subroutine.

4.3.2 Navigation tools
The navigation tools (indicated by \( \text{\ding{157}} \) in Figure 4.1) are used by the user to traverse the program. The user can select to view and edit a different subroutine by selecting it on the drop down list in the upper right corner of the main window. The horizontal scroll bar located on the bottom of the main window is used to scroll through the pages of the subroutine. The user can also jump to the first and last page of the subroutine by clicking on the first page and last page buttons. Information on the current range of icons displayed on screen and the total number of icons in the subroutine are displayed in a status box on the bottom part of the main window.

4.3.3 Editing tools
This section provides users with the familiar editing functions that are normally found in many other programs. The user is able to perform tasks like copy, cut, paste, delete and undo operations on the icons by clicking on the respective buttons in the editing tools section (indicated by \( \text{\ding{157}} \) in Figure 4.1).

4.3.4 Icon palette
There are 12 icons in the icon palette, they are: Assignment, Calculate, Set/Reset, Decision, Decision with counter, Text Box, Goto, Label, Timing, Call, Return, and Selection icon. Users create programs by populating the program workspace area with icons from the icon palette. The selection icon is used to select the icons on the program workspace so that the icon properties can be changed, or editing functions can be performed. The return icon is grayed out when the user is editing the main
program.

4.3.5 Icon properties
This section displays the properties of the individual icons when the user selects the icons on the screen. In most cases, the user is able to make changes to the icon properties by just using the mouse. This minimizes the shift of focus of the user from mouse to keyboard. There is also a comment field for users to attach a comment onto the icons.

4.3.6 Main menu bar
The main menu bar is located near the top of the main window (indicated by ☐ in Figure 4.1) right beneath the title bar. The main menu bar consists of four menu items: File, Edit, Options, and Help. Each item contains a drop down submenu. When the user clicks on any of the menu bar items, the drop down menu will be displayed giving the user more options to choose from. Items (mainly in the Edit submenu) that are unavailable (prerequisite not satisfied) will be grayed out and will not be executable. Figure 4.2 shows all of the items of the main menu bar (indicated in Bold) and its drop down submenu items.

<table>
<thead>
<tr>
<th>File</th>
<th>Edit</th>
<th>Options</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Undo</td>
<td>Send To Pic</td>
<td>Contents</td>
</tr>
<tr>
<td>Open</td>
<td>Cut</td>
<td>Assemble Program</td>
<td>Chip Family</td>
</tr>
<tr>
<td>Save</td>
<td>Copy</td>
<td>View ASM File</td>
<td>Data</td>
</tr>
<tr>
<td>Save As</td>
<td>Paste</td>
<td>Import Subroutine</td>
<td>Chip Technical</td>
</tr>
<tr>
<td>Print</td>
<td>Delete</td>
<td>Export Subroutine</td>
<td>Data</td>
</tr>
<tr>
<td>Print Setup</td>
<td>Save Clipboard</td>
<td>Show Full IC-Prog</td>
<td>About</td>
</tr>
<tr>
<td>Exit</td>
<td>Load Clipboard</td>
<td>Chip Configuration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subroutines</td>
<td>Hardware Settings</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Labels</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2: Main menu bar with all its dropdown submenus
4.3.6.1 File main menu bar item
This menu bar item consists of six submenu items. The first four items are used to create a new program, load an existing program and to save the currently open program. The Print item allows printing out of the visual program in a pseudo code format along with additional program information (refer to Appendix C for a sample printout). The print setup item is used to set up the printing options where user can select the printer, configure printer properties, and choose paper type and orientation. The Exit submenu item is used to terminate the application. The user will receive a prompt confirming the termination if the current program has not been saved. This is a means of avoiding accidental loss of data through careless decision.

4.3.6.2 Edit main menu bar item
There are eleven submenu items in the edit menu. The first five items are normal editing functions found in other editors (Cut, Copy, Paste, etc...). Save/Load Clipboard is used for copying icons across different instances of CoreChart. A user can have multiple instances of CoreChart running, each instance editing a different program. To copy icons across different instances of CoreChart, the user will first use the normal copy function, and then use the Save Clipboard function to make it available for other instances of CoreChart. After that, the user is able to use the Load Clipboard function on another instance of CoreChart to load the copied icons into that instance of CoreChart.

The last four items in the submenu is used to bring up relevant dialogs that allows the user to edit and maintain the mentioned items. The individual items and its functions will be described in more detail in the dialog windows section.

4.3.6.3 Options main menu bar item
There are ten submenu items in the Options menu. (1) The Send To Pic submenu item is used to download the program onto the microcontroller. It will first check if the program is saved and assembled. If not, it will prompt the user to save the program and assemble it before download to the microcontroller; (2) The Assemble Program submenu item is used to only assemble the program. This is used to check for potential errors and to generate the binary file (with a .HEX extension); (3) The View
ASM File submenu item allows users to view the assembly file generated from the visual program. This way, users can familiarise themselves with assembly language by cross referencing with the visual program; (4) The Import/Export Subroutine submenu item allows users to save a subroutine as a module and later import it into another program. This will be discussed in more detail in the dialog windows section; (5) The Show Full IC-Prog submenu item enables/disables running the third party programmer (IC-Prog) in full mode during the programming operation. This option provides users with additional control when programming microcontrollers. The remaining four submenu items will invoke its respective dialogs which will be described in more detail in the dialog windows section.

4.3.6.4 Help main menu bar item
This main menu bar contains submenu items that provide information to assist users in learning how to use CoreChart, and technical information on PIC microcontrollers. The first submenu item (Contents) brings up a CoreChart help file where information on various functions of CoreChart can be obtained. The second submenu item (Chip Family Data) opens a PDF file that provides information on the mid-range PIC microcontroller family. The third submenu item opens a technical datasheet file specific to the PIC microcontroller that is currently used in the program. The last submenu item opens up an About dialog box, displaying general information about CoreChart.
4.4 Dialog Windows

This section describes the various dialogs present in CoreChart. All of the dialogs can be accessed from the drop down menus or using short-cut keys.

4.4.1 Subroutines Dialog

The subroutines dialog allows user to maintain the subroutines in the program. A list of subroutines currently in the program is displayed in subroutine list (indicated by ○ in Figure 4.3). A new subroutine is added by first typing the subroutine name into the textbox (indicated by ○ in Figure 4.3), then clicking on the ‘Add Subroutine’ button. Selecting a subroutine in the subroutine list will cause the status bar (indicated by ○ in Figure 4.3) to display additional information related to the subroutine. Selecting a subroutine also enables a host of functions that can be applied to the subroutine. These functions include rename, delete, edit, and move up / down. Right-clicking on the selected subroutine further brings up three additional functions: Set/Clear ORG Directive and Subroutine Information. The Set/Clear ORG Directive allows the user to specify the program memory location of subroutine which was normally assigned automatically. The move up/down function and Set/Clear ORG Directive is for advanced programming where it is necessary to control the physical position of the subroutine in the microcontroller program memory.

![Subroutine List]

Figure 4.3: Subroutines Dialog
4.4.2 Variables Dialog

The variables dialog allows the user to maintain the list of variables used in the program. Currently only a byte variable type is used in CoreChart. Other variable types such as character, string, float, and integer are not supported. This is because the instruction set which the visual language is based on only processes 8-bit data, which is also reflected in the visual language itself. String, float and integer are multiple byte data types which were not supported. Technically, character types can be supported since it only occupies a single byte, but that was not currently implemented in CoreChart.

The variables dialog uses a similar layout to the subroutines dialog and thus follow similar conventions in creating and maintaining variables. Variables must first be declared before they can be used in the visual program. Corechart only supports creation of variables in the first bank. This limits the maximum number of user variables to 96 or the total number of general purpose registers the microcontroller has, whichever is lower. The general purpose registers in other banks which are not supported by CoreChart can still be utilized by using the register indirect addressing method. The user variable list (indicated by \( \bigcirc \) in Figure 4.4) shows the list of variables created by the user. The list also displays the memory location of the variables. The user has no direct control over the placement of variables in data memory. However, the variables can be shifted up and down using the Move Up/Down buttons.

![User Variable List](image)

Figure 4.4: Variables Dialog
4.4.3 Constants Dialog

The constants dialog allows the user to declare constants that are used in the program. The constants list shows all of the constants declared by the user for the program. The user adds a constant by entering the information into the respective textboxes on the right, and then pressing the ‘Add’ button. To modify an existing constant, select the constant to be modified, the values on the textboxes will change to the values of the selected constant. Change the values, then press ‘Apply Chgs’ to save the changes.

![Constants Dialog](image)

Figure 4.5: Constants Dialog

4.4.4 Labels Dialog

The labels dialog displays all valid labels in the program, these includes start label and subroutine names. The relative position of the labels in the subroutine are also shown. The user is able to rename or delete the label using the buttons on the dialog.

![Labels Dialog](image)

Figure 4.6: Labels Dialog
4.4.5 Import Subroutine Dialog

This dialog allows the user to import a subroutine previously exported from CoreChart using the Export Subroutine Dialog. The user can import a subroutine that is located on another folder by using the folder list window (indicated by ○ in Figure 4.7). The exported subroutine modules available in the folder are displayed in the middle window (indicated by ○ in Figure 4.7). More information regarding the subroutine will be displayed on the right textbox (indicated by ○ in Figure 4.7) once a subroutine is selected.

To import a subroutine, the user will need to select the subroutine to be imported by placing a tick beside the subroutine, then click on ‘Import Selected’ button. A status screen will be displayed showing the result of the import process. The user can have more control over the import process by checking the ‘Manual Control of Name Change’ checkbox. This causes a message box to be displayed asking for confirmation to auto-rename every time a duplicate label or variable is found.

![Import Subroutine Dialog](image)

Figure 4.7: Import Subroutine Dialog
4.4.6 Export Subroutine Dialog

This dialog allows the user to export a subroutine from the current program. The subroutine is exported as a file with a .BSM extension. The main program can also be exported as a subroutine. A subroutine that contains nested subroutines is also exportable. When a subroutine is selected in the subroutine list (indicated by \( \mathbf{O} \) in Figure 4.8), additional information will be displayed in the status bar, listing the number of user variables used and the total number of subroutines it contains. The user can enter additional information in the textbox (indicated by \( \mathbf{O} \) in Figure 4.8). This information is displayed to the user when the user selects the subroutine in the import subroutine dialog.

![Figure 4.8: Export Subroutine Dialog](image)
4.4.7 Chip Configuration Dialog

The chip configuration dialog allows the user to configure common microcontroller settings. These settings include data direction of the I/O ports (configured by clicking on the direction arrows), oscillator type, data protection bits, etc. Some of the configuration settings are model specific, hence the chip configuration dialog is designed as an external shared library which can be changed for different microcontroller models.

![Chip Configuration Dialog](image)

Figure 4.9: Chip Configuration Dialog
4.4.8 Hardware Settings Dialog
This dialog enables the user to configure the third-party software programmer (IC-Prog) hardware settings. IC-Prog supports a comprehensive list of hardware programmer interfaces (selectable from the dropdown listbox).

Other settings include selection of ports, interface type and I/O Delay.

Figure 4.10: Hardware Settings Dialog
4.4.9 Program Preference Dialog

The program preferences dialog allows users to adjust some of the program settings to suit personal preferences. The dialog contains three pages, User Interface, Display Size, and File Associations. They are accessed via the tabs near the top of the dialog. The User Interface page contains settings that slightly alter the user interface of the program. Some users may find some particular settings useful while others may find it less appealing. The second tab allows the user to change the display size of CoreChart. A larger display size allows users to see more icons on the screen but requires a higher resolution screen. The last tab allows users to set the associations of CoreChart save files and change the icons for the save files.

![Program Preferences Dialog](image)

Figure 4.11: Program Preferences Dialog
4.4.10 Chip Model Selection Dialog

The chip model selection dialog allows the user to switch between different microcontroller models that share the same instruction set. CoreChart was written with the intention of supporting multiple microcontroller models from the same family (i.e. same instruction set, different peripherals and memory size). Hence all information and data that is microcontroller specific is placed into a separate library file where it can be swapped to support a different microcontroller model. The dialog shown in Figure 4.12 facilitates the process by listing available library files. The user is able to change to another microcontroller model by selecting it from the list.

![Select Chip Model]

Figure 4.12: Chip Model Selection Dialog
4.4.11 ASM Viewer and Assembly Status Dialog

The ASM Viewer dialog provides a convenient way for users to view the assembly equivalent of the visual program. As CoreChart only generates the assembly program during the save file process, visual programs must be saved before their assembly equivalent can be displayed in the ASM Viewer.

The Assembly Status dialog only shows up when there are errors during assembly. It displays the error messages on screen and is able to highlight the error-causing line when the user clicks on the error in the list. This allows the user to quickly locate the source of the error and rectify the problem. Assembly errors are uncommon in CoreChart but not unavoidable. Most errors originated from TextBoxes as it allows the user to insert any word directly into the assembly code. Figure 4.13 is an example of an error generated by improper placement of words in a TextBox.

![ASM Viewer](image)

![Assembly Status](image)

Figure 4.13: ASM Viewer and Assembly Status Dialog
4.5 CoreChart Icons

This section explains the individual functions of the flowchart icons used in CoreChart. There is a total of thirteen different flowchart icons used in CoreChart, of which eleven can be manipulated by the user. The icons are Start Main, End Main, Assignment, Calculate, Set/Reset, Decision, Decision with Counter, Text Box, Goto, Label, Timing, Call, and Return icon. With the exception of Start and End icons, all icons have configurable properties that can be changed using the icon properties sub window. When the program is saved, a textual equivalent of the visual program is generated by translating each CoreChart icon to its assembly instruction equivalent based on its icon properties. Table 4.13 lists all configurable CoreChart Icons and their assembly equivalents.

Table 4.14: CoreChart Icons

<table>
<thead>
<tr>
<th>CoreChart Icon</th>
<th>Function Description</th>
<th>Example operation</th>
<th>Assembly Instruction</th>
</tr>
</thead>
</table>
| Assignment     | Used to assign values to a variable. Source can be a numerical value or another variable. | <Var> = 0  
W = 0  
W = <Variable>  
<Variable> = W  
W = <Lit> | CLRF  
CLRWF  
MOVWF  
MOVW |
| Calculate      | Used to perform arithmetic or logical operations on variables. | W = <Var> + W  
W = <Var> - W  
W = <Lit> + W  
W = <Lit> - W  
<Var> = <Var> + 1  
<Var> = <Var> - 1  
W = <Var> OR W  
W = <Lit> OR W  
W = <Var> XOR W  
W = <Lit> XOR W  
W = <Var> AND W  
W = <Lit> AND W  
Complement <Var>  
Bit Rotate <Var>  
Bit Swap <Var> | ADDWF  
SUBWF  
ADDLW  
SUBLW  
INCF  
DECF  
IORWF  
IORLW  
XORWF  
XORLW  
ANDWF  
ANDLW  
COMF  
RRF, RLF|
| Set/Reset      | Used to perform bitwise operations on variables | Set particular bit of variable ON/OFF | BSF, BCF |
Table 4.14: CoreChart Icons, continued from previous page

<table>
<thead>
<tr>
<th>Decision</th>
<th>Used to evaluate a condition and perform branch if true.</th>
<th>Skip the next icon if particular bit of (&lt;Var&gt;) is ON/OFF</th>
<th>BTFSS, BTFSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision with Counter</td>
<td>A combination of calculation and decision icon. Mainly used in a control loop.</td>
<td>Increase or decrease (&lt;Var&gt;) by 1, Skip next icon if (&lt;Var&gt;) equals 0</td>
<td>INCFSZ, DECFZ</td>
</tr>
<tr>
<td>Goto</td>
<td>Performs an unconditional branch, destination specified by label.</td>
<td>Goto Label1</td>
<td>GOTO</td>
</tr>
<tr>
<td>Text Box</td>
<td>A compatibility icon, any text entered in the text box will be transferred to the assembly format unaltered, can be used to put comments or assemble directives.</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Timing</td>
<td>Contains microcontroller specific instructions</td>
<td>No operation Standby Mode Clear WDT</td>
<td>NOP SLEEP CLRWDT</td>
</tr>
<tr>
<td>Return</td>
<td>Used to return control to the calling program.</td>
<td>Return Return from Interrupt Return with literal</td>
<td>RETURN RETFIE RETLW</td>
</tr>
<tr>
<td>Call</td>
<td>Used to call a subroutine.</td>
<td>Call</td>
<td>Call</td>
</tr>
<tr>
<td>Label</td>
<td>A destination anchor for goto icons.</td>
<td>(&lt;Label&gt;)</td>
<td>(&lt;Label&gt;)</td>
</tr>
</tbody>
</table>
4.6 Example Program in CoreChart

Figure 4.14 shows an example program constructed in CoreChart and its equivalent flowchart diagram. It is a simple program that counts down from ten until it reaches zero, then stops. The program is constructed by successively placing the icons on the work area and setting the individual icon properties. Figure 4.15 shows the assembly code generated by CoreChart for the visual program in figure 4.14.

Figure 4.15: Comparison of a CoreChart Program vs. a flowchart diagram

<table>
<thead>
<tr>
<th>START</th>
<th>;Start of Main Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>movlw 10</td>
<td>;Assign: A = 10</td>
</tr>
<tr>
<td>movwf A</td>
<td>;Label</td>
</tr>
<tr>
<td>decfsz A,F</td>
<td>;Count: A = A - 1, Skip if 0</td>
</tr>
<tr>
<td>goto Loop1</td>
<td>;GoTo: Loop1</td>
</tr>
<tr>
<td>goto $</td>
<td>;End of Main Program</td>
</tr>
</tbody>
</table>

Figure 4.16: Assembly program equivalent of the CoreChart Program
4.7 Limitations of CoreChart

The visual language structure of CoreChart is limited by the syntax of the Assembly Language it was based on. One apparent example was the usage of decision icons. In conventional flowchart diagrams (Figure 4.15) a decision icon usually redirects the flow of the program based on an outcome of a calculation. In CoreChart, the decision icon either executes or skips the next icon based on an outcome of a calculation. Usually, this icon is combined with a Goto icon to achieve the desired effect. Although it is possible to combine the two icons to form one compound icon that actually translates into two instructions, doing so would limit the flexibility of the programming language as it is also possible to have any other icons in place of the Goto icon.

However, this limitation was mainly constrained by the assembly language it was based on. Figure 4.17 shows a visual program in a modified variant of CoreChart. The visual language was based on PIC18F Family of Microcontrollers which uses a 16-bit instruction word. The PIC18F family uses an expanded instruction set with a total of 75 instructions. It contains conditional branch instructions which can be utilised by the visual language to produce a decision icon with a conditional branch, which bears more resemblance to conventional flowchart symbol.

Figure 4.17: CoreChart Program based on PIC18F Family
5 Assessment of the effectiveness of VPL

An assessment of the effectiveness of the low level Visual Programming Language is crucial in determining if it has accomplished what it was designed for: to simplify the process of programming microcontrollers and provide novice users with a tool that can assist them to learn microcontroller programming.

5.1 Method

The assessment was conducted via a task driven practical session with a short survey at the end of the session to obtain feedback from the subjects. The aim of the practical session was to test the usability of CoreChart and its effectiveness in allowing novice users to program microcontrollers. The practical consists of three microcontroller programming tasks. The three tasks are inter-related and need to be completed sequentially. The microcontroller hardware used is a low cost microcontroller development board from elabtronics called elab628. The development board uses a 16F628 mid-range PIC microcontroller. The development board contains eight light emitting diodes (LED), a buzzer and a press switch, which are connected to the microcontroller's 8bit input/output ports (PORTA & PORTB). The LED and buzzers provide a means of feedback to the user and the press switch is used as an input device to the microcontroller. The board contains a built in JDM programmer circuit to facilitate in-circuit programming of the PIC microcontroller using the PC serial port.

The first survey task aims to familiarize the subjects with programming microcontrollers by providing an example program which turns the first LED continuously on and off on the development board at a rate that is noticeable to the human eye. The subjects are guided through the process of opening the example program, compiling the program then downloading it into the microcontroller via the JDM interface on the development board. The subjects are also given an explanation of how the example program works.
The second task requires the subjects to alter the sample program to sequentially flash all eight LEDs on the development board. The aim of this task is to let the subjects familiarize with the VPL and also to introduce additional commands and functions of the VPL. The subjects are briefed on how this is done and are guided by a demonstrator until the task is complete.

The last task is to further modify the program so that the LEDs are flashed sequentially in the opposite direction when a button on the development board is pressed. This task is to demonstrate how microcontrollers obtain external input and produce a response to it. Detailed instructions were provided in writing so that the subjects were able to follow through the survey on their own. The subjects are also free to ask any questions. The time taken for the subjects to complete all three tasks is recorded and is used as a gauge for the effectiveness of the VPL.

Eighteen subjects were recruited to participate in this evaluation survey on the effectiveness of the VPL. Ten were high school students from Urbrae High School, aged between 14 and 17. Eight were undergraduate students from Adelaide University, aged between 20 and 23. Of the ten high school students, five students have prior knowledge of programming ("Experts") and are interested in the field of programming and electronics, while the other five have never written a program before and are completely new with computer programming ("Novices"). The remaining eight volunteers were undergraduate students from Adelaide University. All the university students have prior knowledge in textual programming languages and have at least taken a basic course in programming. Three out of the eight university students considered themselves proficient in programming and had some knowledge in assembly language programming ("Experts"). The remaining five students had rudimentary programming knowledge and were unfamiliar with assembly language programming ("Novices").

All the subjects were given the tasks based on CoreChart and a maximum time of ninety minutes to complete all three tasks. The survey was carried out in groups of three to five. All subjects in a group were given the same survey and were assisted by a demonstrator. The results have been divided into two main groups, high school students and university students. They were then further divided into two groups
based on their prior knowledge in programming languages. The entire survey was conducted over a period of two months, which included the time taken to produce the survey questions. The long period of the survey was mainly due to limited time arrangements with the participating high school and slow turn up of university survey subjects.

5.2 Results

The survey tested the effectiveness of the low level VPL in assisting novice users to learn microcontroller programming and to provide experienced users with a flexible programming tool that they would find easy to use and would reduce programming time. The assessment was based on the total time taken for an individual subject to complete all three tasks described in the previous section and the feedback from the subjects.

The results were as expected, with “expert” subjects faring better than “novice” subjects, and the university subjects doing better than high-school subjects. All high-school “novice” subjects were unable to complete the practical within the allocated time while the high-school “experts” took an average of 51 minutes. The university subjects were able to complete the tasks as all of them have some programming knowledge. On average, the “novice” subjects took 65 minutes while the “expert” subjects took 38 minutes to complete the practical. The post practical survey have revealed that “novice” subjects are less interested in programming and do not put an effort to do the practical. On the other hand, “expert” subjects are faring better presumably due to their proficiency in programming and interest in electronics.

<table>
<thead>
<tr>
<th></th>
<th>University</th>
<th>High School</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Novice</td>
<td>Expert</td>
</tr>
<tr>
<td>Time (Mins)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>65</td>
<td>45</td>
</tr>
<tr>
<td>65</td>
<td>73</td>
<td>28</td>
</tr>
<tr>
<td>73</td>
<td>64</td>
<td>42</td>
</tr>
<tr>
<td>58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td>65.2</td>
<td>38.333</td>
</tr>
</tbody>
</table>

Note: There is a maximum session time of 90 minutes.

Figure 5.1: Time taken to complete practical
5.3 Discussion

The results have indicated that the VPL is better accepted by the "expert" users than the "novices". This would seem to contradict with the purpose of the VPL as a means of providing novice users access to microcontroller programming. However, one must see that the notation of "novice" and "expert" is relative. A high-school student who shows interest in programming can be judged as an "expert" when compared to average high-school students, but in terms of accumulated knowledge in programming, he can only be classed as a "novice". The post practical survey has indicated that the high-school "experts" have generally found the VPL to be intuitive and easy to use, but not entirely foolproof. A common comment was the method of placing icons onto the work area (hence the process of building the program) was not the common drag-n-drop method. This caused some minor confusion but was quickly resolved when the user was accustomed to the method used in the VPL. Other comments include having more professional looking icons and some debugging functions.

However, the practical did not show strong evidence that the low level VPL would aid non-programmers (high school students who have never learned programming) in programming. Casual interviews with the "novice" subjects show that CoreChart did in fact assist some of them to better understand the program code, but not all subjects share this view. Some remain uninterested and just waited for time to pass or until someone tells them what to do. Generally, users who had no previous experience in programming found it hard to follow the instructions and complete the three tasks. They were also unfamiliar with flowcharts or flow diagrams used to aid code comprehension. This negates the effect of having the flowchart metaphor since the subjects were never familiar to it.

One common problem with the practical is that the subjects (especially the high school students) were reluctant to read through long paragraphs of instructions in order to do the survey. They resorted to asking questions even though the answers could be found on the instruction sheet. One method of addressing this issue in future would be to employ the use of a demonstration program that automatically guides subjects through the practical, displaying instructions step by step and highlighting
any potential problems. This would also improve the accuracy of the collected data as data collection can be handled by the demonstration program. Another issue is the level of involvement from the students. The students doing the survey were on a voluntary basis and they had no incentive to perform well, and hence they may not have taken the survey seriously. This becomes more apparent if the subject is not interested in the field. This issue can also be addressed by providing some form of reward for successful completion of the practical or by making the survey part of their coursework for which they will be credited based on their performance.

Overall, the assessment on the usability of the VPL has shown positive results for subjects that are interested. However, for the result to be credible, a more formal study has to be conducted on a larger scale with more samples. The current one can be viewed as a pilot study in which a more structured one can be devised.
6 Current Status and Future Work

A fully functional low level visual programming language has been developed using Visual Basic and is currently extensively used by eLabtronics. Branded CoreChart, it has been used both internally by eLabtronics and made available for sale as a product. CoreChart has also been used by numerous educational institutions. Some use it as a teaching aid in advanced electronics courses, others used by students in their school projects. Over the months, valuable feedback from the staff at eLabtronics and customers have helped improve CoreChart. It is now a viable tool that can be used to aid users in programming microcontrollers.

This concept was also implemented on the more advanced PIC microcontroller family, the PIC18C series. The adaptation was without many complications as the more advanced PIC18C series share a similar architecture with the mid-range series, with the only major change in instruction word length (from 14-bit to 16-bit). This led to the inclusion of 40 additional instructions which include conditional branch, table read and multiplication instructions. All additional instructions can be implemented into existing CoreChart icons with some modifications to the property settings of the icon. This is a good indication that the low level visual programming concept can be adapted to other microcontroller architectures.

One of the most common comments from users concerns compatibility with Windows XP. Users have problems downloading the compiled code onto microcontrollers due to improper set up of the third party programmer software. The third party programmer software requires direct hardware access to the computer’s I/O port in order to program the microcontroller. However, direct hardware access is disabled by default in windows XP and must be enabled manually. The programmer also supports programming the microcontroller using the Windows XP software application programming interface, but it has been less stable than the direct hardware access method. Currently, the workaround for this issue involves using an installation script to enable direct hardware access in Windows XP systems and configuring the third party programmer to use direct hardware access. In the long term, the programmer function should be integrated into CoreChart.
Other useful improvements to CoreChart would be:

- An addition of debugging functions like a simulator whereby the visual program can be executed immediately within CoreChart to provide users with immediate feedback. Another useful debugging function would be the ability to comment out blocks of icons so that it can be excluded from execution.

- Ability to provide a hierachal view of the visual program. This gives the user an overall perspective on the visual program.

- Allow users to create customized icons for subroutines so that the subroutine icon can relate more to its function.

- The ability to create macros and better utilization of assembler directives.

- Implementation on other microcontroller models with radically different architecture, possibly microcontrollers from other manufacturers like Atmel and Motorolla.

- Support for more types of programmer software. Currently CoreChart only supports IC-Prog which uses serial interface. Support for a USB programmer would be beneficial as all new computers come with USB port.
7 Conclusion

This thesis presented an alternative method of programming microcontrollers, using a low level visual programming language. By combining the use of the familiar flow chart metaphor and fine grained language, CoreChart is able to provide users with a simple to use yet flexible visual language environment to construct visual programs. The results of the survey have indicated that CoreChart may be unsuitable for non programmers as it is still moderately complex. After all, not all people are naturally adept at programming in low level languages. Hence, CoreChart is likely to be more accepted by people with moderate programming skills who are looking to expand their programming knowledge into the field of microcontroller programming.

In order for CoreChart to be usable by non-programmers, the visual programming language must bear more resemblance to flowchart diagrams. This however will require a coarse grained language which will be less flexible. It is quite difficult to design a visual language that can both appeal to non-programmers and more experienced programmers. However, it is possible to develop a separate, less complex version of the low level visual programming language that specifically addresses the issues faced by non-programmers. This way, novice users can begin using the ‘lesser’ version to accustom themselves with programming, then move on to the more complex version when their programming skills have improved.

Until now, textual languages have been the preferred method to write programs for microcontrollers. The low level VPL is not aimed at completely replacing conventional textual programming, but rather as a complementary tool to aid novice users to learn microcontroller programming. Due to the use of fine grained language, there is an upper limit on the complexity and size of the program written in CoreChart. The program will become increasingly hard to manage and debug as it gets larger and more complex. This holds true for any low level languages, textual or visual.

Generally, there has been a slow uptake of visual languages in the industry because the majority of experienced programmers are less willing to learn a completely new
language. On top of this, visual languages are usually tied to a specific platform, or a single programming language vendor. This limits the uptake and coverage of the language. For a programmer, experienced or not, to learn a new language, that language must present a large number of benefits to the programmer above the languages that that programmer currently uses. However, programmers are more likely to accept visual languages for problems requiring specialized languages because visual languages are usually easier and quicker to learn than many specialized textual languages. The same can be applied to assembly languages. Different microcontrollers have their own instruction set which a programmer needs to learn before writing a program.
8 References


http://www.csacademy.com/automation/faq/primer/


Appendix A: Source code of CoreChart

The source code is available as portable document format on the CD. Located in the ‘PDF Files’ folder, named ‘CoreChart Source Code.pdf’

The printout is organised by files, listed below:

**Forms:** Form files in Visual Basic contain both the graphical and textual elements. A program dialog is created by placing elements like buttons or textboxes into a form, them writing event codes for the individual elements. Below is a list of forms in the CoreChart source code.

<table>
<thead>
<tr>
<th>Form Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPORTDLG.FRM</td>
<td>Export Subroutines Dialog, used for exporting subroutines, contains code for listing available subroutines and generating subroutine modules.</td>
</tr>
<tr>
<td>FRMABOUT.FRM</td>
<td>About Dialog, displays the about box, mainly informative.</td>
</tr>
<tr>
<td>RMASM2BST.FRM</td>
<td>ASM to CoreChart Dialog, contains code that reads an assembly file and imports into CoreChart.</td>
</tr>
<tr>
<td>FRMASMVIEW.FRM</td>
<td>ASM View Dialog, contains code that formats the CoreChart generated assembly file and displays it on the screen.</td>
</tr>
<tr>
<td>FRMCOMSTAT.FRM</td>
<td>Assembly Status Dialog, parses the .err file generated by MPLAB assembler and displays any error messages.</td>
</tr>
<tr>
<td>FRMCONST.FRM</td>
<td>Constants Dialog, allows user to add or delete program constants.</td>
</tr>
<tr>
<td>FRMHOPTIONS.FRM</td>
<td>Hardware Settings Dialog, provides a dialog for user to change programmer and port settings.</td>
</tr>
<tr>
<td>FRMSETDISSZ.FRM</td>
<td>Preferences Dialog, allows user to change various UI settings.</td>
</tr>
<tr>
<td>FRMSPLASH.FRM</td>
<td>Splash Screen, a small dialog that is displayed when the program starts up, displays information while the program loads in the background.</td>
</tr>
</tbody>
</table>
**FRMTXTBOX.FRM**  
TextBox Dialog, provides a larger area for user to edit textboxes.

**IMPORTDLG.FRM**  
Import Subroutine Dialog, lists available subroutine modules for user to import.

**LBLDIALOG.FRM**  
Labels Dialog, allows user to view/edit labels in the current program.

**MAINFORM.FRM**  
Main Program Window, the main window. It is the first window displayed after the splash screen.

**PPROGDLG.FRM**  
Print Progress Dialog, displays the progress of printing.

**SUBDIALOG.FRM**  
Subroutines Dialog, displays a list of subroutines available in the current program.

**VARDIALOG.FRM**  
Variables Dialog, displays a list of variables declared by the user.

**Modules**: Module files are mainly used to group functions or procedures that are not specifically linked to any events and are called from different forms. Module files also contain API declarations and also global type or variable declarations.

**BANKING.BAS**  
Contains procedures to generate code to automatically switch banks for variables.

**CLIPBOARDEXPORT.BAS**  
Contains procedures to export clipboard contents so that it can be copied across different instances of CoreChart.

**COMPATIBILITYMODULE.BAS**  
Provides code to load older versions of Save file.

**DISPLAYICONS.BAS**  
Contains code to format the icons and properly displaying them on the main window work area.

**EASYREADWRITEREGISTRY.BAS**  
Contains functions to read and write values into and from windows registry.

**EDITFUNCMODULE.BAS**  
Contains procedures for the editing functions. i.e. Copy, Cut, Paste, Delete, Undo.
<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILEASSOC.BAS</td>
<td>Contains code to associate .bst files in Windows file system with CoreChart.</td>
</tr>
<tr>
<td>GENERALPROCEDURES.BAS</td>
<td>Contains procedures and functions that is not grouped.</td>
</tr>
<tr>
<td>ICONPROPMODULE.BAS</td>
<td>Contains code to generate selection lists for individual icon properties.</td>
</tr>
<tr>
<td>LABELFORMATMODULE.BAS</td>
<td>Contains code to format the labels displayed in the icons</td>
</tr>
<tr>
<td>REGUNREGSVR.BAS</td>
<td>Used to register/unregister shared libraries, mainly used for loading</td>
</tr>
<tr>
<td></td>
<td>configuration files.</td>
</tr>
<tr>
<td>SAVELOADMODULE.BAS</td>
<td>Contains code for saving and loading program files.</td>
</tr>
<tr>
<td>TXTPRINTMODULE.BAS</td>
<td>Contains code for converting the visual program into a pseudo text format</td>
</tr>
<tr>
<td></td>
<td>and printing it out.</td>
</tr>
</tbody>
</table>

**Classes:** Classes are definitions of objects and is used to define custom objects used in Visual Basic programs.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPROGRAM_INTERFACE.CLS</td>
<td>An abstract class that defines the structure for storing the</td>
</tr>
<tr>
<td></td>
<td>visual program.</td>
</tr>
<tr>
<td>ICONCLUSTER.CLS</td>
<td>A class that implements the structure for storing the visual</td>
</tr>
<tr>
<td></td>
<td>program.</td>
</tr>
<tr>
<td>VARIABLE_INTERFACE.CLS</td>
<td>An abstract class that defines the structure for storing</td>
</tr>
<tr>
<td></td>
<td>variables.</td>
</tr>
<tr>
<td>VARIABLECLUSTER.CLS</td>
<td>A class that implements the structure for storing variables.</td>
</tr>
</tbody>
</table>
Appendix B: Source code of Shared Library

The shared library was also written in Visual Basic (as an Active X DLL). A new library is normally generated by using an existing library as a template and modifying it. It was then compiled into a binary DLL file and then renamed with a ‘CFG’ as extension.

The shared library provides the following external interfaces that can be accessed by CoreChart during runtime.

**External interfaces:**

**Properties:**
- Name (Read Only)
- ICProgSet (Read Only)
- Version (Read Only)
- MaxUVar (Read Only)
- GPRAddSt (Read Only)
- ASMString (Read Only)
- ASMHeader (Read Only)
- INCFile (Read Only)
- IOPrintout (Read Only)
- IObitStr (Read/Write)
- CFGBitStr (Read/Write)
- OPTBitStr (Read/Write)

**Methods:**
- SetDefault( )
- ShowCfgDialog( )

Note: A Copy of the source code is available on the CD in portable document format, inside the folder: “PDF Files”.
Appendix C: Pseudo Code printout

The following pages contain a pseudo code printout of the CoreChart program shown below:

Note: A Copy of the printout is available on the CD in portable document format, inside the folder: “PDF Files”.

[Diagram of pseudocode flowchart]
Program Information

Main Program: 16 Icons
(Sub) INTERRUPT: 2 Icons
(Sub) DELAY: 7 Icons

Total number of Icons: 25

User Variables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Y</td>
</tr>
<tr>
<td>33</td>
<td>X</td>
</tr>
</tbody>
</table>

2 Refs
2 Refs

Total of 2 User Variables

Chip Settings

Configuration String (13:0)....... 1111100010000
Option String (7:0)............... 10000000
Port IO Settings:
  Port A (7:0).................... .11110111
  Port B (7:0).................... .00000000

Main Program

1. START:
2. Assign: W = 7
3. Assign: CMCON = W
4. Assign: STATUS = 0
5. Set: STATUS Bit C ON
6. Label: here
7. IF: PORTA Bit 5 ON Then
8. Calc: PORTB = << PORTB
9. IF: PORTA Bit 1 ON Then
10. Calc: PORTB = >> PORTB
11. Set: PORTA Bit 3 OFF
12. IF: STATUS Bit C ON Then
13. Set: PORTA Bit 3 ON
14. Call: DELAY
15. GoTo: here
16. END:
Subroutines

**INTERRUPT**

1. INTERRUPT: 
2. Return: From Interrupt

**DELAY**

1. DELAY: 
2. Label: loop
3. Count: X = X - 1, Skip if 0
4. GoTo: loop
5. Count: Y = Y - 1, Skip if 0
6. GoTo: loop
7. Return: From Subroutine
Appendix D: Survey tasks

This section contains material used in the practical session conducted to assess the usability of the VPL.

Included Corechart program Task1.bst
Task 0 – Programming the elab628

Step 1: Connecting elab628 to PC

- Connect the supplied DB9 Serial Cable from the COM port connector on the elab628 Programmer to an available COM port on your computer. In most cases the other end of the cable should already be connected to a COM port on your computer, you just need to connect the serial cable to the elab628 board.

- Make sure the elab628 is in the Programming Mode (PROG) by switching the program/run switch towards the PROG on the board. The switch needs to be in program mode to be able to program the Microcontroller.
**Step 2: Programming the elab628**

1. Make sure the CoreChart Program is installed and run CoreChart.

2. On the left hand top menu select “File”. Position the pointer onto **Open** and left click the mouse. Locate “Task0.bst” in “C:\Survey Tasks” and double click on the file to open it. This is a simple program that flashes the first LED “TS0” periodically.

3. You will now see that the program “Task0.bst” is loaded in CoreChart. Basically, the program just turns the LED connected on Port B Bit 0 on and off. It calls a Delay subroutine in between the ONs and OFFs so that it doesn’t switch the LED too fast until it is unnoticeable by the human eye. Because the microcontroller is capable of executing 1 million instructions (or roughly the same number of icons) per second. Without the delay subroutine to slow things down, the LED would turn on and off thousands of times a second. This would be impossible to see.

4. Go to “Options” and click “Send To Pic”. The Program will now transfer the Sample program “Task0.bst” to your Hardware.

Note: Be sure that there is no tick beside the option “Show Full IC-Prog”. If there is a tick, move your mouse to highlight the “Show Full IC-Prog” and click on it once to turn off the tick.
5. To test your elab628 board, Switch the elab628 board into Run Mode by sliding the program/run switch towards the ‘RUN’ label on the board. Make sure power is connected to the board or the board is plugged into the serial port. If the program is running you will notice that the LED “TS0” will flash periodically.

If you have any problems, check the hardware and check the steps above carefully. Otherwise you are ready to proceed on to the next task.
Task 1 – LED Chasers

Objective of this task is to modify the program in Task 0 such that the LEDs blink sequentially, this will give the visual effect of the LED travelling or chasing.

```
... → ... → ... → ... → ... → ...
```

The Pseudo code for this program:

<table>
<thead>
<tr>
<th>Codes</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>START:</td>
<td>; Start Label</td>
</tr>
<tr>
<td>Declare LED as Byte</td>
<td>; Declare a variable</td>
</tr>
<tr>
<td>LED = 1</td>
<td>; Set initial value, LED = 1</td>
</tr>
<tr>
<td>Clear carry bit</td>
<td>; We need to clear carry bit because we are going to use the rotate instruction to produce the LED chasing effect</td>
</tr>
<tr>
<td>LOOP1:</td>
<td>; A Label for Goto instruction</td>
</tr>
<tr>
<td>Port B = LED</td>
<td>; This effectively lights up the LEDs according to the values of variable ‘LED’. In the first loop, LED 0 would light up.</td>
</tr>
<tr>
<td>Delay</td>
<td>; Wait a while such that the switching is noticeable</td>
</tr>
<tr>
<td>Rotate LED 1 bit to the left</td>
<td>; This will make ‘00000001’ → ‘00000010’</td>
</tr>
<tr>
<td>Goto LOOP1</td>
<td>; go back to execute instruction after label LOOP1</td>
</tr>
</tbody>
</table>

You will need to implement the above program in CoreChart by modifying the existing program (Task0.bst). We will be reusing the Delay subroutine that is available in the program. In this exercise we need to know how to declare a variable, Add/Delete icons from the program, assigning values to a variable, how to access the carry bit, and using the rotate instruction. Read thru the instructions in the following pages to familiarise yourself with the commands needed in the program. Then, begin modifying the program to make the LEDs flash in sequential order. Below are a few simple steps to guide you:

- Start by saving the program as a different name. Use File → Save As… (or press F8) then type in a new name (i.e. Task1.bst) and left-click on save.

- Delete all the icons which are not needed. We only need one ‘call delay’ icon, and one ‘Set On/Off’ Icon. We also need to declare the variable LED etc…

- Start adding icons and modifying its properties so that its function matches with the pseudo code given above.

- When you are done, save the program. Make sure the elab628 board is connected to the PC via the serial cable, and ensure that the Program/Run switch is set on Program. Then select ‘Send to Pic’ under the options menu (or press F9) to download the program onto elab628 board.

- After the program has successfully downloaded, slide the Program/Run switch to run, if the program is written correctly you should see the LEDs flashing sequentially.
Declaring Variables

To declare a variable in CoreChart, you will need to access the User Variable List Dialog. The dialog can be accessed via the drop down menu Edit → Variables... or by pressing the Ctrl+A shortcut key.

You will then see the User Variable List Dialog, shown in the above picture. To declare a variable, type in the variable name in the textbox (1), then click the ‘Add Variable’ button (2). To rename or delete a variable, select the variable on the list (3), then click on ‘Rename Var’ (4) to rename the variable or ‘Delete Variable’ (4) to delete the variable. To exit the dialog, click on the ‘Close’ (5) button.

Note: You will only be able to use the variable after you declared it in the user variable list.

Add/Delete ICONs

To delete icons, simply select the icons you want to delete by clicking on it, then click on the delete button to delete the icon (X). You will be asked for confirmation to delete the icon, select OK to delete the icon. Multiple Icons can be selected and deleted by pressing (and holding down) the shift key while selecting the icons.

To add or insert an icon, first select the icon you want to add on the icon toolbar by left clicking on the icon. You will notice that the icon that is currently selected is pressed in. Next move the cursor to a location which you want to insert the icon. When you move the cursor to the space in between the 2 icons (as show in the picture to the right), a box will appear selecting the connecting arrow between the 2 icons. Left-clicking the mouse will insert the icon in between the 2 icons.

Modifying ICONs

To modify an icon, select the icon you want to modify by left clicking on it. The icon properties will be displayed on the lower right corner of the program (as shown in the picture on the right). Change the properties of the icon by selecting the values from the drop down list or by typing the values in. After when you are done, click on the ‘Apply Changes’ button to save the changes to the icon and update the main display screen.
Assigning values to a variable

The Assignment icon (\( \text{Assignment} \)) is used to assign values to a variable. Select the variable you want to assign values to in the first listbox (1), then either type in the value you want to assign in the second listbox (2) or select from a list of variables whose values you want to assign to the first variable.

Special Function Registers (Carry Bit)

The special function registers are registers used by the Microcontroller for controlling the desired operation of the device. Some examples of special function registers are:

- **PORTA**: Provides access to the Input/Output port A
- **PORTB**: Provides access to the Input/Output port B
- **STATUS**: Contains the arithmetic status of the ALU, the RESET status and the bank select bits for data memory (SRAM).

The carry bit (C) is located in the STATUS register and can be accessed directly. (Z, DC & C are arithmetic status bits)

<table>
<thead>
<tr>
<th>IRP</th>
<th>RP1</th>
<th>RP0</th>
<th>TO</th>
<th>PD</th>
<th>Z</th>
<th>DC</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 7</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
<td>Bit 0</td>
</tr>
</tbody>
</table>

Bit 0 (C) is the carry bit. A ‘1’ means there is a carry out from the most significant bit of the previous arithmetic operation.

To access the individual bits of the special function register, we use the Set/Reset Icon (\( \text{Set/Reset} \)). Using this icon, we can set the individual bits of the variable on or off. In this exercise, we will set the Carry bit in the STATUS register to OFF.

Rotating a Variable (bitwise)

We ‘rotate’ a variable by shifting the bits in the variable to the left or right. The bits are actually rotated through the Carry bits (illustrated in figure below), this is why we need to clear the carry bit prior to using this function. This function is implemented in the Calculate Icon (\( \text{Calculate} \)).

To use the rotate function in the calculate icon, first select the variable you want to rotate bit-wise (1), then select nothing on the next drop-down list (2). You will notice that there is a new set of operators in drop down list (3). Select the operator ‘\( \ll \)' which is the rotate left function. The variable in the box (4) will automatically change to equal the variable in box (1).
Task 2 – User Inputs and Decision ICONs

In this task we will learn how to get obtain inputs from the user via the Input/Output ports of the elab628, and get to know about how CoreChart makes decisions. In this exercise we will be modifying the program we have written in Task1 such that it will flash the LEDs in reverse sequence when a button is pressed.

Shown below is the Pseudo code from the previous exercise, with modifications (Shown in red) to accept inputs from PORTA Bit 5 (where the push button is connected) and make a decision based on the input.

<table>
<thead>
<tr>
<th>Codes</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>START:</td>
<td>; Start Label</td>
</tr>
<tr>
<td>Declare LED as Byte</td>
<td>; Declare a variable</td>
</tr>
<tr>
<td>LED = B'00000001'</td>
<td>; Set initial value, Note this method is used to assign binary values B'00000001' to the variable LED</td>
</tr>
<tr>
<td>Clear carry bit</td>
<td>; We need to clear carry bit because we are going to use the rotate instruction to produce the LED chasing effect</td>
</tr>
<tr>
<td>LOOP1:</td>
<td>; This effectively lights up the LEDs according to the values of variable 'LED'. In the first loop, LED 0 would light up.</td>
</tr>
<tr>
<td>Port B = LED</td>
<td>; Just a Label for Goto instruction</td>
</tr>
<tr>
<td>Delay</td>
<td>; Wait a while such that the switching is noticeable</td>
</tr>
<tr>
<td>IF PORTA Bit 5 is ON then</td>
<td>; Will execute the next line if true, else skip next line</td>
</tr>
<tr>
<td>Rotate LED 1bit to the left</td>
<td>; This will make ‘00000001’ → ‘00000010’ and so forth</td>
</tr>
<tr>
<td>IF PORTA Bit 5 is OFF then</td>
<td>; Will execute the next line if true, else skip next line</td>
</tr>
<tr>
<td>Rotate LED 1bit to the right</td>
<td>; This will make ‘00000001’ → ‘10000000’ and so forth</td>
</tr>
<tr>
<td>Goto LOOP1</td>
<td>; Loop forever</td>
</tr>
</tbody>
</table>

Begin by reading thru the information provided below about how CoreChart makes decisions. Then start modifying the previous program to accept inputs from PORTA. Use the Pseudo code above as a guideline. Note the only changes in the program from the previous task are highlighted in red.

Save the program as ‘Task2.bst’ then download it onto elab628 board. If you have done it correctly the chasing LEDs will change directions when you press and hold the button on the elab628 board. It will revert back to the original direction when you are not pressing the button.

How CoreChart makes decisions

There are 2 decision making icons in CoreChart, the Decision Icon ( ) and Count and Decide Icon ( ). The Decision Icon works by checking if a particular bit is ON or OFF. If the expression is true, it will execute the next line. If it is not true, it will skip the next line. The Count and Decide Icon works a bit different. There is a variable which will be increased or decrease by 1 each time it is run and when the variable becomes zero, it will skip the next line. We will only be using the Decision Icon in this exercise.
Appendix E: CD Rom containing CoreChart

Note:
The CD-Rom does not auto-run. To install CoreChart, execute ‘CoreChart_Setup.exe’ located in the root directory of the CD. This CD also contains the modified version of CoreChart that produces PIC18F microcontroller assembly code (folder 'CoreChart PIC18C'). The program does not come with an install package and can be run directly by double-clicking on the 'CoreChart.exe'. However, CoreChart should be installed prior to running the program to ensure that all the required windows libraries are installed.