LEARNING TO PROGRAM WITH VISUAL BASIC AND .NET GADGETEER

A guide to accompany the Fez Cerberus Tinker Kit

Sue Sentance
Steven Johnston
Steve Hodges
Jan Kučera
James Scott
Scarlet Schwiderski-Grosche

LEGAL NOTICE: The source code available in this book is subject to the Apache License, version 2.0. To view a copy of this license, visit http://www.apache.org/licenses/LICENSE-2.0.html.

All other content is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Edition 1.0, 22 October 2013

FOREWORD

Computer programming can be fun! This book on Visual Basic and .NET Gadgeteer shows how. Aimed at high school students and first-time programmers, the authors use a combination of hardware and software to make programming come alive - audibly, visually, and tangibly. Using small hardware components - a standalone processor, simple sensors and actuators - students build their own little computers in hardware and then program them to do fun things, such as play music (Chapter 3), blink lights (Chapter 6), or draw pictures (Chapter 10). By the end of the book, students have learned all the basics of programming: variables, basic data types, arrays, conditionals, iteration, procedures, and functions. More importantly, they learn a fundamental "computational thinking" concept - modularity. From the very first exercise, students become engaged through the tactile experience of assembling hardware components together to build real devices which they program through standard interfaces, protocols, and built-in libraries. Without a lot of fuss, the authors teach these concepts using the widely-adopted Visual Studio software development tool, exposing students to a modern programming environment that supports the engineering cycle of design, build, test, and debug. At the same time, students naturally assimilate a better understanding of how electronic devices work and how they are made, valuable skills in our increasingly digital lives.

Jeannette M. Wing
Corporate Vice President, Microsoft Research
15 October 2013

ACKNOWLEDGEMENTS

The authors would like to acknowledge several others who have been invaluable in creating the material contained within this book.

First and foremost we would like to thank our colleagues Clare Morgan and Nicolas Villar: Clare has been instrumental in facilitating school outreach and supporting the team; Nicolas is the original inventor of the platform without whom we would have no .NET Gadgeteer. We have worked with a number of Gadgeteer hardware manufacturers during this project, but we would particularly like to call out GHI Electronics who were excited to explore how Gadgeteer could be used in the classroom from the outset of our work and who tailored a kit of Gadgeteer parts to support the learning points and exercises we wanted to cover. We are extremely grateful to the young people who have spent dedicated time working through the book, testing the exercises, and giving us their invaluable feedback, including: Ellen Curran, Thomas Denney, David and Jonathan Goh, Armin Grosche, Maeve McLaughlin, Alistair Sentance, and Alfie Sharp. Finally, we owe a great deal to the hundreds of students and educators who have embraced Gadgeteer in their classrooms over the past two years. In a world where it is all too easy to become a consumer of digital technologies, we hope that this book will inspire a new generation of digital creators!

AUTHORS

Sue Sentance works for Computing At School (CAS), the professional association in the UK for Computer Science school teachers. Her work revolves around bringing more Computer Science teaching into schools, and she has been developing teaching materials for schools using .NET Gadgeteer since its launch in 2011. She has worked in schools as a teacher of Computing and as a lecturer in Initial Teacher Education, and has a PhD from the University of Edinburgh.

Steven Johnston is a program manager on the .NET Gadgeteer project at Microsoft Research and has a PhD from the University of Southampton where he is also a Senior Research Fellow in the Faculty of Engineering and the Environment. Much if his work involves exploiting new, and up and coming technologies and applying them to the field of engineering.

Steve Hodges leads the Sensors and Devices research group at Microsoft Research and is also a visiting Professor at Newcastle University. His work centres around new tools and technologies for prototyping and fabrication, new ways of interacting with computer systems, and wearable devices. He has a PhD from the University of Cambridge and is a Fellow of the Institution of Engineering and Technology.

James Scott is a researcher in the Sensors and Devices group at Microsoft Research. His research interests span a wide range of topics in ubiquitous and pervasive computing, including novel devices and sensors, rapid prototyping, mobile computing and interaction, and security and privacy. He has a PhD from the University of Cambridge. He is one of the creators of the .NET Gadgeteer platform.

Scarlet Schwiderski-Grosche is a senior research program manager in the Microsoft Research Connections team. She is responsible for academic research partnerships relating to .NET Gadgeteer and other Microsoft Research projects, and drives liaison with a number of joint research centres. Scarlet has a PhD in Computer Science from University of Cambridge and worked in academia for almost 10 years before joining Microsoft in 2009.

TABLE OF CONTENTS

Chapter 1. Introduction 7
Objectives of this book 8
Modules used in this book 8
How to use this book 10
Chapter 2. Getting started with .NET Gadgeteer 11
Assembling the hardware 11
Starting a project in .NET Gadgeteer 11
The Gadgeteer Designer in Visual Studio 13
Chapter 3. Playing tunes 17
Overview 17
Programming in Visual Studio 17
Tutorial: Playing a tune 18
Exercises 22
Summary 22
Chapter 4. Clicker 23
Overview 23
New concepts: What is a variable? 23
Tutorial: Building a clicker 24
Exercises 28
Summary 28
Chapter 5. Stop watch 29
Overview 29
New concepts: If … Then … Else… statement 29
Tutorial: Building a stop watch 29
Exercises 33
Summary 34
Chapter 6. Traffic lights 35
Overview 35
New Concepts: More on If statements and the Select…Case statement 35
Tutorial: Traffic lights 36
Exercises 40
Summary 40
Chapter 7. Counting in binary 41
Overview 41

New concepts: Binary numbers 41
New concepts: The For loop in Visual Basic 42
Tutorial: Counting in binary 43
Exercises 48
Summary 48
Chapter 8. Burglar alarm 49
Overview 49
Tutorial 1: Creating a burglar alarm 49
Exercises 51
New concepts: Saving to the SD card 51
Tutorial 2: Keeping a record of an intrusion 52
Exercises (Continued) 55
Summary 55
Chapter 9. Morse code 57
Overview 57
New concepts: Reading from a file in Visual Basic 57
New concepts: Creating a function in Visual Basic 57
Tutorial: Morse code 58
Tutorial: Reading a word from a file and translating to Morse code 60
Exercises 64
Summary 64
Chapter 10. Drawing 65
Overview 65
Tutorial 1: House draw 65
New concepts: Arrays in Visual Basic 67
New concepts: Passing parameters into procedures and functions – By Val and By Ref 69
Tutorial 2: Graph draw 69
Exercises 72
Summary 73
Chapter 11. Reaction game 75
Overview 75
New concepts: Working with random values 75
New concepts: Logical operators 76
Tutorial: Building the reaction timer game 76
Exercises 83
Summary 83
Chapter 12. Build your own module 85

Overview 85
Tutorial 1: Make a noise! 85
Tutorial 2: Use the joystick to control the pitch and duration 88
Exercises 89
Summary 90
Appendix A . Where to buy .NET Gadgeteer 91
Appendix B . Installing the "FEZ Cerberus Tinker Kit" 92
Appendix C . Getting to know Visual Studio 94
The Solution Explorer Window 94
The Toolbox window 94
The Designer window 95
The Output window 95
IntelliSense in Visual Studio 96
Appendix D . Updating the firmware 98
Checking the firmware version 98
Updating the FEZ Cerberus Firmware using the FEZ Config Tool (recommended) 100
Updating the FEZ Cerberus TinyBooter manually 103
Updating the FEZ Cerberus TinyCLR manually 105
Appendix E . How to debug Visual Basic in Visual Studio 107
Printing debug messages 107
Setting a breakpoint 107
The conditional breakpoint 109
Walking through your code 110
Debugging tips 112
Appendix F . Coping with Out-of-Memory exceptions 114
Appendix G . Troubleshooting: Visual Studio will not deploy 115
Appendix H . List of current socket types 120
Appendix I . Operators in Visual Basic 123
Appendix J . Typical Lux values 124
Appendix K . Resources available online 125

CHAPTER 1. INTRODUCTION

Microsoft .NET Gadgeteer is a really easy-to-use platform for creating new electronic devices using a wide variety of hardware modules and a powerful programming environment. Students with little or no electronics background can design and build devices that sense and react to their environments using components such as switches, displays, buzzers, motor controllers and more. Using cables these various modules are plugged into a mainboard which is programmed to make everything work together.

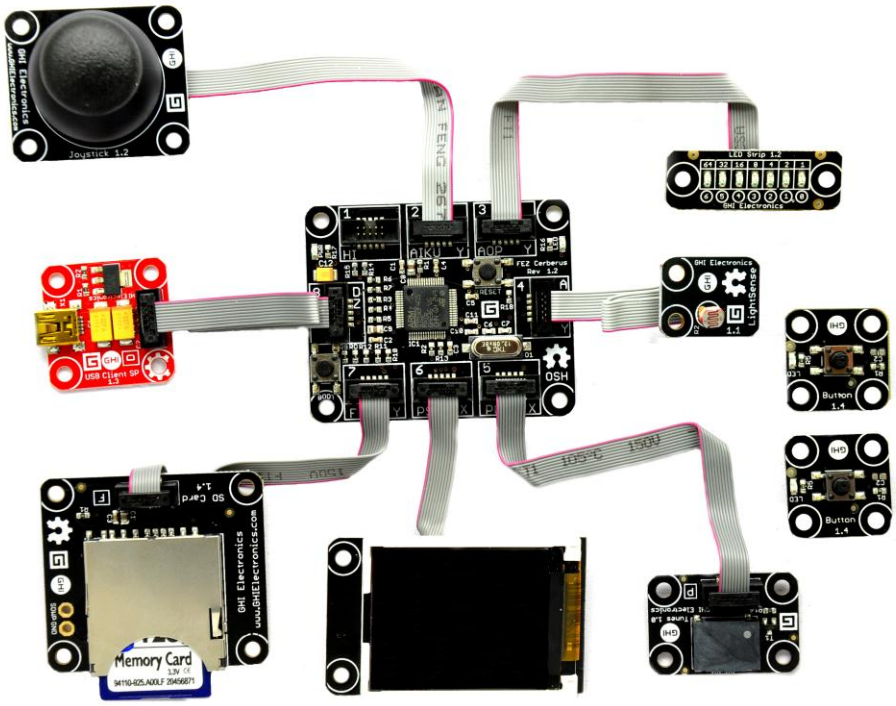


Figure 1: FEZ Cerberus Tinker Kit, showing mainboard and modules.

.NET Gadgeteer originated at Microsoft Research in Cambridge, U.K. It was initially designed as a tool for researchers to invent new kinds of devices more quickly and easily. It very quickly proved to be of interest to hobbyists and for secondary and higher education. In particular, several pilots in schools in the UK and the US with students ranging from 11 to 18 years of age demonstrated that .NET Gadgeteer is a motivating environment for teaching computer programming both within the curriculum and as an extra-curricular club. Gadgeteer is programmed in a modern event-based and object-oriented style using Visual C# or Visual Basic. In addition to teaching programming, Gadgeteer offers exciting possibilities for teaching electronics and computer-aided design.

Microsoft Research launched .NET Gadgeteer as an open platform in 2011. As a result .NET Gadgeteer components are now available from several hardware vendors, whilst the software required to program the devices is available for free from Microsoft. The Fez Cerberus Tinker Kit from GHI Electronics consists of a mainboard and various modules which have been selected to allow someone new to Gadgeteer to build a wide variety of exciting devices. Other Gadgeteer kits are available and there are also a wide variety of individual modules which are available separately.

With .NET Gadgeteer we hope that we can give students a better understanding of how the devices and technology all around us work, as well as the skills to create their own. We hope to inspire a future generation of producers of electronic devices, not just consumers!

OBJECTIVES OF THIS BOOK

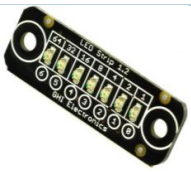
This book is intended for school students and others learning to program in Visual Basic. It assumes no prior knowledge of programming, electronics, Visual Basic or the Visual Studio environment. Programming concepts are introduced and explained throughout the book. Each chapter is structured in a similar way: firstly a new concept to be learned is introduced, secondly there is a step-by-step tutorial on how to develop a simple example in Gadgeteer which uses that concept, and finally a set of exercises is given which enable the reader to practise the main points. This last step is the most important: when learning to program, practising new programing skills is crucial! Answers to all the exercises are available at http://gadgeteering.net.

This book is based on the Fez Cerberus Tinker Kit made by GHI Electronics, which is a good value-for-money .NET Gadgeteer kit produced specifically for education. Although images of this kit are used throughout this book, different mainboards and modules from other manufacturers or kits can be substituted with little or no changes to the associated code.

MODULES USED IN THIS BOOK

This book uses the following modules, which are all included in the FEZ Cerberus Tinker Kit apart from the 'Extender' module in Chapter 12. Note that in many cases other kits and modules can be substituted.

Module name	Image	Used in chapters
Fez Cerberus mainboard This is at the heart of every project. It has a processor, memory and a series of sockets. You can connect modules into these sockets.		All
USBClientSP power module The mainboard and modules require power. This module not only provides power to your device it also lets you program and debug over USB.		All
Display N18 module This LCD module lets you display images and text in colour. It is backlit and is visible in the dark. The display is 128 pixels wide and 160 pixels high.		Chapter 4 Chapter 5 Chapter 10 Chapter 11

Joystick The joystick moves up-down and left-right. You can then read values corresponding to how far the joystick has been moved in each direction. You can also press the joystick and it acts like a normal button.		Chapter 11 Chapter 12
LED Strip There are 7 LED lights in a row: 2 red, 3 amber and 2 green. You can turn each LED on/off individually.		Chapter 6 Chapter 7 Chapter 8
Light sensor This module returns a value that corresponds to the amount of light to which the sensor is exposed.		Chapter 8
Button The module detects when the button is pressed and released and also has an LED which can be turned on/off.		Chapter 4 Chapter 5 Chapter 6 Chapter 7
SD card module This modules accepts a full sized SD card and lets you read from and write to files. You will need an SD card to use it; not all SD cards work so use the one provided with the Tinker Kit.		Chapter 8 Chapter 9
Tunes This module lets you play individual notes and construct basic melodies.		Chapter 3 Chapter 9
Extender This is an advanced module that lets you connect existing electronic components or custom devices to your project. You can use any extender or breakout module instead.		Chapter 12

HOW TO USE THIS BOOK

This book consists of a series of projects built using a Fez Cerberus Tinker Kit. There are 9 projects using the kit, with an additional project in chapter 12 which needs a Gadgeteer Extender module and a buzzer which you can purchase separately.

Chapters 1 and 2 introduce you to how to use .NET Gadgeteer and Visual Studio so you will need to read these first and make sure that you have installed Gadgeteer successfully and are familiar with the Visual Studio interface.

Chapters 3 to 12 each contain a project and are divided up as follows:

- **Key Terms** – these are the new concepts to be covered in this chapter;
- **Modules you will need** – which Gadgeteer modules you will need for this project;
- **Overview** – what you will build in this chapter;
- **New Concepts** – an explanation of the new programming concepts. Programming concepts are introduced in this order: variables, data types, if statements, loops, functions, arrays and file-handling;
- **Tutorial** – a step-by-step tutorial on how to develop the core project. Follow the instructions carefully, referring to the appendices if you have problems getting anything to work;
- **Exercises** – the most important part of each chapter! This is where you try to build some projects on your own, building on what you have learned in that chapter. Solutions to the exercises can be found at http://www.gadgeteering.net/learn;
- **Summary –** this gives some key points that we think you will have learned in the chapter.

The appendices cover a range of topics from where to buy Gadgeteer to what to do if your project will not work as expected. We recommend you read through the appendices to see the range of topics covered, and then refer to them when you have any difficulties or need additional information. Note that the technical details are correct at the time of writing but can change as new software releases or updates become available. In this case, updated information will be available at http://www.gadgeteering.net.

CHAPTER 2. GETTING STARTED WITH .NET GADGETEER

ASSEMBLING THE HARDWARE

Each project needs a mainboard. This contains the microprocessor which will be programmed to make it behave in the desired way. But a microprocessor only really becomes useful when it is connected to other components to let it display output, react to inputs, process files and so on. To do this with Gadgeteer, different modules are connected to the mainboard using small grey cables.

You will begin each project in this book by plugging together the modules needed for that particular project. There is a list of modules at the start of each chapter. It is important to plug the modules into the right sockets on the mainboard – each module has a letter on it which indicates which socket type it can be attached for. When you have found the socket letter on the module, look for the same letter next to a mainboard socket – there will probably be more than one option – and attach the other end of the cable there.

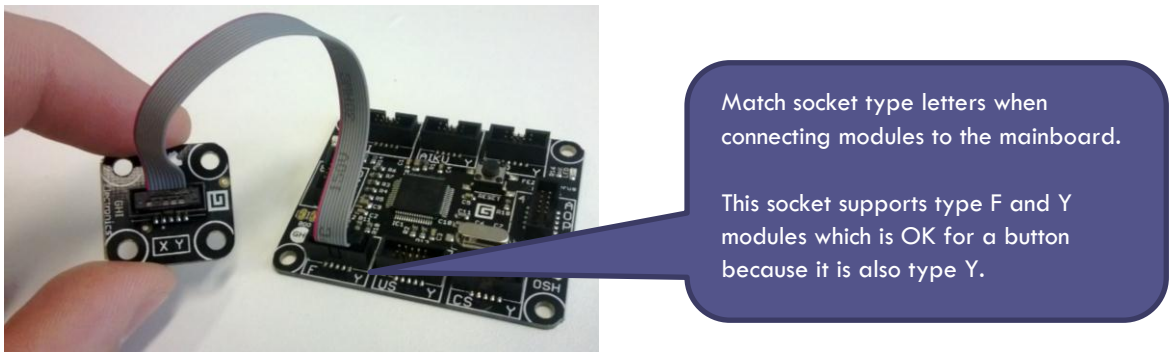


Figure 2: Assembling the hardware

The most important rules for plugging together Gadgeteer are:

- When connecting modules, always make sure that the mainboard is not connected to the PC or powered on using a different power supply.
- Red modules supply power to the mainboard. Only one red module should ever be connected to the mainboard at any time.
- Match the socket type letters between the modules and the mainboard as described above.

If a module is connected to the wrong socket type it won't work but it shouldn't damage anything either.

STARTING A PROJECT IN .NET GADGETEER

You need a desktop or laptop computer in order to program a Gadgeteer mainboard. Before starting to work with Gadgeteer, ensure that you have downloaded and installed all the software on to your machine. You need three different pieces of software:

- The Microsoft Visual Studio programming application for developing code;
- The Microsoft .NET Micro Framework system which allows your code to run on a Gadgeteer mainboard;
- A software development kit (SDK) from the manufacturer of the particular Gadgeteer components you are using.

All modern software is updated frequently by the companies who develop it, in order to fix any problems and add new features. The same is true of the Gadgeteer software components listed above and it's important that you use the right versions! There are currently two popular versions of Microsoft Visual Studio and you can use either one as long as you install the associated version of the .NET Micro Framework. The installers are pretty

easy to use. You must install the software in the order given below. Some of the installations involve downloading a zipped file. In this case you must extract all the contents before running the Setup program. For detailed installation instructions go to http://www.gadgeteering.net/content/installation-instructions and check Appendix B.

If you want to use Visual Studio 2012 edition, this is exactly what you need to get your Fez Cerberus Tinker Kit up-and-running:

- Visual Studio Express 2012 for Desktop;
- Microsoft .NET Micro Framework 4.3 SDK;
- GHI NETMF and Gadgeteer Package 2013 R2.

Alternatively you can use the following:

- Visual Basic 2010 Express;
- Microsoft .NET Micro Framework 4.2 SDK;
- GHI NETMF and Gadgeteer Package 2013 R2.

The Express Editions of both Visual Basic 2010 and Visual Studio 2012 are free, but if you already have another version of Visual Studio 2010 or 2012 which supports Visual Basic it is fine to use this instead. The rest of the software is also free.

To start using .NET Gadgeteer, first launch Visual Studio 2012 or Visual Basic 2010. On launching the software, you will see a screen enabling you to choose a .NET Gadgeteer Application (see Figure 3). If you do not see this, select NEW and then Project… from the FILE menu. You will need to make sure you select Visual Basic and then Gadgeteer on the left-hand side before you are offered the chance to select a new .NET Gadgeteer application as shown in Figure 3. The screenshots in this book all use Visual Studio 2012; if you are using Visual Basic 2010 things will look slightly different but all the principles are the same.

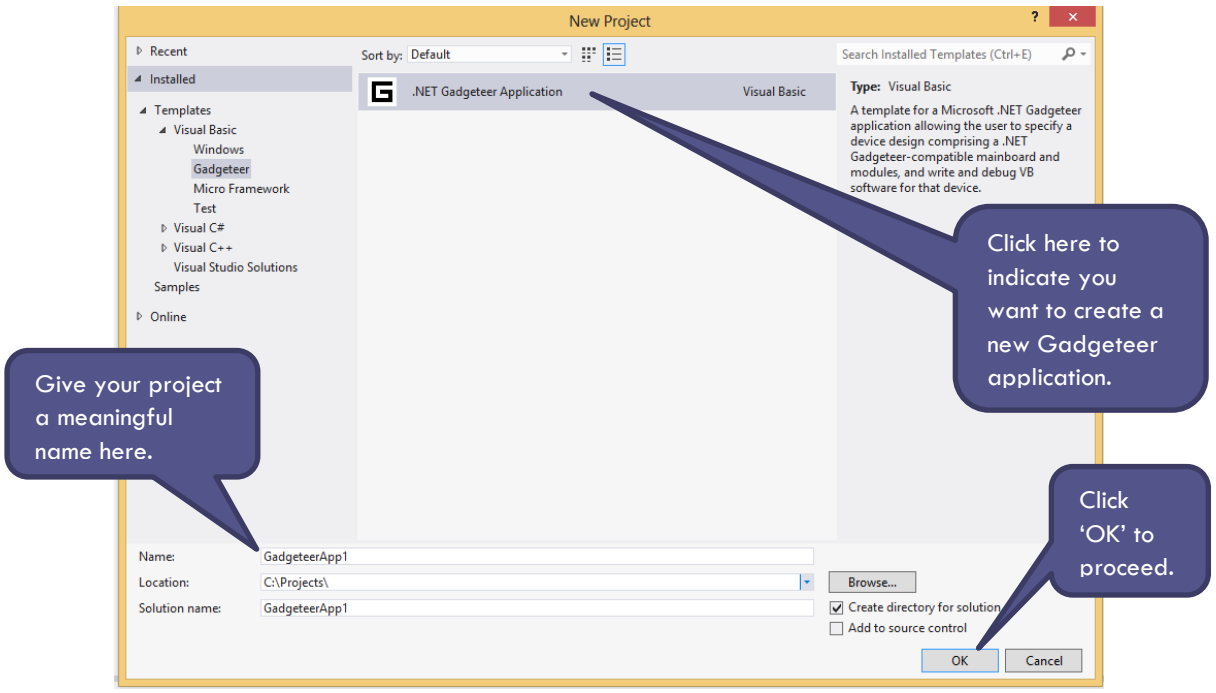


Figure 3: Starting to use .NET Gadgeteer

You are then prompted to choose the mainboard you want. There are many different mainboards available from different manufacturers but the Fez Cerberus mainboard comes with the Fez Cerberus Tinker Kit and this is what

we will be using in this book. So choose the Fez Cerberus mainboard as shown in Figure 4 and then click on 'Create'.

This will create a new project in Visual Studio and you are ready to start!

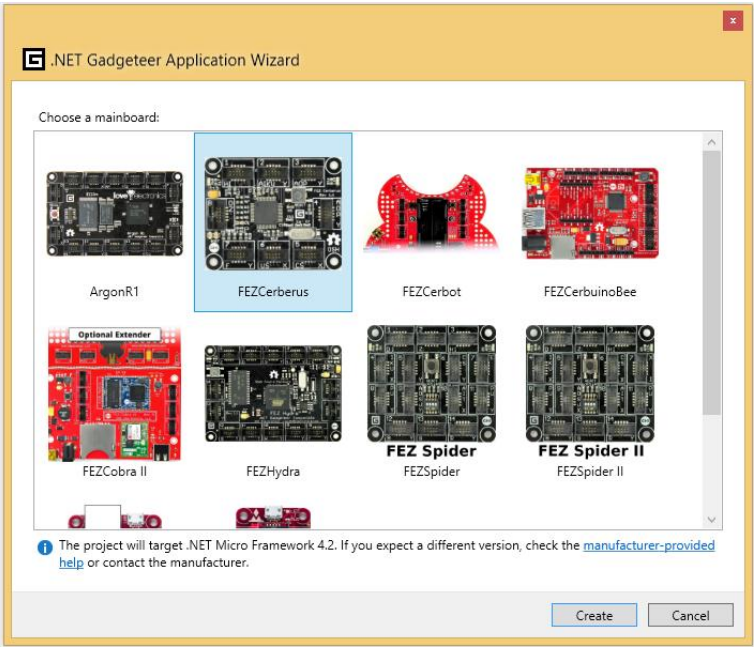


Figure 4: Choosing the Cerberus mainboard

THE GADGETEER DESIGNER IN VISUAL STUDIO

Before you start to write any code to run on your Gadgeteer mainboard, you need to specify which modules you are going to use in your new project and which sockets you are going to connect them to. This is done in Visual Studio using a feature called the Gadgeteer Designer. This Designer is a special picture-based view of your project – the idea is to drag photos of each of the modules you are using into the Designer window and then click on the module sockets to draw lines on the screen which connect up to sockets on the mainboard you are using.

When you created a new Visual Studio '.NET Gadgeteer Application' in the previous section you probably ended up with something like the screenshot in Figure 5. This is the Designer view, and to start with it only includes the mainboard you selected when creating your new project. If you ever want to return to the Designer view, look for the tab labelled 'Program.gadgeteer' and click on this.

The next stage is to add the modules you want to use for your project. This is done by selecting them from the Toolbox. The Toolbox is a window which runs down the left hand side of the screen in Visual Studio and lists all the modules that were included in the SDK you installed previously. If the Toolbox is not showing, click as shown in Figure 5 to reveal it. Bring the modules you need into the Designer window by dragging or double-clicking the relevant module names in the Toolbox. Then connect them to the mainboard.

The steps shown in the pictures in Figure 5, Figure 6, Figure 7 and Figure 8 will guide you through this process.

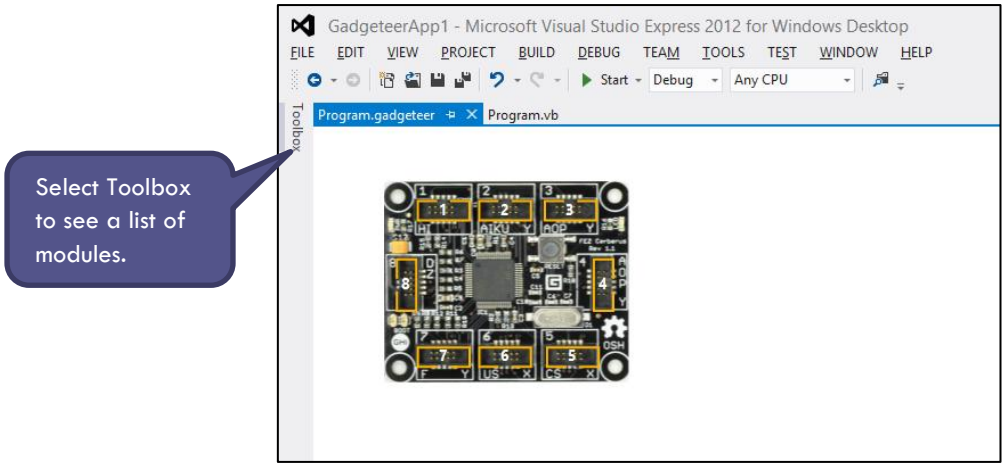


Figure 5: Using the Toolbox

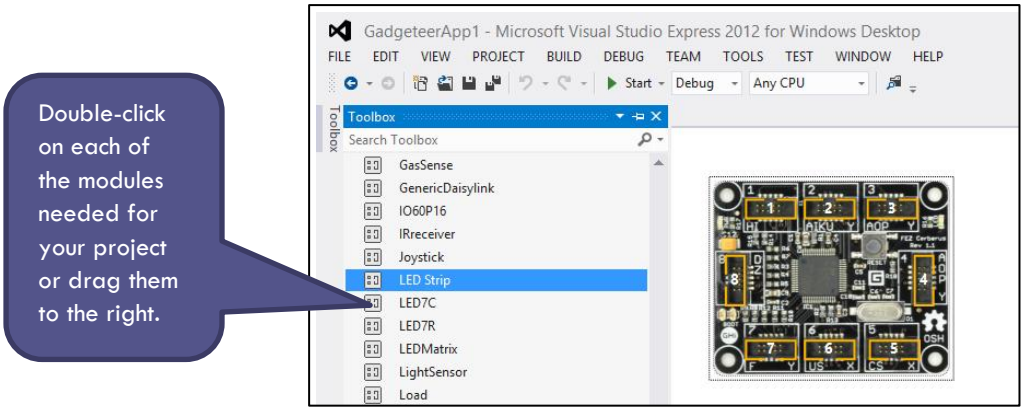


Figure 6: Adding modules in the Designer

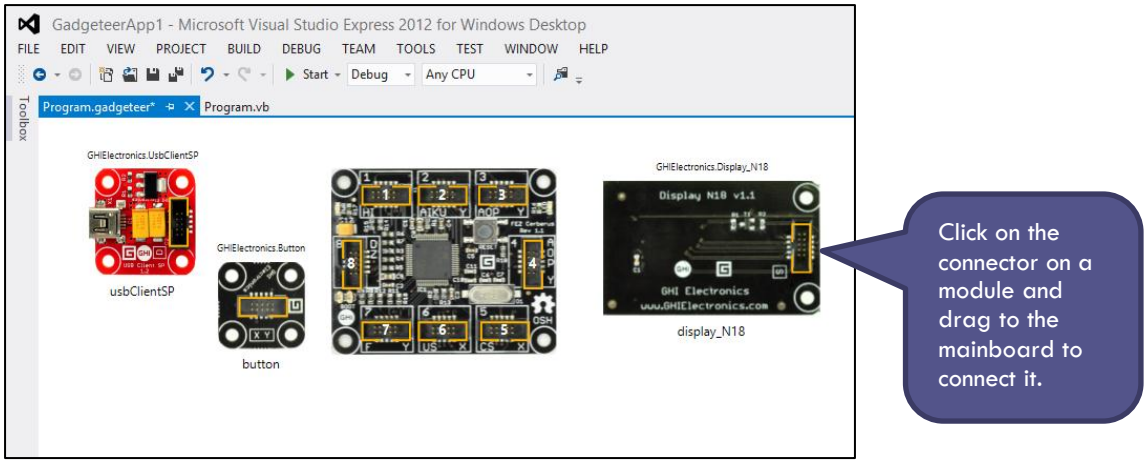


Figure 7: Modules in the Designer

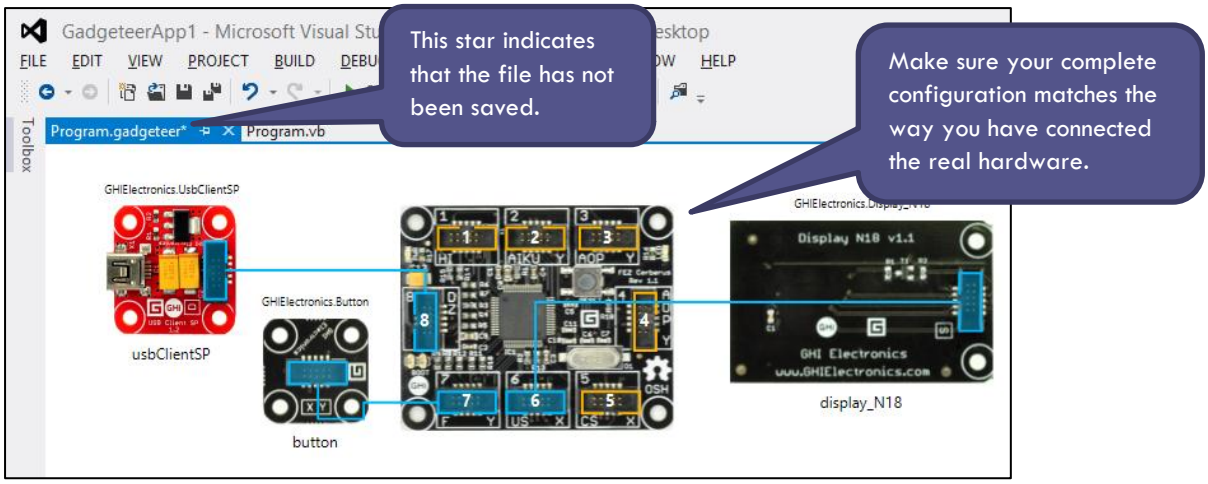


Figure 8: Connecting modules in the Designer

You will probably notice that when you 'wire up' a module in the Designer it is only possible to connect to a compatible socket on the mainboard – one with the same socket type letter.

When you have finished connecting the modules to the mainboard and your configuration matches your hardware, select Save All under the FILE menu to save the whole project. Connect the hardware to your computer using the USB cable. You are then ready to start programming your project. If you ever go back and make changes in the Designer be sure to save the file, there is an indicator on each tab showing if it has been saved as shown in Figure 8.

To start writing code for your project, click on the Program.vb tab at the top of the project (see Figure 9). This takes you away from the Designer and replaces it with a code window as shown in Figure 10. As you will see, Visual Studio automatically creates the framework for your code using a template which was installed with the Gadgeteer software. The template contains several lines of comments – text that can be included in a program in order to help programmers understand how a piece of code works but which the computer will ignore. These comments can be deleted of course, but they are often useful to help you get started with programming.

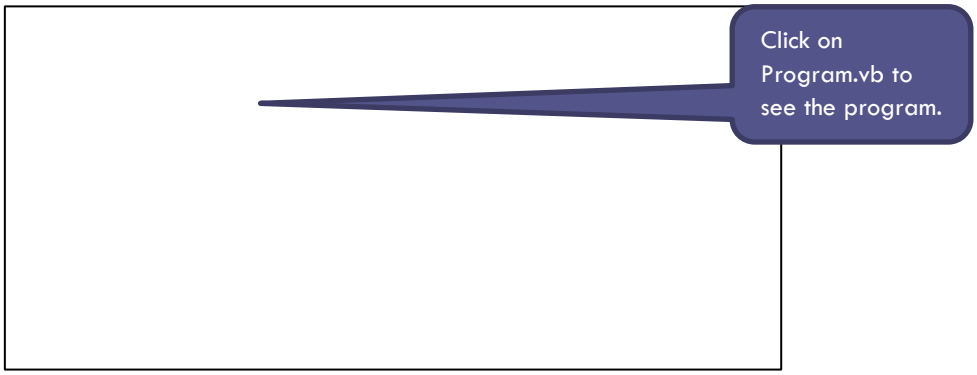


Figure 9: Locating the Program.vb tab

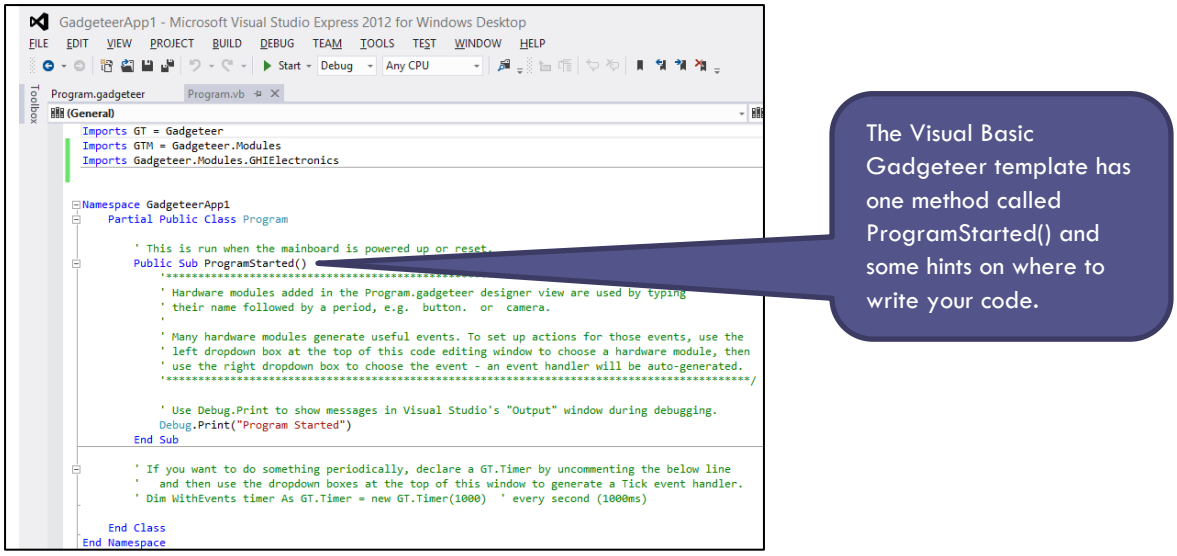


Figure 10: The ProgramStarted() method

The next step is to write the code for your project. The remaining chapters in the book explain how to write the code for several different projects. While you are doing this you will be learning how to write computer programs. The skills you learn when programming in one language are very transferable to other languages you learn in the future. Also you will be learning a lot about how electronic devices and computer applications are made.

CHAPTER 3. PLAYING TUNES

KEY TERMS
Program Sub Procedure Comment

MODULES YOU WILL NEED
Cerberus mainboard USBClientSP power module Tunes module

OVERVIEW

In this first project you will learn to use the tunes module to play a tune using .NET Gadgeteer. Before starting, the next section will give you a short introduction to what a program in Visual Basic looks like.

PROGRAMMING IN VISUAL STUDIO

Visual Studio can look very complicated when you start using it but you will quickly learn which parts are important. When you are programming you need to focus on the main window.

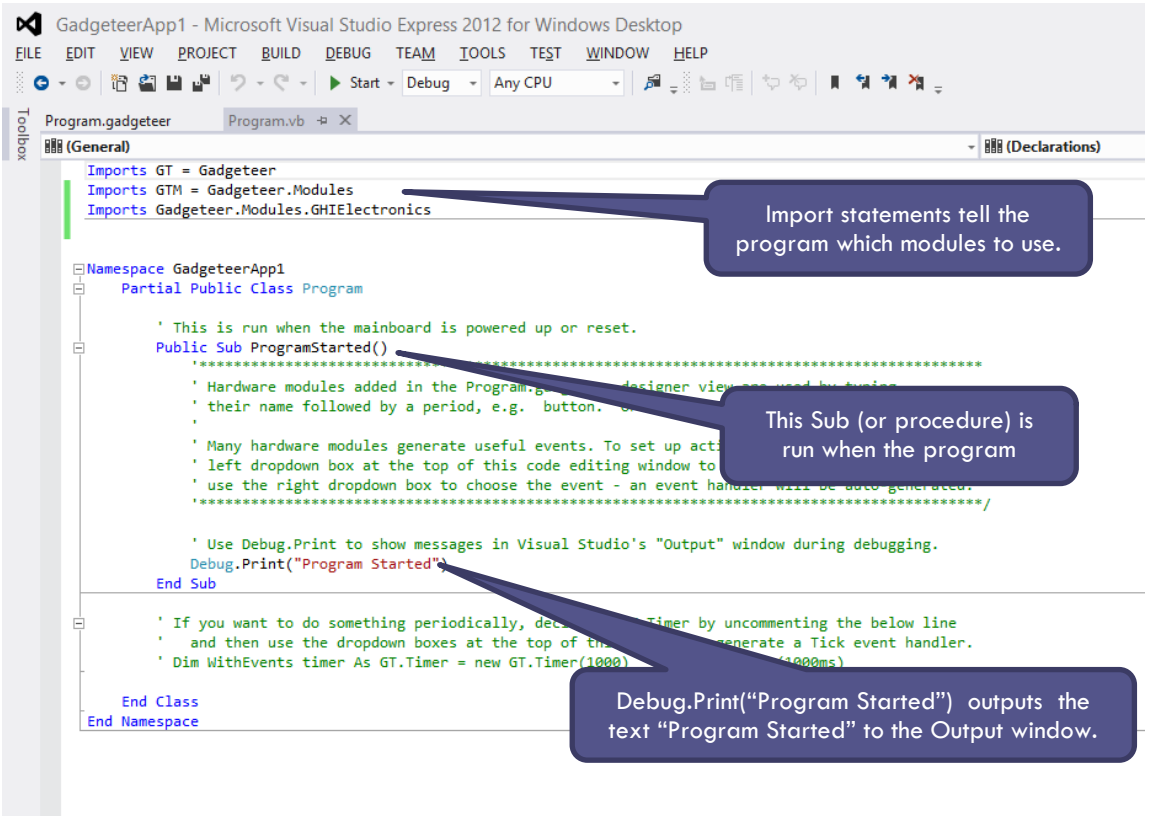


Figure 11: The main code window in Visual Studio

A program begins with `Import` statements and these specify what modules will be used in the Gadgeteer program. The next section of Program.vb declares the program itself. The program begins where the code says `Partial Public Class Program` and ends with `End Class` at the bottom of the page. This will always be written for you by the Gadgeteer Visual Basic template so you will not need to worry too much about it.

Inside a Program Class are procedures. A procedure in Visual Basic is known as a Sub.

Each Sub (procedure) begins with `Public Sub` and ends with `End Sub`. In a Gadgeteer program the beginning and end of procedures will nearly always be written for you, either in the template or added automatically when you need them. In this case the Sub is called `ProgramStarted()` and not surprisingly, runs when your Visual Basic program starts.

Later on you will create other Subs (procedures). But in this project we will put all the code we write inside the `ProgramStarted()` Sub so that it is run as soon as the program starts.

Inside `ProgramStarted()` there is some code already written but most lines start with an apostrophe and are in green. These lines are comments which are ignored by the computer when the program runs. They are there for the humans! These help you to understand what is going on and you should add comments to your own programs as you write them so that you can remember how they work too.

The only line that is not in green inside `ProgramStarted()` is `Debug.Print("Program started")`. This is an instruction to the computer to write the words "Program started" to the Output window (see Appendix C). This will be useful it provides a signal to you that the program is running correctly. See Figure 11 for details of what this looks like on-screen.

In this project we will add some code to the `ProgramStarted()` Sub and this will tell the computer to play a tune using the tunes module.

TUTORIAL: PLAYING A TUNE

Step 1: Assemble these modules

Referring back to Chapter 1, the first step when building a Gadgeteer project is to connect up the hardware modules you will need. For this project, you will need a mainboard, a tunes module and a power module as shown in Figure 12.

Cerberus Mainboard

USB Client SP power module

Tunes module

Figure 12: Modules needed for Tunes project

Connect these modules together using the cables to the correct sockets on the mainboard. The letter on the module tells you which sockets are allowed (see page 11 for more information).

Step 2: Use the Gadgeteer Designer to link the modules together

The next step is to open Visual Basic and to start a new .NET Gadgeteer application (as shown on page 11). Following the instructions there, drag a tunes module, mainboard and power module on to the Gadgeteer Designer screen and connect them up in exactly the same way as you have connected your hardware. The diagram in Figure 13 shows an example.

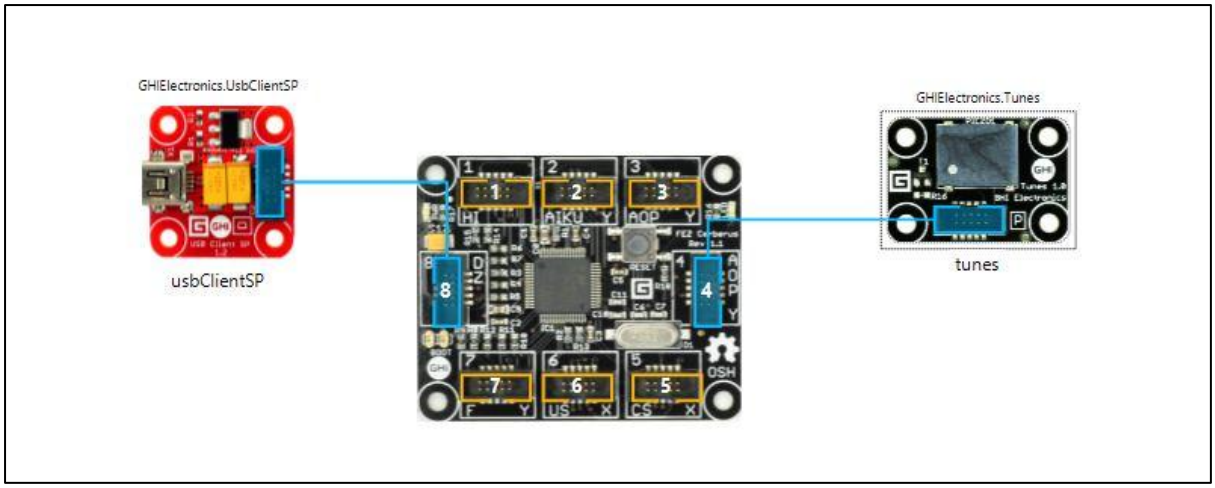


Figure 13: Modules in the Designer for the Tunes project

When you have finished connecting the modules to the mainboard and your configuration matches your hardware, select Save All under the FILE menu to save the whole project. Ensure the hardware is connected to your computer using the USB cable. You are then ready to start programming your project. If you ever go back and make changes to the Designer be sure to save the file.

Step 3: Write your program

Now click the Program.vb tab at the top of your program and your Gadgeteer Designer window should disappear and the code window should appear.

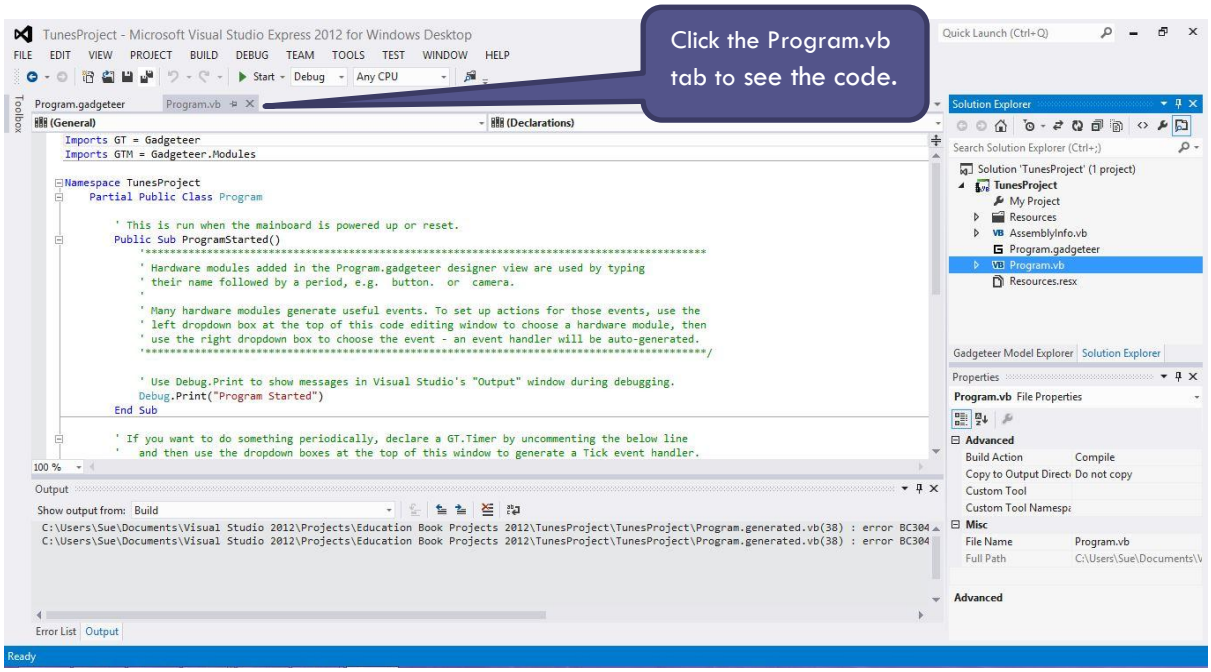


Figure 14: Code window for Tunes project

Visual Studio has various features that assist with editing your code, many are intuitive but it is worth looking at the IntelliSense appendix on page 96 for a quick introduction to these features that can save time and make coding these examples easier.

Underneath the line `Debug.Print("Program Started")` add the line highlighted in yellow below:

```
Public Sub ProgramStarted()
    ' Use Debug.Print to show messages in
    ' Visual Studio's "Output" window during debugging.
    Debug.Print("Program Started")
    tunes.Play(261)
```

`tunes.Play(number)` plays a sound at a certain pitch or frequency. The frequency is specified by the number in brackets and is measured in a unit called Hertz, abbreviated to Hz. A frequency of 261 Hertz is the frequency of middle C, if you play the piano! Now add the lines highlighted in yellow in the code below.

```
Public Sub ProgramStarted()
    ' Use Debug.Print to show messages in
    ' Visual Studio's "Output" window during debugging.
    Debug.Print("Program Started")
    tunes.Play(261)
    Thread.Sleep(1000)
    tunes.Stop()
```

`Thread.Sleep(1000)` causes the program to pause for 1000 milliseconds, which is one second. After this `tunes.Stop()` will stop the note playing.

This program should print `Program Started` in the Output window (if you cannot find this, see Appendix C) and then play a middle C for one second. So now it is time to get this first program running.

Now is a good time to save your program. To do this, click on the Save All icon . It is very important to save 'all' as there are lots of different files in a Gadgeteer application and they all need to be saved together. If you are working in Visual Studio 2010, you may need to name your project for the first time – call it "Playing Tunes". If you are using Visual Studio 2012 Express you will already have named your project as this is done when you create a new project.

Now press the green play button at the top of the screen to run the program. Allow around half a minute for the program to load onto the Gadgeteer mainboard and then it should play automatically.

Does your program work? If it does not, check the following:

- Is the USB cable plugged in and attached to the red power module (USB Client SP)?
- Are the connections to the sockets on the mainboard the same as those that you have used in the Designer?
- Did you install the Gadgeteer software that comes with the FEZ Cerberus Tinker Kit?
- Did you type in the code exactly as written above?

If you have checked all these and you still have errors, go to page 98 (the troubleshooting section of the appendices) to look up some strategies that will help you check that your code is written correctly and get it working.

When you press the green play button your program is deployed to the hardware that you have plugged in via USB. The play button starts a process called debugging and is covered in more detail on page 107. Since your program has been deployed to the hardware, every time you reset the Gadgeteer device from now on (or power it up) your program will execute. In the case of the Tunes example, it will beep every time you connect it over USB. Remember that you are programming the hardware to create a standalone gadget.

You cannot edit your code or alter the modules of the Designer if your program is running. You will first need to press the stop button or Stop Debugging on the Debug menu at the top (Shift + F5).

When your program successfully plays one note, it is possible to create a longer tune. You could do this with a sequence of `tunes.play(<number>)` commands. Alternatively you can save a series of individual notes into a melody. Firstly, here is how to create a longer tune with more `tunes.play()` commands.

The scale of C Major uses frequencies shown in Table 1.

Table 1 : Scale of C Major and corresponding frequencies

Note	Frequency (in Hz)
Middle C	261
D	293
E	329
F	349

Note	Frequency (in Hz)
G	392
A	440
B	493
C	523

To play the scale of C Major using Gadgeteer, add these frequencies to your program. Simply add some more lines with `tunes.play(<frequency>)` using the numbers in the table. Using Thread.Sleep() between notes makes the computer wait for a certain number of milliseconds before playing the next note. If you are not sure how to do this, refer to the code below to help you.

```
Partial Public Class Program

    ' This is run when the mainboard is powered up or reset.
    Public Sub ProgramStarted()
        Debug.Print("Program Started")

        tunes.Play(261) ' Middle C
        Thread.Sleep(1000)
        tunes.Play(293) ' D
        Thread.Sleep(1000)
        tunes.Play(329) ' E
        Thread.Sleep(1000)
        tunes.Play(349) ' F
        Thread.Sleep(1000)
        tunes.Play(392) ' G
        Thread.Sleep(1000)
        tunes.Play(440) ' A
        Thread.Sleep(1000)
        tunes.Play(493) ' B
        Thread.Sleep(1000)
        tunes.Play(523) ' C
        Thread.Sleep(1000)

        tunes.Stop()
    End Sub

End Class
```

Another way of playing a tune is to add some notes to a "melody" and then play the melody.

To do this you will need to declare a variable melody. A variable is a container that holds a value. Each variable has a particular data type associated with it – this refers to the type of information (or data) that the variable will be used to store. For example whole numbers are stored using a data type called an Integer. In this case we are creating a variable called melody which is an instance of a data type called Melody (capital M). Variables will be described in detail in the next chapter.

You can declare a melody using this line of code (we have also added a comment):

```
' declare a new melody
Dim melody As Tunes.Melody = New Tunes.Melody()
```

Once you have declared the melody, you can add notes to the melody – for each note you add you need to specify the frequency (pitch) and the duration (length). The frequency is in Hertz as before and the duration is in milliseconds (thousandths of seconds).

So for example, to play middle C for 2 seconds use the line of code:

```
melody.add(261,2000)
```

```
Dim melody As Tunes.Melody = New Tunes.Melody()
' declare a new melody

melody.Add(261, 600) ' Middle C
melody.Add(293, 600) ' D
melody.Add(329, 600) ' E
melody.Add(349, 600) ' F
melody.Add(392, 600) ' G
melody.Add(440, 600) ' A
melody.Add(493, 600) ' B
melody.Add(523, 600) ' C

' Now play the melody
tunes.Play(melody)
```

Edit your program by changing the lines of code starting with `tunes.play()` and instead use `melody.add(<frequency>, <time>)` as shown in the code.

Run your program and check that it works as you would expect. Then try the following exercises.

EXERCISES

1. Edit your Visual Basic program to see if you can play the scale of middle C going down as well as up.
2. Edit your program so that it adds the notes for the tune "Happy Birthday" to a new melody and then plays that melody. You will need to research some more notes and frequencies.
3. (Advanced). Edit your program so that it uses a loop to repeat two notes five times, using the code given below. This uses a `For … Next` loop which you will learn properly in Chapter 7.

```
For counter As Integer = 1 To 5
    tunes.Play(261) ' Middle C
    tunes.Play(293) ' D
Next
```

4. Once this works see if you can use `For … Next` loops to play the popular tune "Frère Jacques".

SUMMARY

In this chapter, you have learned how to:

- write and run your first Visual Basic program using .NET Gadgeteer;
- use the tunes module and the `Play()` method to create a tune;
- create a new melody and add notes and time intervals to it.

CHAPTER 4. CLICKER

KEY TERMS
Variable Integer Event Event-handler Comment

MODULES YOU WILL NEED
Cerberus mainboard USBClientSP power module 2 Button modules Display N18 module

OVERVIEW

This program introduces output to a display screen. It is a simple clicker that somebody might use at an event to count how many people have gone into a room. Imagine a very popular event taking place where it is important it does not get too crowded. We need to click when somebody goes in and display how many people have gone into the room. We also need to be able to reset the counter.

NEW CONCEPTS: WHAT IS A VARIABLE?

A key concept in programming is understanding what a variable is. A variable is like a box which holds a value. This value can change when the program is running.

Figure 15: Assigning values to variables

```
myName = "Bob"
myName = "Carla"
```

In these examples, "Bob" is a value, `myName` is a variable. The variable `myName` is assigned the value "Bob". Then it is assigned the value "Carla" which overwrites the previous value.

Each variable has a specific 'data type'. This refers to the type of information that the variable is able to store. The myName variable above had text data in it and so that's its data type. We can also have variables that have numbers in them. One type of number is called integer. This is a whole number with no fraction or decimal place.

```
myNumber = 46
myAge = 15
```

Before using a variable in Visual Basic, it needs to be declared and this is done with the `Dim` statement. `Dim` actually stands for Dimension, but you could think of it as standing for "Declare in Memory" which is what it is used for. When a variable is declared, a space in memory is set aside for it.

The `Dim` statement needs to specify the name of the new variable and its data type. An example of declaring a variable in Visual Basic and then assigning a value to it is given below:

```
Dim number As Integer
number = 356
```

You can also declare the variable and assign the value all in one go:

```
Dim number As Integer = 356
```

There are many data types that can be used with variables but you will use just a few to start with. The commonly used ones are shown in Table 2.

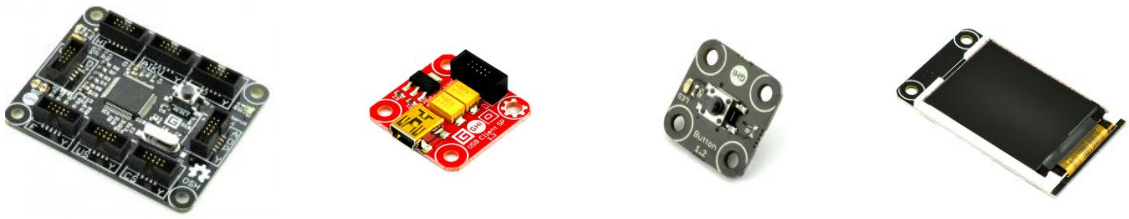
Table 2: Different data types and their uses

Data type	What it is used for?	Example
Integer	Whole number	1, 5, 35, -5
Double	Decimal (or fractional) number	1.56, 4.78
String	Piece of text	"Happy Birthday"
Boolean	True or False only	True, False

TUTORIAL: BUILDING A CLICKER

Start by putting together the hardware you need as you have done previously (see page 11). Use the socket letters to see how the modules attach to the mainboard.

Step 1: Assemble these modules



Cerberus mainboard | USB Client SP power module | Button | Display N18 module

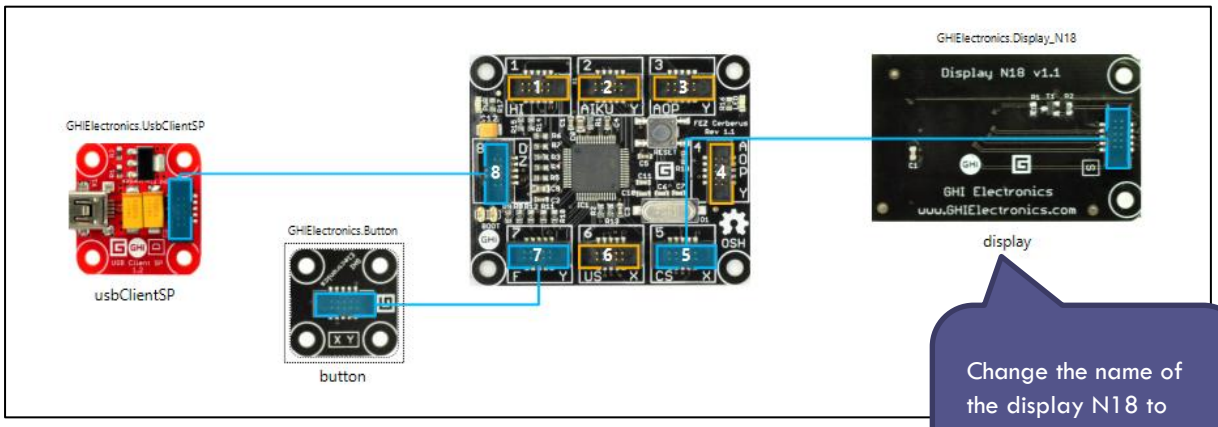
Figure 16: Modules needed for Clicker project

Step 2: Use the Gadgeteer Designer to link the modules together

Open Visual Studio, save your program as "Clicker" and then use the Designer to drag on the modules you are using and link them together. The diagram in Figure 17 shows an example, but you can wire things differently if you want. Just make sure you connect the modules together in the Designer using the same sockets as you have used when assembling your real hardware.

When you initially drag the modules on to the Designer screen the display module will be called “display_N18”. This can be seen as text which appears under the photo of the module. You should rename the display by clicking on the text “display_N18” underneath the picture and then changing it to read “display”. This will make your code easier to read and is done for all exercises in this book.

When you have finished connecting the modules to the mainboard and your configuration matches your hardware, select Save All under the FILE menu to save the whole project. Ensure the hardware is connected to your computer using the USB cable. You are then ready to start programming your project. If you ever go back and make changes to the Designer be sure to save the file.



Change the name of the display N18 to “display”. This will make your program easier to read.

Figure 17: Modules in the Designer for the Clicker project

Step 3: Write your program

A very fundamental concept when programming in Visual Basic is the event. An event is when something happens and the computer reacts to it by running a piece of code. This code is something you program yourself and it is called an event-handler.

Our first program in this chapter is going to do the following:

- Display “0” on the screen when the program starts.
- Repeatedly
  - Do nothing until the user presses the button – that will be the “event”.
  - Add one to a variable called ‘count’ in the button event-handler.
  - Display the new value of the count variable on the screen whenever it changes.

This list of steps tells you that we will need to have a variable to hold the most up-to-date value of the number of people in the room. A variable is a container with a name that holds a value. This value can change while the program is running. Our variable is going to be called “count”.

In this example we will also need a second variable. This is needed to hold the value of the font (or typeface) that we use to display the number on the screen. There are only two fonts built into Gadgeteer by default but later in the book we will add more.

So the code can be written in three stages. Firstly, when the program starts, we need to display a “0” on the screen. We start by declaring the font variable which we will use later. This code goes at the top of the program just after the line `Partial Public Class Program`. The green lines beginning with an apostrophe are comments which won’t be executed by the computer. However they are important when you look back at the code because they help you understand what you wrote and why. They are also very important when someone else tries to understand your code.

```
Namespace Clicker
    Partial Public Class Program
        ' font is the variable used to hold the font name.
        ' Gadgeteer comes with two fonts: ninaB and small
        Dim font As Font
        ' This is run when the mainboard is powered up or reset.
```

After the class definition, there is a procedure called `ProgramStarted()`. As described previously, the `ProgramStarted()` procedure contains the code that runs once the program starts. By default there are a lot of comments in this section to help you get started with your first program.

Having declared the font as Font, we need to give it a value. We're going to use the NinaB font. Enter the yellow highlighted line below to do this.

```
Namespace Clicker
    Partial Public Class Program
        ' font is the variable used to hold the font name.
        ' Gadgeteer comes with two fonts: NinaB and small
        Dim font As Font
        ' This is run when the mainboard is powered up or reset.
        Public Sub ProgramStarted()
            '*******************************************************************************
            ' Hardware modules added in the Program.gadgeteer designer view are used by typing
            ' their name followed by a period, e.g.  button.  or  camera.
            '
            ' Many hardware modules generate useful events. To set up actions for those events, use
            ' left dropdown box at the top of this code editing window to choose a hardware module,
            ' use the right dropdown box to choose the event - an event handler will be auto-
            '*******************************************************************************/

            ' Use Debug.Print to show messages in Visual Studio's "Output" window during debugging.
            font = Resources.GetFont(Resources.FontResources.NinaB)
            Debug.Print("Program Started")
            display.SimpleGraphics.DisplayText(count.ToString(), font, GT.Color.Red, 50, 50)
        End Sub
```

Secondly, you will need to set up the variable count and initialise it to 0. The variable will be declared at the top of the program directly under the line where the font variable was declared.

```
Namespace Clicker
    Partial Public Class Program
        ' font is the variable used to hold the font name.
        ' Gadgeteer comes with two fonts: ninaB and small
        Dim font As Font
        ' count is a variable used to hold the value which will be displayed on the screen.
        Dim count As Integer
```

Using the data type Integer means a whole number will be stored in the variable.

You can set the count to be 0 at the same time as declaring it. This is doing two things at once but it will save remembering to set the count to 0 later on. Alter your code like this:

```
Namespace Clicker
    Partial Public Class Program
        ' font is the variable used to hold the font name.
        ' Gadgeteer comes with two fonts: ninaB and small
        Dim font As Font
        ' count is a variable used to hold the value which will be displayed on the screen.
        Dim count As Integer = 0

        ' This is run when the mainboard is powered up or reset.
```

The next step is to write the code to add one to the count and display the updated value on the screen. To do this we have to find the event handler which corresponds to the action of "pressing the button". In Visual Basic you just need to select the appropriate action from a list of all possible events and the outline of the associated event-handling procedure will be displayed.

A procedure is a section of code that has a particular purpose. We will need to write the code that we want to be executed when the button is pressed.

To find the event handler which corresponds to pressing the button, go to the top of the screen under the toolbar in Visual Studio and select button in the grey pull-down box marked "General". This selects all the events associated with the Button module. Then select ButtonPressed in the grey area to the right which is labelled "Declarations".

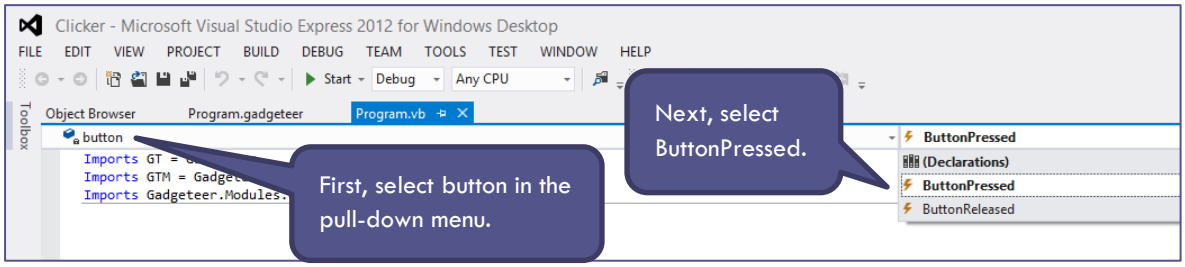


Figure 18: Finding an event handler

An empty procedure will appear at the bottom of your code listing:

```
Private Sub button_ButtonPressed(sender As Button, buttonState As Button.ButtonState)
                                                    Handles button.ButtonPressed

End Sub
```

This is the empty shell of your event handler. The next stage is to write the code to add one to the variable `count`, then to clear the screen (otherwise the next number you write will sit on top of the previous one), and then display the new value of count. This can be done by entering the following code. Once again add the highlighted lines then add a comment that makes sense to you on the line before each line of code

```
Private Sub button_ButtonPressed(sender As Button, buttonState As Button.ButtonState)
Handles button.ButtonPressed
        count = count + 1
        display.SimpleGraphics.Clear()
        display.SimpleGraphics.DisplayText(count.ToString(), font, GT.Color.Red, 50, 50)

End Sub
```

Enter the code above, save your program and test that the basic clicker works! To run your program click on the button.

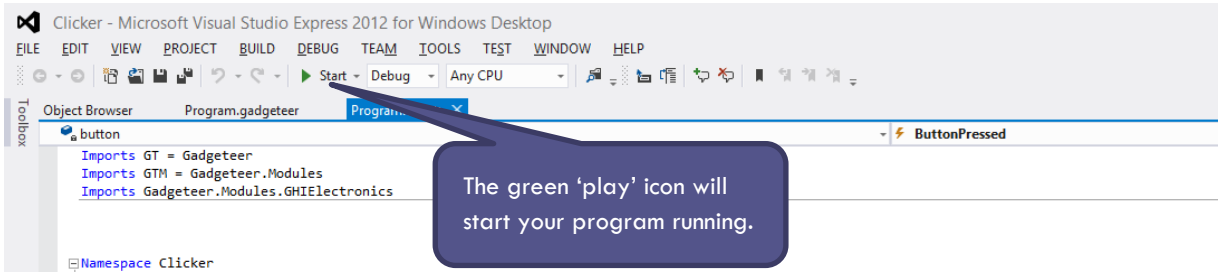


Figure 19: Use the green 'play' icon in Visual Studio to start your program running

Wait for the program to load. You should see an Output window and after a few seconds some lines of text will start to scroll by. If you cannot find the Output window, see Appendix C. After a few more seconds the words "Program Started" should be displayed. This is printed because the Visual Basic template includes a `Debug.Print("Program Started")` statement. It tells you that the program is running correctly.

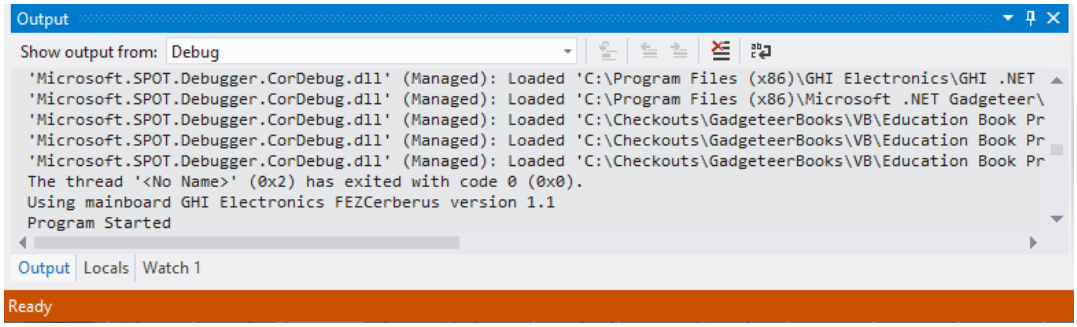


Figure 20: The Output window

Once your program runs successfully, work through the exercises below to extend the clicker.

EXERCISES

1. Add a second button which resets the clicker to 0. This button will have a different name (for example `button1`) and so you will need a separate event handler. You could change the event handler so that the second button decrements the count whenever anyone leaves the room.
2. Set a limit to your clicker so that it stops when it reaches 20 people entering the room and gives a message to say the room is full.

SUMMARY

In this chapter, you have learned to:

- write code that responds to an event;
- use variables in Visual Basic;
- change the value of a variable, for example, by adding a number;
- display the contents of a variable on the display.

CHAPTER 5. STOP WATCH

KEY TERMS
Boolean statement `If … Then … Else` statement Timer events

MODULES YOU WILL NEED
Cerberus mainboard USBClientSP power module 2 Button modules Display N18 module

OVERVIEW

This program builds on the previous chapter and enables you to build a stop watch. This will have two buttons – one to stop and start, and one to reset. To do this you will be working with timers – once you have understood how to use these you will be able to do many more projects that work in the same way.

NEW CONCEPTS: IF … THEN … ELSE… STATEMENT

The structure of a simple `If` statement in Visual Basic is shown below.

```
If <condition> Then
        <statements>
Else
        <statements>
End If
```

It is important to remember when writing an `If` statement that the `<condition>` must be something that has a "yes" or "no" answer. In other words, when you work out what the condition is, it can only be `True` or `False`. A statement which can only be `True` or `False` is called a Boolean statement.

These would be acceptable conditions to put into an `If` statement:

- `number = 5`
- `name = "Carlos"`
- `age >= 17`
- `timer.IsRunning = True (this can also be written as just timer.IsRunning)`

In this chapter you will use an If statement to test whether the timer is running. If it is running you will stop it, but if it is not running you will start it. This will enable you to create a Stop/Start button on your stop watch.

TUTORIAL: BUILDING A STOP WATCH

The modules and the Designer view are exactly the same as the Clicker, assuming that you completed the exercises and added a second button, so the code below should look familiar.

Step 1: Assemble these modules

These are the modules that you will need for this project. Use the letters on the module to connect them together using the cables.

Cerberus mainboard

USBClientSP power module

2 Button modules

Display N18 module

Figure 21: Modules needed for Stop watch project

Step 2: Use the Gadgeteer Designer to link the modules together

Open Visual Basic and use the Gadgeteer Designer to recreate the hardware you have just put together. Drag the modules on to the Designer screen as you have done previously and then link them together. The diagram in Figure 22 gives an example, but you should connect the modules in the Designer to exactly the same sockets as you used when putting the hardware together.

When you initially drag the modules on to the Designer screen the display module will be called “display_N18”. This can be seen as text which appears under the photo of the module. You should rename the display by clicking on the text “display_N18” underneath the picture and then changing it to read “display”. This will make your code easier to read and is renamed for all exercises in this book.

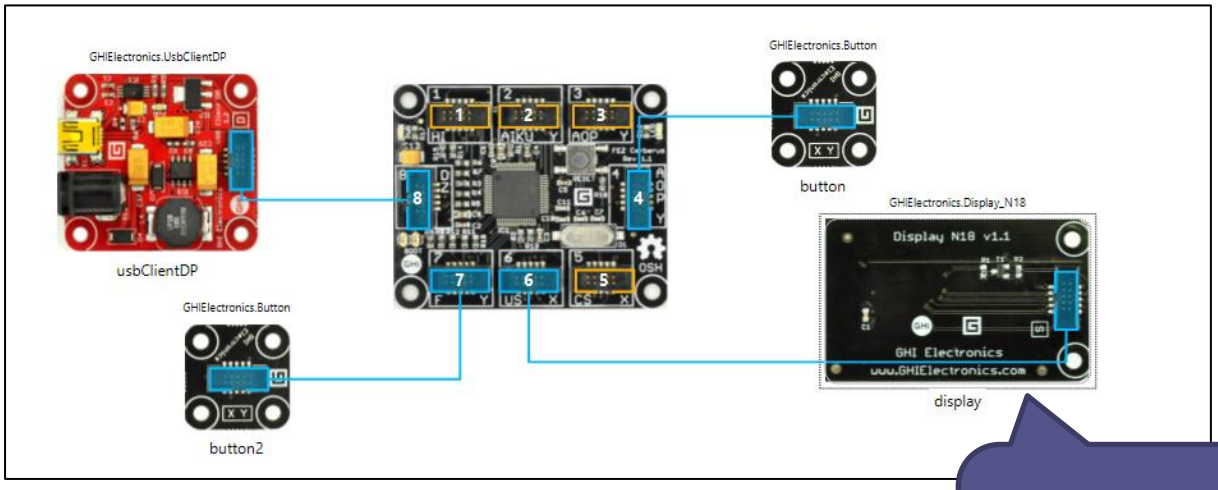


Figure 22: Modules in the Designer for the Stop watch project

Change the name of the display N18 to “display”. This will make your program easier to read.

Step 3: Write your program

The stop watch program is going to do the following:

1. Display 0 on the screen when the program starts, but also print some text with instructions.
2. Add one to the counter every second.

3. Display the new value of the counter on the screen every second.
4. Respond to the other (reset) button being pressed by setting the counter back to 0.

You should start by declaring the variables needed, see Table 3.

Table 3: The stop watch variables and their use

Variable	Data Type	Description
seconds	Integer	Used as the second counter
font	Font	Small font for instructions
font2	Font	NinaB font for the time display

Note that only two fonts, called `Small` and `NinaB`, are included with Gadgeteer by default. However, it is possible to import other fonts into .NET Gadgeteer using a utility called Tiny Font Tool.

The code you need to type in is highlighted below:

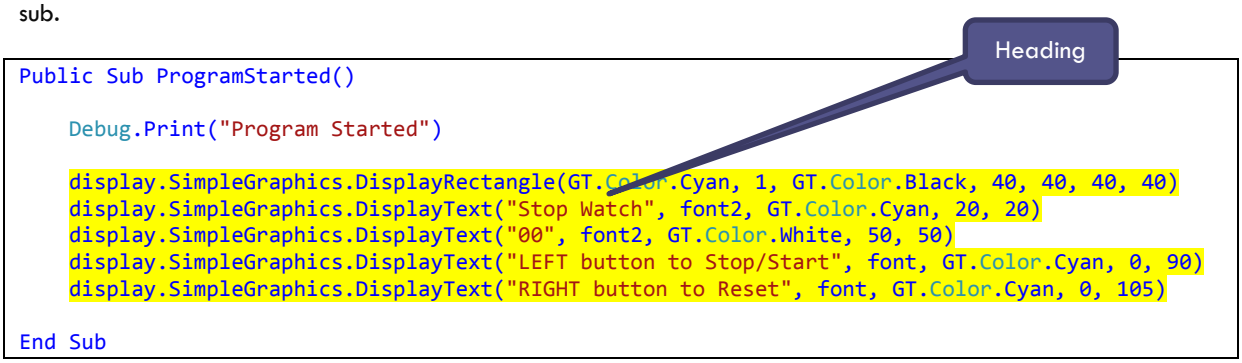
```
    Partial Public Class Program

        Dim seconds As Integer = 0
        Dim font As Font = Resources.GetFont(Resources.FontResources.small)
        Dim font2 As Font = Resources.GetFont(Resources.FontResources.NinaB)

        ' This is run when the mainboard is powered up or reset.
        Public Sub ProgramStarted()
```

Having declared and assigned the variables the next step is to set up the display on the screen.

This can be done in the `ProgramStarted()` sub (Visual Basic's name for a procedure)). Depending on the size of the screen the coordinates may be different to the example below. Here we are using a GHI Electronics Display N18 which has 128x160 pixels. Add the lines of code that are highlighted in yellow to the `ProgramStarted()` sub.



Test this to see that this works and that the screen looks as you would expect, see Figure 23.

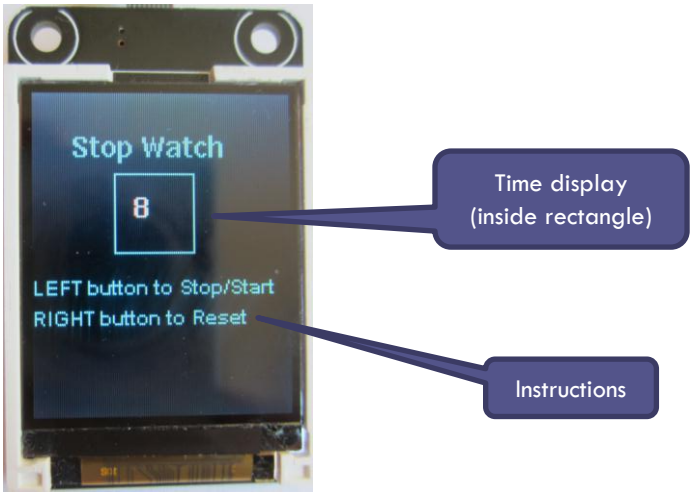


Figure 23: The stop watch in operation

The next stage is to add code to create a timer. A timer is a special type of variable which creates (or raises) an event periodically. It is useful if you want to run some specific code every so often – for example every second or every minute. The code which needs to be run periodically is put inside the timer event handler.

It is possible to have many different timers in a single program, and to start and stop them (or enable and disable them) as your program runs if you want to. Here we only need a single timer. Start by adding the following line of code which creates a new Timer variable called `timer`:

```
    Dim WithEvents timer As GT.Timer = New GT.Timer(1000)   ' every second
```

This line of code should already be in your project because it is included in the template in green as a comment. So all you need to do is to remove the apostrophe that is in front of the timer definition. Changing a comment into working code is something programmers often do and is called 'uncommenting your code'. Similarly, sometimes you want to temporarily stop certain lines of code from working and to do this you can turn them into comments by putting an apostrophe at the front. This is called 'commenting out' lines of code.

Select the `timer` from the pull down menu as shown in Figure 24 and then select the `Tick` event from the top of the program window. This will generate a `timer_Tick` procedure ready to program with your code.

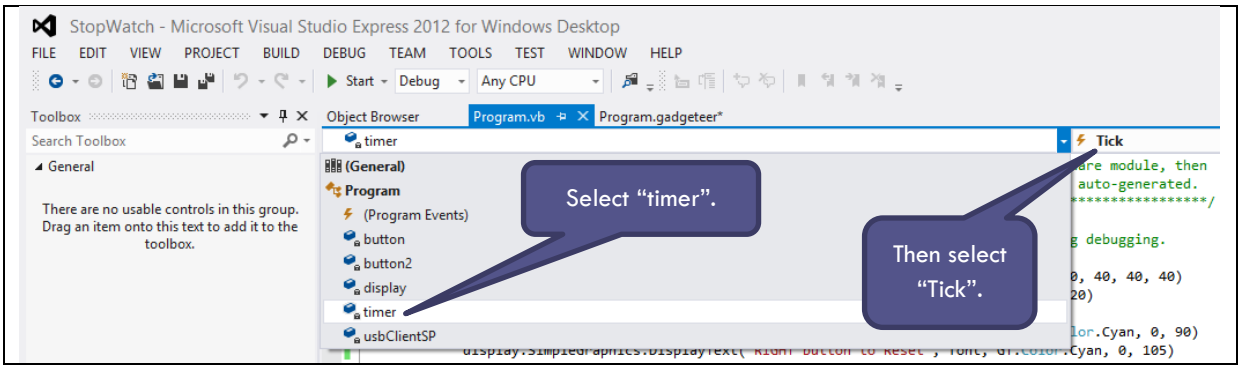


Figure 24: Selecting the `timer_Tick()` event handler

Then add the timer code as follows:

```
Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick
    seconds = seconds + 1
    display.SimpleGraphics.DisplayRectangle(GT.Color.Cyan, 1, GT.Color.Black, 40, 40, 40, 40)
    display.SimpleGraphics.DisplayText(seconds.ToString(), font2, GT.Color.White, 50, 50)
End Sub
```

Add one to the seconds counter.

Display the time again.

Then program the button to start and stop the timer. Use the pull-down menus to create the `button_ButtonPressed()` procedure from the top of the program window as before.

```
Private Sub button_ButtonPressed(sender As Button, state As Button.ButtonState)
Handles button.ButtonPressed

End Sub
```

The first step is to start the timer when the button is pressed. To do this just add the `timer.Start()` line of code inside the `button_ButtonPressed()` procedure:

```
Private Sub button_ButtonPressed(sender As Button, state As Button.ButtonState)
Handles button.ButtonPressed
    timer.Start()

End Sub
```

Save the program and test that it works so far. At the moment the button just starts the timer but we want to use it as a toggle to turn the timer on and off. To do this we need to use a property of the timer. Properties are values that are associated with certain variables. One of the properties that `timers` have is called `IsRunning`. `IsRunning` is `True` if the timer has been started and `False` otherwise.

We can test the current value of the `IsRunning` property using an `If` Statement. The Visual Basic code we need to use in the `button_ButtonPressed()` procedure is shown highlighted below:

```
    Private Sub button_ButtonPressed(sender As Button, state As Button.ButtonState)
Handles button.ButtonPressed
        If timer.IsRunning Then
            timer.Stop()
        Else
            timer.Start()
        End If

    End Sub
```

Enter this code to check if the timer is already running, then test that your button will stop and start the timer! Then try the exercises below which will make your stop watch more sophisticated, with a reset button and minutes and seconds.

EXERCISES

1. Add the second button which will reset the timer to 0.
2. Improve your stop watch so that it counts to 60 seconds and then resets to 0 and continues counting.
3. Edit your stop watch so that it displays minutes and seconds. When the seconds reset to 0 (as in 2 above) the minutes should increment by 1.

SUMMARY

In this chapter, you have learned to:

- use a timer to control events;
- use a simple `If` statement.

CHAPTER 6. TRAFFIC LIGHTS

KEY TERMS
`Select … Case` `If … Then … Else`

MODULES YOU WILL NEED
Cerberus mainboard USBClientSP power module Button module LED Strip

OVERVIEW

In this project you will use the LED Strip module. This has seven lights in a row: two red, three amber and two green. You will program them to light up in a particular sequence and create a set of traffic lights.

NEW CONCEPTS: MORE ON IF STATEMENTS AND THE SELECT…CASE STATEMENT

As a reminder, the structure of the If statement in Visual Basic is as follows:

```
If <condition> Then
       <statements>
Else
       <statements>
End If
```

A more complex version of the `If` statement in Visual Basic chains together several test conditions. The `ElseIf` keyword is used for this as shown here:

```
If <condition1> Then
       <statements>
ElseIf <condition2>
       <statements>
Else
       <statements>
End If
```

If you have lots of different conditions you can use the `Select ... Case` statement instead of lots of `ElseIf` statements. You can use `Select ... Case` when you have an integer or char (single character) variable and you want to control what your program will do depending on the value of these variables. For example, with an Integer variable called `count`, a `Select` statement could be used to control what would happen if `count` was 1 or `count` was 2 etc. The syntax of this statement in Visual Basic is shown below:

```
          Select Case count
                Case 1
                    led_Strip(5) = True
                    led_Strip(6) = True
                Case 2
                    led_Strip(2) = True
                    led_Strip(3) = True
                    led_Strip(4) = True
                Case 3
                    led_Strip(0) = True
                    led_Strip(1) = True
                    count = 1
            End Select
```

Step 1: Assemble these modules

Cerberus Mainboard

USBClientSP power module

LED Strip

Figure 25: Modules needed for Traffic lights project

Step 2: Use the Gadgeteer Designer to link the modules together

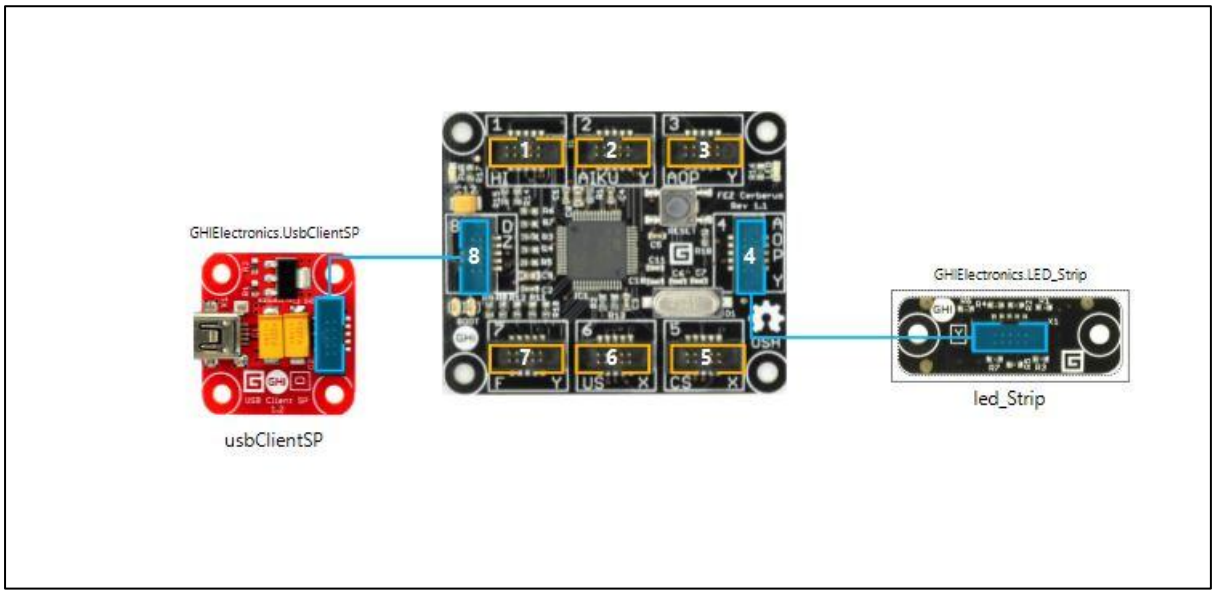


Figure 26: Modules in the Designer for the Traffic lights project

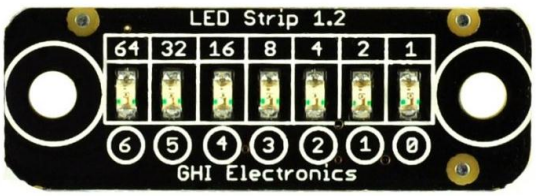
Step 3: Write your program

The basic traffic lights program is going to do the following:

- Display a red light when the program starts.
- Use a timer to display an amber light and then a green light.

You will then be able to extend this simple sequence to make it more realistic!

The LED Strip module has 7 individual lights which are numbered from zero to six. The colour of each light is as follows:



Colour	Red	Red	Amber	Amber	Amber	Green	Green
Index	6	5	4	3	2	1	0

Figure 27: The LED Strip module

An array is a special type of variable that can be used to store a sequence of values, similar to a list. Each member of the sequence is also known as an array element and is identified by its position in the list, known as the index. The first element is at position 0 as the array index always starts at 0, not 1.

The LED Strip has an array which corresponds to the seven LED lights on the module. The hardware module also has a number from 0 to 6 printed under each LED so you can easily identify the correct LED (see Figure 27). Setting one of the array elements to either `True` (on) or `False` (off) will turn the appropriated LED either on or off. For example if you wanted to turn the green (first) LED in the strip on, look to see which number is printed underneath. In this case it is 0, so you must set the value of the array at position 0 to `True` (on) as shown below:

```
led_Strip(0) = True
```

The first step in our program is to display red which means setting light 5 and light 6 to `True` so try the code shown below inside the `ProgramStarted()` procedure.

```
    Partial Public Class Program

        Public Sub ProgramStarted()
            Debug.Print("Program Started")
            led_Strip(5) = True
            led_Strip(6) = True

        End Sub
```

Add the highlighted code and write your own comments above.

Enter the code above and test that the red lights come on!

The next step is to have a sequence of lights and for this you will need to set up a timer. In the Gadgeteer template, underneath the `ProgramStarted()` procedure, there are some comments containing instructions on how to set up a timer, and an example which you can copy and use is given.

```
    ' If you want to do something periodically, declare a GT.Timer by uncommenting the below line
    '   and then use the dropdown boxes at the top of this window to generate a Tick event handler.
    ' Dim WithEvents timer As GT.Timer = new GT.Timer(1000)  ' every second (1000ms)

End Class
```

Remove the comment from this line of code.

This line of code sets up a Gadgeteer timer which will cause an event-handler to execute every 1000 milliseconds, which is once every second. Change the interval to 3000 so that the timer event is called every three seconds – this will help us run the traffic lights program at an appropriate speed.

```
Dim WithEvents timer As GT.Timer = New GT.Timer(3000)  ' every three seconds
```

The next step is to write the code that needs to be executed every three seconds, which is every time the timer tick event is raised.

Select the `timer` from the pull down menu as shown in Figure 28 and then select the `Tick` event from the top of the program window. This will generate a `timer_Tick` procedure ready to program with your code.

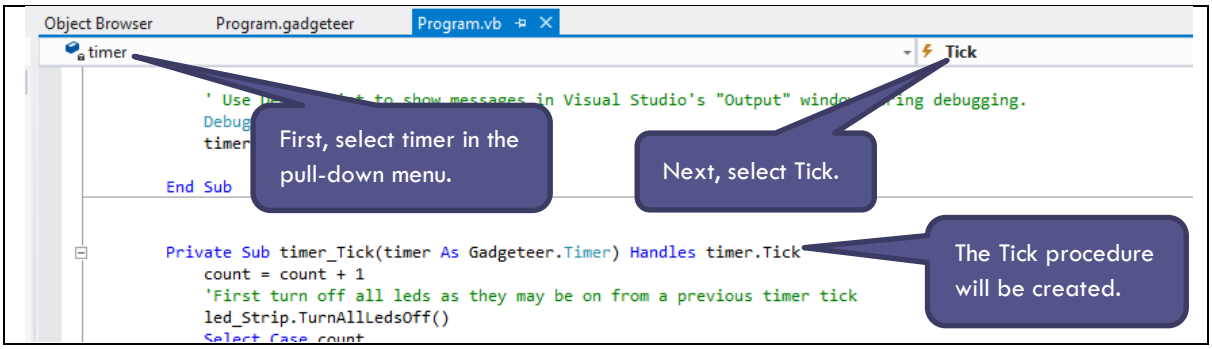


Figure 28: Creating a timer tick sub-routine

For this example we are going to run a traffic light sequence of 9 seconds in total, spending 3 seconds on red, 3 seconds on amber and 3 seconds on green. The exercises below will extend this simple example.

You will need a counter variable so declare this at the top of the program:

```
Dim count As Integer
```

Since the counter should be set to `0` at the beginning of the code add "`= 0`" to the variable declaration:

```
Dim count As Integer = 0
```

Every time the timer `Tick` procedure is called the counter should be incremented, so inside the `Tick` procedure add one to the counter as follows:

```
Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick
        count = count + 1
End Sub
```

Then the lights need to be set according to the value of the counter, as shown in the following table. The counter starts at 0 but the first instruction in the timer event handler adds one to it, so the first value of the counter that the timer will see is 1.

Table 4: Traffic light sequence

Value of count	Light wanted	What else?
1	Red	
2	Amber	
3	Green	Set counter back to 0

The following algorithm is used. An algorithm is a sequence of instructions that define the logic used within a program.

```
If counter is equal to 1 then
        change colour to red
Else if counter is equal to 2 then
        change colour to amber
Else if counter is equal to 3 then
        change colour to green and set counter back to 0
```

This algorithm includes an `If … Else` statement. This allows you to control the flow of your program depending on the value of a variable.

To program the traffic lights in Visual Basic using an `If` statement use the following code inside the `Timer_Tick` procedure:

```
Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick
    count = count + 1
    If count = 1 Then
        led_Strip(5) = True
        led_Strip(6) = True
    ElseIf count = 2 Then
        led_Strip(2) = True
        led_Strip(3) = True
        led_Strip(4) = True
    ElseIf count = 3 Then
        led_Strip(0) = True
        led_Strip(1) = True
        count = 0
    End If
End Sub
```

Nothing will happen until the timer is started. So to get this to work, add the line to start the timer in `ProgramStarted()`:

```
Public Sub ProgramStarted()
    Debug.Print("Program Started")
    led_Strip(5) = True
    led_Strip(6) = True
    timer.Start()
End Sub
```

Now test your program. What happens? The sequence is not quite as expected because none of the lights are ever switched off. To fix this problem insert the line `led_Strip.TurnAllLedsOff()` after adding one to the `count`, as shown in the yellow highlighted line below.

```
Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick
    count = count + 1
    led_Strip.TurnAllLedsOff()
```

Test your program again and it should work as expected!

Another way of writing the same program would be to use the `Select … Case` statement. This looks slightly neater than lots of `ElseIf` statements. To see the difference, edit your code as shown below, changing the `If` statement lines to the lines highlighted in yellow.

```
    Select Case count
        Case 1
            led_Strip(5) = True
            led_Strip(6) = True
        Case 2
            led_Strip(2) = True
            led_Strip(3) = True
            led_Strip(4) = True
        Case 3
            led_Strip(0) = True
            led_Strip(1) = True
            count = 0
    End Select
```

Now work through the exercises below to extend the traffic lights.

EXERCISES

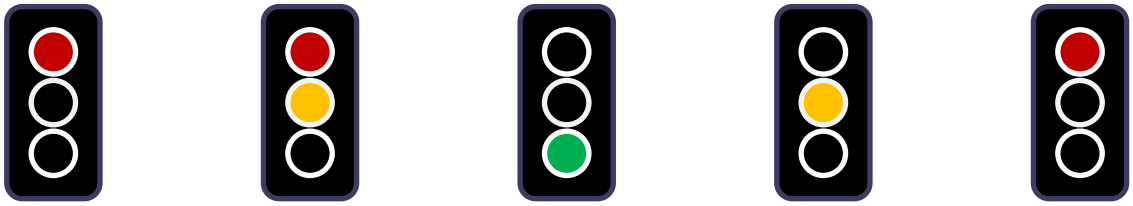


Figure 29: UK traffic light sequence

1. Change the traffic lights so that they go in the correct sequence for the UK. They should start at red, then go to red and amber together, then green, followed by just amber and finally back to red again.
2. Program the button so that it represents a pedestrian pressing a button at the pedestrian crossing. This should stop the lights at red after a small delay. When the pedestrian has crossed, the regular sequence resumes.
3. Using two kits, make a more complex system with two sets of traffic lights, and program them so that one is on red whilst the other one is on green. You may need to research some real traffic lights to see when amber is used. This is a safety critical exercise!
4. Extend the previous exercise to use the pedestrian button to override both sets of lights and set them to red.

SUMMARY

In this chapter, you have learned to:

- use an `If` Statement in Visual Basic, with `ElseIf` for different conditions;
- use a `Select … Case` statement;
- create and use a timer to control your program.

CHAPTER 7. COUNTING IN BINARY

KEY TERMS
For loop
Binary number system
Bits and bytes

MODULES YOU WILL NEED
Cerberus mainboard
USBClientSP power module
LED Strip

OVERVIEW

In this chapter you will create a binary counter which lights up a number of lights on the Led_Strip corresponding to binary numbers. In the process you will learn about binary numbers and how to use a For loop to repeat a process a number of times.

NEW CONCEPTS: BINARY NUMBERS

The number system people usually use for counting is called denary or base ten. There are ten symbols to represent numbers (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) and when those are not enough, we use more of these ten symbols in different positions to represent bigger numbers. Each time we use an extra symbol we are adding another digit to the number.

It is also useful to understand binary which is a different way of counting that computers use. Binary is sometimes known as base two. In the binary number system there are only two symbols to represent numbers (0 and 1) and when these are not enough we use more of the same symbols in different positions to make bigger numbers. Each symbol that makes up a binary number is called a bit, which is an abbreviation of binary digit.

The positions at which the symbols appear in a number are called place values. In the denary number system the place values are all powers of 10. The right-hand symbol (or digit) represents a place value of 1, next to that the place value is 10, then 100, 1000, etc. So as an example, the denary number 357 is 3 X 100s, added to 5 X 10s, added to 7 X 1s. This is shown in the Table 5 below.

Table 5: The number 357 in denary (base 10) is calculated as (3x100) plus (5x10) plus (7x1)

100	10	1
3	5	7

In binary, the place values are powers of 2, starting with 1 at the right-hand side. The next position represents a place value of 2, then 4, 8, 16 and so on. You might have noticed that rather than multiplying by 10 to calculate the place value of the next position (like we do in denary), with binary we have to multiply by 2. So the binary number 1010 is 1X 8s added to 0 X 4s added to 1 X 2s added to 0 X 1s, see Table 6. This gives an equivalent denary value of 10.

Table 6: The number 1010 in binary (base 2) is calculated as (1x8) plus (1x2)

8	4	2	1
1	0	1	0

To convert a binary number to denary, write the bits of the binary number under the appropriate place values. For example, the 8 bit number 10010100 would correspond to the binary place values as shown in Figure 30. When converted to denary this binary number equates to 148.

128	64	32	16	8	4	2	1
1	0	0	1	0	1	0	0

128 + 0 + 0 + 16 + 0 + 4 + 0 + 0 = 148

Figure 30: Binary place values

Computers often need to use 8 bit numbers because of the way that computer processors are designed. An 8 bit number can have 256 different values, ranging from 0 to 255. An 8 bit number is often called a byte.

To convert a denary number to its binary equivalent, follow these steps:

- Write down the column headings for the binary number (where you start will depend on how big your number is:
  64 32 16 8 4 2 1
- Process each column from left to right.
- If the denary number to be translated is greater than or equal to the column heading, place a 1 in the column and subtract the value of the column from the denary value.
- If the denary value is smaller than the column heading, place a 0 in the column.

In this program you will write a program which will follow these steps and convert a denary number to a binary number. Firstly, you will also be able to display a binary number as a series of lights on the LED Strip.

NEW CONCEPTS: THE FOR LOOP IN VISUAL BASIC

An important part of learning to program is learning how to repeat statements. To do this we can use a programming concept called a loop. There are two types of loop that all programming languages use – a loop that repeats a certain number of times (a `For` loop) and a loop that repeats until something happens that tells it to stop (a `While` loop).

In this program we will use the first kind of loop which is called a `For` loop. In Visual Basic the syntax of a simple `For` loop is as follows:

```
For <stepperVariable> = 1 to <number of times to loop>
       <statements>
Next
```

Using "`For`..." uses a counter to iterate through each index in the array.

Here are some Visual Basic examples using `Debug.Print()` to write to the Output window (see Appendix CE).

```
            For count As Integer = 1 To 8
                Debug.Print("Hello World!")
            Next
```

This example would write "Hello World!" 8 times. Here is another example showing what is happening to the stepper variable, in this case count.

```
            For count As Integer = 1 To 8
                Debug.Print(count)
            Next
```

This will produce the numbers 1 to 8, printed out in the Output window (see Appendix C) as shown in Figure 31.

It is important to understand that the stepper variable automatically takes on the value of the next number in the sequence from 1 to 8. Note that the counter is automatically incremented every time it goes around the loop.

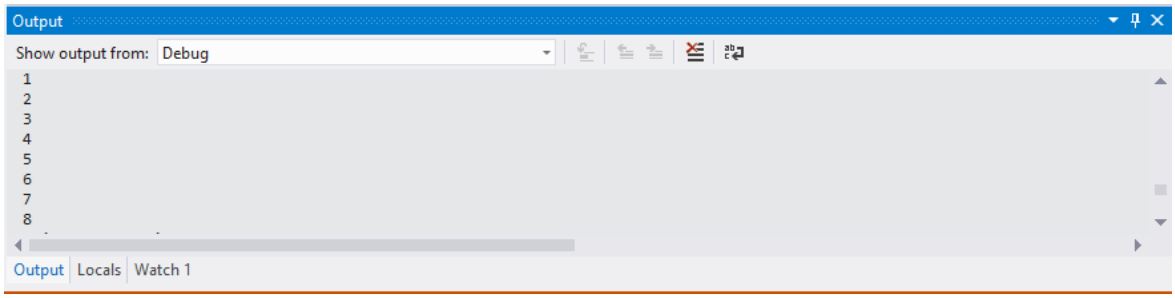


Figure 31: Output window showing debug messages

In this chapter we want to display a new binary number each time we loop.

TUTORIAL: COUNTING IN BINARY

In this example you will build a counter that outputs numbers in binary using the LED Strip module. When you have created the program, you will be able to press the button to show the next binary number as a series of lights. For example, 0 will be no lights showing and 127 will be all the lights lit up. The number 13 in binary is shown in Figure 32.

Figure 32: Binary number using the coloured lights on the LED strip (number 13 as LEDs 0, 2 and 3 are on)

For this program you will need a mainboard, power, button (input) and LED Strip (output).

Step 1: Assemble these modules

Cererbus Mainboard

USBClientSP power module

Button module

LED Strip

Figure 33: Modules needed for Counting in binary project

Step 2: Use the Gadgeteer designer to link the modules together

Assemble the modules in the Gadgeteer Designer as shown in Figure 34.

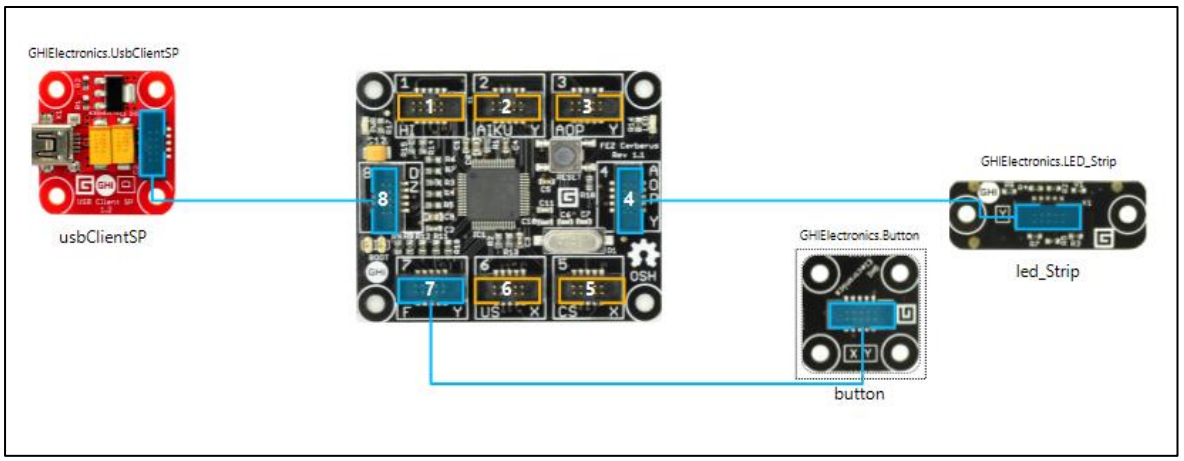


Figure 34: Modules in the Designer for the Counting in binary project

Step 3: Write your program

To work out how to write this program we need an algorithm. An algorithm is a sequence of steps for solving a particular problem. Always think carefully about your algorithm before you start to code.

The algorithm for this process can be shown in the flowchart in Figure 35.

To write the program, start by declaring the variables you will need as shown in Table 7.

Table 7: The variables needed for the binary counter

Variable	Data type	What is this for?
`whole`	Integer	The result of dividing the number by another, as a whole number.
`remainder`	Integer	The remainder when one number is divided by another and stored as an integer.

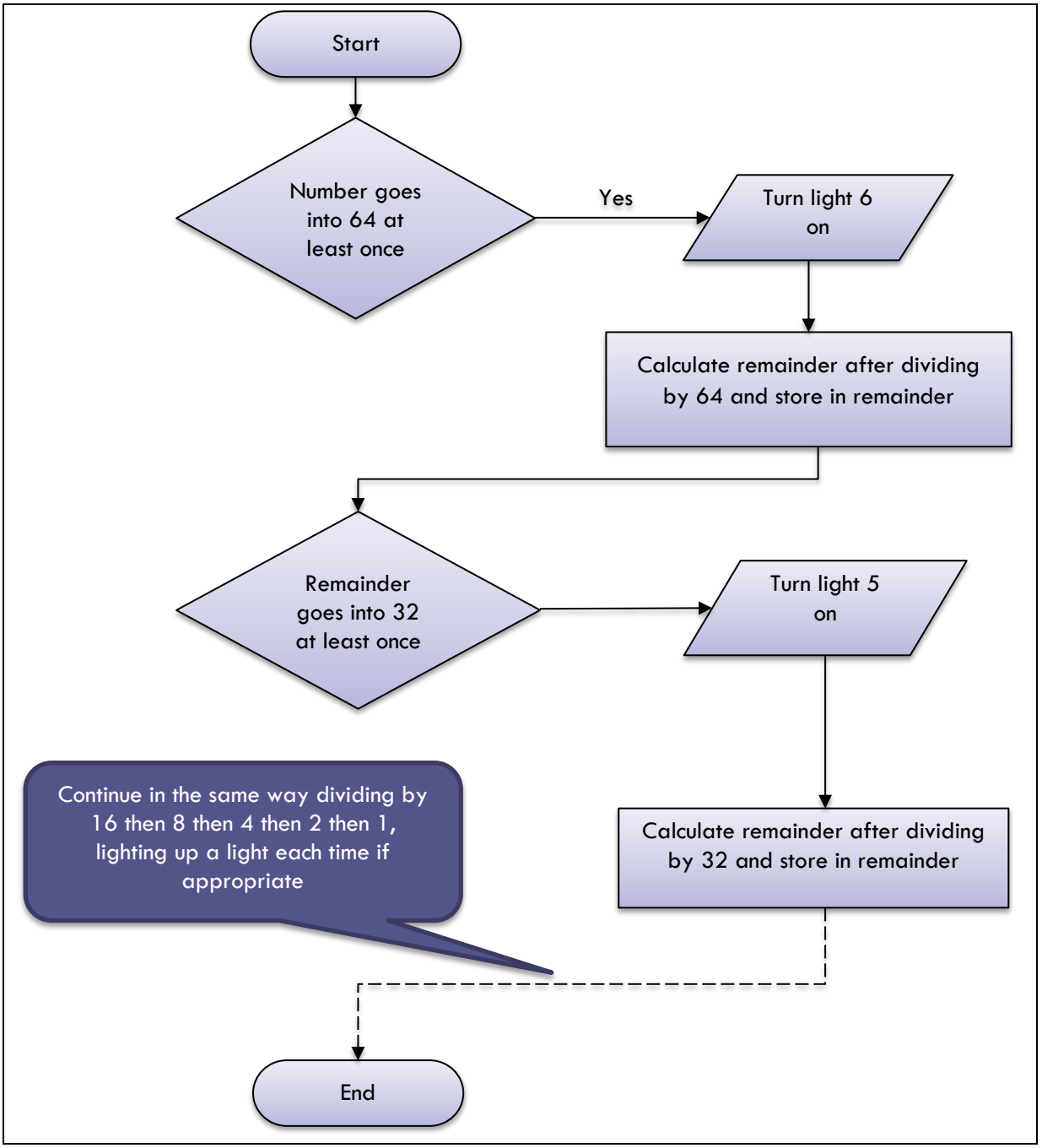


Figure 35: Flowchart for binary number counting

You will be using the LED Strip so here is a reminder of the procedures that can be used with this module:

```
TurnAllLedsOff()       ' turns all the lights off
led_Strip(X) = True    ' turns light in position X on
led_Strip(Y) = False   ' turns light in position Y off
```

To write the program the first step is to create an event handling procedure that is called when the button is pressed. To find the event handler which corresponds to pressing the button, go to the top of the screen under the toolbar in Visual Studio and select button in the grey pull-down box marked "General". This selects all the events associated with the button module. Then select ButtonPressed in the grey area to the right which is labelled "Declarations".

```
Private Sub button_ButtonPressed(sender As Button, buttonState As Button.ButtonState)
Handles button.ButtonPressed

End Sub
```

Within this procedure, declare the variables given above, as highlighted below in yellow:

```
Private Sub button_ButtonPressed(sender As Button, buttonState As Button.ButtonState)
Handles button.ButtonPressed

    Dim whole As Integer
    Dim remainder As Integer

End Sub
```

You are going to use a loop to slowly cycle through each number from 1 to 127. However before writing the code for the loop you should use the algorithm to code how just one number will be converted into binary and displayed.

We will use the following lights for each binary position as shown in Table 8. In the table, the number is 0 and all the lights would be off.

Table 8: Binary numbers shown on the LED Strip

Place value	64	32	16	8	4	2	1
Binary	0	0	0	0	0	0	0
LED Strip index	6	5	4	3	2	1	0
LED Strip index value	False	False	False	False	False	False	False

If the denary number is 21, the lights which will be on will be as shown in Table 9: The number 21 shown on the LED Strip.

Table 9: The number 21 shown on the LED Strip

Place value	64	32	16	8	4	2	1
Binary	0	0	1	0	1	0	1
LED Strip index	6	5	4	3	2	1	0
LED Strip index value	False	False	True	False	True	False	True

The Visual Basic code to convert a single number to binary and display the appropriate lights is as follows, assuming that the variable you are working with is called `number`. This follows the logic in the flowchart in Figure 35 above.

Enter the following code inside the `buttonPressed()` event handler, after declaration of the two variables.

```
            led_Strip.TurnAllLedsOff()
              ' display number
              remainder = number
              ' store value of number in for loop
```

```
            ' integer division tells you if 64 goes into the number (remainder)
            whole = remainder \ 64

            If whole = 1 Then
                ' light up the appropriate light
                led_Strip(6) = True
                ' work out the remainder after the 64 is accounted for
                ' if whole is 0 we don't need to reassign remainder
                remainder = remainder Mod 64  ' the remainder
            End If

            whole = remainder \ 32
            If whole = 1 Then
                ' light up the appropriate light
                led_Strip(5) = True
                ' work out the remainder after the 64 is accounted for
                ' if whole is 0 we don't need to reassign remainder
                remainder = remainder Mod 32  ' the remainder
            End If

            whole = remainder \ 16
            If whole = 1 Then
                ' light up the appropriate light
                led_Strip(4) = True
                ' work out the remainder after the 64 is accounted for
                ' if whole is 0 we don't need to reassign remainder
                remainder = remainder Mod 16  ' the remainder
            End If

            whole = remainder \ 8
            If whole = 1 Then
                ' light up the appropriate light
                led_Strip(3) = True
                ' work out the remainder after the 64 is accounted for
                ' if whole is 0 we don't need to reassign remainder
                remainder = remainder Mod 8 ' the remainder
            End If

            whole = remainder \ 4
            If whole = 1 Then
                ' light up the appropriate light
                led_Strip(2) = True
                ' work out the remainder after the 64 is accounted for
                ' if whole is 0 we don't need to reassign remainder
                remainder = remainder Mod 4  ' the remainder
            End If

            whole = remainder \ 2
            If whole = 1 Then
                ' light up the appropriate light
                led_Strip(1) = True
                ' work out the remainder after the 64 is accounted for
                ' if whole is 0 we don't need to reassign remainder
                remainder = remainder Mod 2  ' the remainder
            End If

            whole = remainder \ 1
            If whole = 1 Then
                ' light up the appropriate light
                led_Strip(0) = True
                ' work out the remainder after the 64 is accounted for
                ' if whole is 0 we don't need to reassign remainder
                remainder = remainder Mod 1  ' the remainder
            End If
```

The final stage is to add a loop so that this code will be executed for every number from 1 to 127. Use the `For` loop as described earlier on in this chapter.

```
        For number As Integer = 1 To 127
            led_Strip.TurnAllLedsOff()
            ' display number
            remainder = number


...REST OF CODE ENDING IN THE FINAL END IF...


        Next
```

Don't forget to add Next at the end of the For loop which sends it round the loop again.

Run your code and see what happens. Does it work correctly? The problem is that the numbers increment too quickly, so we need some code to make the program wait one second (or half a second) between each number. Try adding this line just before the `Next` part of the `For` loop.

```
        For number As Integer = 1 To 127
            led_Strip.TurnAllLedsOff()
            ' display number


...REST OF CODE ENDING IN THE FINAL END IF...


            remainder = number
            Thread.sleep(500) ' wait for half a second

        Next
```

Now test your program again. It should work as expected, lighting up the lights corresponding to all the numbers in the array.

EXERCISES

1. Edit your program so that it shows the number in denary on the display. You will need to add and attach the display in the Designer. Remember to save you project once you have added items to the Designer"
2. Edit your program so that it uses a timer rather than a `For` loop. What are the advantages?
3. Make a binary convertor. Use a timer to count on the screen from 1 to 127 slowly. Then when the user clicks a button, stop the timer and convert the number to binary (show this in lights on the LED Strip and on the screen.

SUMMARY

In this chapter, you have learned to:

- convert binary to denary and vice versa;
- use a `For` loop to repeat some code a set number of times;
- use an algorithm represented as a flowchart to develop a program in Visual Basic;
- set different light combinations on the LED Strip.

CHAPTER 8. BURGLAR ALARM

KEY TERMS
Illuminance Write to file Storage device Caching Data types: Boolean and Double

MODULES YOU WILL NEED
Cerberus mainboard USBClientSP power module Tunes module Light sensor

OVERVIEW

In this chapter, you will build a burglar alarm that uses the light sensor module in Visual Basic to detect when light levels change and give an alarm using the Tunes module.

The tutorial will demonstrate how to make a simple burglar alarm which sounds when light comes into the area where the precious items are stored, and then a more complex one which plays two sounds over and over using a while loop to control what happens.

TUTORIAL 1: CREATING A BURGLAR ALARM

Step 1: Assemble these modules

Firstly, assemble these modules, connecting them using the letters on the modules to help you.

Cerberus mainboard

USBClientSP power module

Light Sensor

Tunes module

Figure 36: Modules needed for Burglar alarm

Step 2: Use the Gadgeteer designer to link the modules together

Use the Gadgeteer Designer to put together the modules as shown in Figure 37.

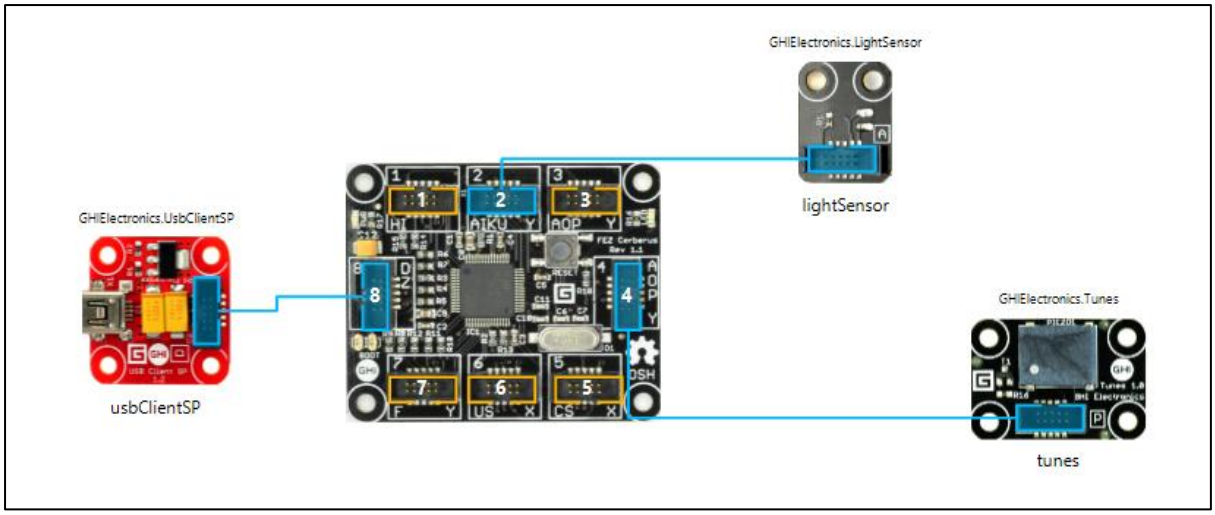


Figure 37: Modules in the Designer for the Burglar alarm project

Step 3: Write your program

The light sensor within your .NET Gadgeteer kit provides a procedure to return the amount of light recorded in a measurement known as Lux. This procedure (or Sub) is called `GetIlluminance()`. Lux values vary from household lighting at around 50 or 60 lux to a bright summer's day which can be 10,000 lux. You will be able to experiment with lux values once you have built your burglar alarm. See 0 for a list of lux values.

When the program runs, a timer will be used to detect a light reading every few seconds. If the light reading is greater than 50 lux it could be assumed that the sensor has detected indoor lighting. If you hide your Gadgeteer burglar alarm in a dark place with all your valuables, the presence of light would indicate that the dark place had been disturbed.

To start writing the code, add a timer by uncommenting the timer line.

```
Dim WithEvents timer As GT.Timer = New GT.Timer(1000)  ' every second (1000ms)
```

Then use the pull-down menus to create the Timer Tick event handler as shown in the previous chapter, Figure 28.

```
    Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick

    End Sub
```

The simplest burglar alarm will use the following algorithm:

```
Every few seconds:
        Take a light level reading
        If the reading from the light sensor is greater than 50 then
                Play a sound at frequency 800 Hertz
```

Write the code inside the `timerTick()` procedure as follows:

```
    Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick
        Dim amountLight As Double
        amountLight = lightSensor.GetIlluminance()
        If amountLight > 50 Then
            tunes.Play(800)
        End If
    End Sub
```

Don't forget to start the timer! To do this put the following code into the `ProgramStarted()` procedure.

```
Partial Public Class Program

        ' This is run when the mainboard is powered up or reset.
        Public Sub ProgramStarted()

            Debug.Print("Program Started")
            timer.Start()

        End Sub
```

Test your code works and that the alarm goes off.

The final stage in this program is to record the fact that the burglar alarm went off by writing this information to a file. This will use the SD card module. But before moving on to that section, try the exercises below.

EXERCISES

1. Add a button to the burglar alarm so that the button stops the alarm from going off, and resets it.
2. Add an LED status indicator to the burglar alarm using the LED Strip module, for example going from green to red when an intruder is detected.

NEW CONCEPTS: SAVING TO THE SD CARD

Once you start to write more complex programs, you will probably need to store data somewhere permanently. This means that your program can access the data again, or you might want to transfer the data to your computer for further processing. The Fez Cerberus Tinker Kit includes an SD card module and an SD card that allows you to do this. If you place the SD card into the slot on the SD card module and attach the module to the correct socket on your mainboard then you can access the data on the SD card from your Gadgeteer program.

From Visual Basic you can read data that is already stored on the SD card and you can also write new data onto the card. In this chapter we will do the latter. You will also need an SD card slot in your computer so that you can see what data has been stored on the actual SD card.

Some of the procedures and properties associated with the SD card module are given in the Table 10 below.

Table 10: Procedures and properties associated with the SD card module

Name	Function	What it is for?
`sdCard.IsCardInserted`	Property	Returns True or False. To test whether an sdCard is inserted or not
`sdCard.GetStorageDevice().WriteFile(<file>,<contents>)`	Procedure	Used to write to file
`sdCard.GetStorageDevice().ReadFile(<contents>)`	Procedure	Reads the file into an array of bytes
`sdCard.GetStorageDevice().ListFiles(<filepath>)`	Procedure	Lists the files in the folder specified by the path

In this tutorial you will use `WriteFile()` and `IsCardInserted`.

When working with files you need to know the name of your file if you are reading from a file, or to tell Visual Basic the name of a new file if you are writing to a file. The file will be created if it does not exist already and overwritten if it already exists.

You also need to convert the information you want to write to the file into a list of bytes so that the `writeFile()` method can write it into your file. This can be done using the following command:

```
System.Text.Encoding.UTF8.GetBytes("Hello")
```

This will convert the string "Hello" into a list of bytes using a text coding system called Unicode (that is what UTF8 means).

This all sounds very complicated but it is just a matter of being very careful typing in the specific commands needed to work with files. The idea behind it is quite straightforward once you know how to convert the text into bytes and then use `WriteFile()` to write it to your SD card.

TUTORIAL 2: KEEPING A RECORD OF AN INTRUSION

Step 1: Add the SD card Module

Firstly, add the SD card module to the existing modules you have been using.

Cerberus mainboard | USBClientSP power module | Light sensor | Tunes module | SD card module

Figure 38: Modules needed for Burglar alarm saving to file project

Step 2: Use the Gadgeteer designer to link the modules together

Use the Gadgeteer Designer to put together the modules as shown in Figure 39, or using the sockets that you have chosen when putting the hardware together.

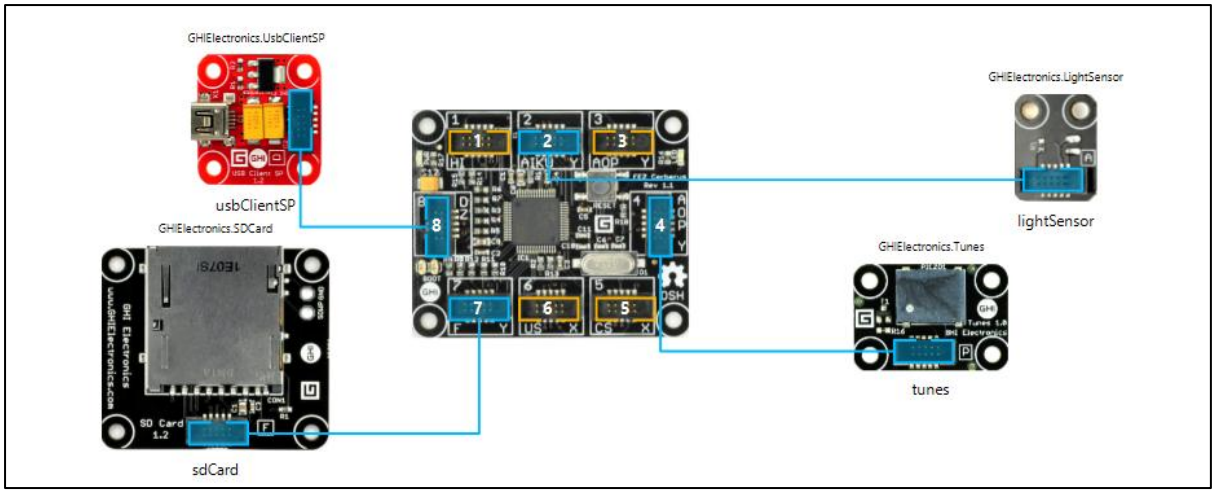


Figure 39: Modules in the Designer for the Burglar alarm saving to file project

Step 3: Write your program

To save to file you will now write a separate procedure that records when an intruder is detected for the first time. You will also need a new variable intruderAlert. This variable will be a Boolean variable which can take only two values: True or False. The intruderAlert variable will start off with the value False and then the first time the alarm sounds the program will change the value of the intruderAlert to True. When the intruderAlert variable becomes True the fact that this has happened will be saved to a file.

The variable intruderAlert will be used in several procedures so can be declared globally at the top of the program. Add it to the top of your program as shown below where the line is highlighted yellow.

```
Namespace GadgeteerApp1
    Partial Public Class Program

        Dim intruderAlert As Boolean = False
        ' This is run when the mainboard is powered up or reset.
        Public Sub ProgramStarted()

            Debug.Print("Program Started")
            timer.Start()

        End Sub
```

Now create a new procedure in your program as shown below.

```
Private Sub saveIntruderAlert()

End Sub
```

You will need to create two variables to work with files. One is just a shorthand for repeatedly writing `sdCard.GetStorageDevice()` which we will call `sdStorage`. The other is the actual report which will be written to file, in the form of a set of bytes. You cannot write a string of text directly to file.

So within the new procedure you have created add the two Dim statements as follows:

```
Private Sub saveIntruderAlert()
            Dim sdStorage As GT.StorageDevice = sdCard.GetStorageDevice()
            ' declare the intruderReport variable and encode the string "Alert!"
            ' into bytes to save it into your text file.
            Dim intruderReport As Byte() = System.Text.Encoding.UTF8.GetBytes("Alert!")
End Sub
```

The next part of the new procedure is to do the following:

- set the `intruderAlert` variable to `True` (because an intruder has been detected);
- check that the SD card is inserted, if you do not do this you could get an error;
- if it is, write the `intruderReport` to file;
- if it is not, send a message to `Debug.Print()` so that you know this has happened.

To write this code, enter the lines highlighted in yellow below.

Note that writing data to external storage devices can be slow and can use additional power, so computer systems often try to minimise how often they write data to files. They do this by keeping a note of what data needs to be written but not actually writing anything until there is enough data to make it worthwhile. This is called caching the data, and it is usually done automatically by the operating system – so even if a program includes commands to write data to storage, the data may not be written straight away in practice. With Gadgeteer we can ensure that any cached data is actually written to the SD card by issuing a command to unmount the card. This gets the card ready to be removed by a user and so the operating system makes sure that any cached data is written onto the card.

```
    Private Sub saveIntruderAlert()
            Dim sdStorage As GT.StorageDevice = sdCard.GetStorageDevice()
            ' declare the intruderReport variable and encode the string "Alert!"
            ' into bytes to save it into your text file.
            Dim intruderReport As Byte() = System.Text.Encoding.UTF8.GetBytes("Alert!")
            ' set the boolean variable intruderAlert to True
            intruderAlert = True
            ' save the incident to the SD card
            If sdCard.IsCardInserted() Then
                sdCard.MountSDCard()
                ' which is needed for the WriteFile command
                sdStorage.WriteFile("alerts.txt", intruderReport)
                sdCard.UnmountSDCard()
            Else
                Debug.Print("Intruder detected but cannot save this record to file!")
                Debug.Print("Insert the SD card into the SD Module")
            End If
    End Sub
```

Finally, you will need to call this procedure from the `timerTick()` procedure when an intruder is detected. This will be when the light sensor's reading is more than 50 and an intruder has not been detected before (if `intruderAlert` is still set to `False`).

Edit the `timerTick()` procedure adding the lines highlighted in yellow below.

```
        Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick
            Dim amountLight As Double
            amountLight = lightSensor.GetIlluminance()
            Debug.Print(amountLight.ToString())
            While amountLight > 50
                If intruderAlert = False Then
                    ' first time an intruder has been detected
                    saveIntruderAlert()
                End If
```

```
                Debug.Print("Intruder detected")
                tunes.Play(800)
                tunes.Play(300)

                amountLight = lightSensor.GetIlluminance()
            End While
            tunes.Stop()

        End Sub
```

To test that this works, run the program. When the alarm goes off, stop the program running. Take the SD card out and put it into your computer. There should be a file on in called "alerts.txt". Open this file and check that it contains the word "Alert!"

Now try the exercises below.

EXERCISES (CONTINUED)

3. Build on exercise 1 above and add a second button which will also trigger the alarm. Edit the program so that it saves the cause of each alarm – light level or button – as well as the word 'alert'.
4. Edit your program to keep a track of how much time has passed since the alarm went off. If the alarm goes off for more than a certain time interval, write 'the police have been called!' to the log file. If the alarm is cancelled then record this to the log file, along with how long it took, for example "alarm cancelled after 12 seconds".

SUMMARY

In this chapter, you have learned to:

- save text to a file on the SD card;
- understand what caching is and why it's a useful technique;
- use the SD card module as a storage device.

CHAPTER 9. MORSE CODE

KEY TERMS
Reading from file
`Select … Case`
User-defined functions
Returning a value from a function
Constant values

MODULES YOU WILL NEED
Cerberus mainboard
USBClientSP power module
Tunes module
SD card module

OVERVIEW

In this program you will firstly develop a program to play a word as the dots and dashes of Morse code, using the Tunes module. You will then be able to read in a word from a file and play this in Morse code.

NEW CONCEPTS: READING FROM A FILE IN VISUAL BASIC

In the previous program we wrote to a file using the `.txt` extension. We were able to open the file on the SD card and see the contents. In this chapter we look at how to read from a text file containing words.

A file is stored on a storage device as a sequence of bytes. In these examples, the file is stored on the SD card and the methods to write to file and read from file are accessed using `sdCard.getStorageDevice()`. The previous chapter described some of the procedures and properties that can be used with an SD card module when working with files. In this chapter you will use `ReadFile()` and the `IsCardInserted` property, see Table 11 for a reminder about these.

Table 11 : Some properties and procedures associated with the SD card module

Name	Function	What it is for?
`sdCard.IsCardInserted`	Property	Returns True or False. To test whether an sdCard is inserted or not
`sdCard.GetStorageDevice().ReadFile(<contents>)`	Procedure	Reads the file into an array of bytes

To read from a file you need to specify the filename and this must be stored on the SD card. Once you use the command to read in from a file, the contents will be stored as a series of bytes (a byte is binary data).

```
<result of reading in> = sdStorage.ReadFile(<name of file>)
```

The next step is to convert this into text, which is done using the following command:

```
<contents as text> = System.Text.Encoding.UTF8.GetChars(<result of reading in>)
```

In these two lines of code, the <contents as text> is a variable defined as a string, and the <result of reading in> is a variable defined as `Byte()` (an array of bytes).

In the following tutorial initially just one word will be read in from file and converted.

NEW CONCEPTS: CREATING A FUNCTION IN VISUAL BASIC

So far, you have been able to create your own procedure which you have called from another place in your program. As you have seen, procedures in Visual Basic are called Sub and will be written as:

```
Private Sub <procedure name>()

        <statements>

End Sub
```

Sub is short for "sub-routine". You can define your Sub as "`Public Sub`" if you want it to be accessed by another program but this is not the case with the ones we are writing here.

Sometimes you will need something to be sent back from the sub-routine and for this you would use a function instead of a sub.

```
Private Function <function name>() As <data type>

        <statements>

End Function
```

The function looks a bit more complex because it has `As <data type>` at the end of the first line. This is because the function will send back (return) a value when it has finished running and Visual Basic needs to know what kind of variable this will be, for example String, Integer or Boolean.

At the end of the function the value is actually returned and this is done by writing the line:

```
<function name> = <whatever you want to return>
```

In this case, we will define a function called `ReadFromFile()` which will return a string. The function declaration will be as follows:

```
Private Function ReadFromFile() As String

End Function
```

At the end of the function, the return statement will be as follows:

```
ReadFromFile = newWord
```

where `newWord` is a string variable that has been created and set to a particular value inside the function.

TUTORIAL: MORSE CODE

Step 1: Assemble these modules

Cerberus mainboard

usbClientSP power module

SD card module

Tunes module

Figure 40: Modules needed for Morse code project

Step 2: Use the Gadgeteer Designer to link the modules together

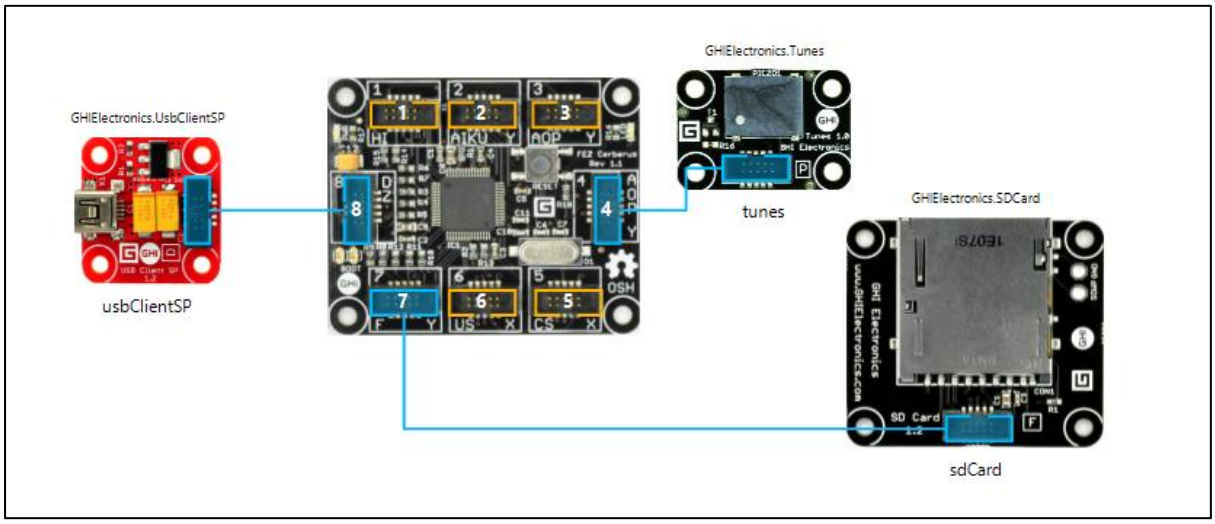


Figure 41: Modules in the Designer for the Morse code project

Step 3: Write your program

Firstly we need to be able to translate a word into Morse code. We will do this first before starting to look at file handling. The code for translating each letter to Morse code is shown in Table 12. In addition the following rules apply:

- the length of a dot is one unit;
- a dash is three units;
- the space between parts of the same letter is one unit;
- the space between letters is three units;
- the space between words is seven units.

Table 12: Morse code conversion

Character	Code		Character	Code
A	.-		B	-...
C	-.-.		D	-..
E	.		F	..-.
G	--.		H	....
I	..		J	.---
K	-.-		L	.-..
M	--		N	-.
O	---		P	.--.
Q	--.-		R	.-.
S	...		T	-
U	..-		V	...-
W	.--		X	-..-
Y	-.--		Z	--..

We want to output each letter as a sequence of dots and dashes followed by a rest. The rest will be 900 milliseconds long. Each dot will be 300 ms and each dash will be 900 ms. Also we will need a 300 ms between

each sound so that it doesn't come out as a continuous noise. As you can see in the chart above a dash should be three times as long as a dot.

We can define these as constants to make the program easier to read. A constant is a type of variable that will not change its value.

```
    Public Sub ProgramStarted()
           Const dot = 300    ' dot
           Const dash = 900   ' dash is three times as long as a dot
           Const rest = 900   ' rest between letters is three times as long as a dot
           Const gap = 300    ' gap between sounds is as long as one dot
           Const frequency = 300 ' chosen frequency in Hertz
```

Using the table above you can see that if you want to send the word "Hello" by Morse code, you will need to know the sequence of dots and dashes for the letters H, E, L and O.

For example:

- H – dot dot dot dot
- E – dot
- L – dot dash dot dot
- L – dot dash dot dot
- O – dash dash dash

Writing this out in Visual Basic code for H and E would give the following:

```
        Dim melody As Tunes.Melody = New Tunes.Melody()
            Debug.Print("Program Started")
            Debug.Print("H")
            ' play H dot dot dot dot
            melody.Add(frequency, dot)
            melody.Add(0, gap)
            melody.Add(frequency, dot)
            melody.Add(0, gap)
            melody.Add(frequency, dot)
            melody.Add(0, gap)
            melody.Add(frequency, dot)
            melody.Add(0, rest)
            ' play E dot
            Debug.Print("E")
            melody.Add(frequency, dot)
            melody.Add(0, gap)
            melody.Add(0, rest)

            tunes.Play(melody)
```

Finish off the code to play the word "Hello" in Morse Code. Then move on to the following tutorial, which shows you how to input a word from a file in Morse Code.

TUTORIAL: READING A WORD FROM A FILE AND TRANSLATING TO MORSE CODE

Obviously the program needs to become a little more efficient and we need to code in the whole alphabet. In Chapter 6 you learned about using `Select … Case` and we will use this to program each letter of the alphabet. To show how to do this we set up a variable called `newWord` which will hold the word that is being entered. Then the algorithm, written in English, will be:

Create a new variable of data type Melody

Create another new variable of type String and give it the value of the word to be converted

For each letter in the word repeat the following steps:

    If the letter is “A”

        add “dot dash” to melody

    Else if the letter is “B”

        add “dash dot dot dot” to melody

    ... (with corresponding logic for each letter)

Add a rest to melody

Play the whole tune

Check that you understand the logic behind this algorithm before continuing. The code for the first few letters of the alphabet is shown below. Type this in and add the remaining letters

```
Dim melody As Tunes.Melody = New Tunes.Melody()
Debug.Print("Program Started")
Dim newWord As String = "ABC"
For count As Integer = 0 To newWord.Length() - 1
    Debug.Print(newWord(count))

    Select Case newWord(count)
        Case "A"  ' dot dash
            melody.Add(frequency, dot)
            melody.Add(0, gap)
            melody.Add(frequency, dash)
            melody.Add(0, gap)

        Case "B"  ' dash dot dot dot
            melody.Add(frequency, dash)
            melody.Add(0, gap)
            melody.Add(frequency, dot)
            melody.Add(0, gap)
            melody.Add(frequency, dot)
            melody.Add(0, gap)
            melody.Add(frequency, dot)
            melody.Add(0, gap)

        Case "C"  ' dash dot dash dot
            melody.Add(frequency, dash)
            melody.Add(0, gap)
            melody.Add(frequency, dot)
            melody.Add(0, gap)
            melody.Add(frequency, dash)
            melody.Add(0, gap)
            melody.Add(frequency, dot)
            melody.Add(0, gap)

    End Select
    melody.Add(0, rest)

Next
tunes.Play(melody)
```

Now test the program works with your name. Insert your name into the code instead of “ABC” in your program and check that the output is what you would expect.

Testing is very important when creating software of any kind. It is important to have a range of tests that cover all the possible inputs – known as test data. Normally, we would draw up a test plan that covers all the various tests to the system and what we might expect. An example of a test plan is given in Table 13. Once you have

carried out the tests you go back to your program and make changes to the code where tests do not work as planned, and then run your tests again.

Table 13: Example of a test plan

Test data for Morse Code program	Why chosen?	Expected outcome	Actual outcome
SALLY	Typical data	Outputs morse code for Sally ... .- .-.. .-.. -.--	Works as expected
Tom	Lower case letters	Outputs morse code for TOM	Only outputs "T"
ABCDEFGHIJKLMNOPQRSTUVWXYZ	Test all letters	.- -.-. . --. .. -.- -- --- --.- ... ..- .-- etc.	Produces correct sounds as expected
99	Numbers	Won't output anything	Works as expected

Once you have got this working correctly, the next step is to create a text file in Notepad and write a word in that file, then read the file into your program and then play it in Morse code.

Firstly, create a file in any text editor such as Notepad and write one word in it. Save your text file as "word.txt" and make sure it is saved on to your SD card. Then put the SD card into the SD card Module.

The next step is to write the code to read from the file. To do this you should create a separate function which has the job of reading from the file and passing back the word to the `ProgramStarted` procedure. We will call this function `ReadFromFile()`.

```
    Private Function ReadFromFile() As String

    End Function
```

We can build up this function a little at a time. First declare the variables you need by entering the highlighted lines below. `sdStorage` is being used as a shorthand for `sdCard.GetStorageDevice()` to save typing. `dataFromFile` is the variable to hold the contents of the file which will be returned as a series of Bytes (binary data), and `newWord` is a string variable that will be used to hold the converted bytes once they have been converted into a string.

```
    Private Function ReadFromFile() As String
        Dim sdStorage As GT.StorageDevice = sdCard.GetStorageDevice()
        Dim dataFromFile As Byte()
        Dim newWord As String

    End Function
```

Next you need to check if the SD card is correctly inserted to make sure there are not any runtime errors. This is the same code we used in the previous chapter. An `If` statement is used as `sdCard.IsCardInserted` will return a `True` or `False` value.

```
    Private Function ReadFromFile() As String
        Dim sdStorage As GT.StorageDevice = sdCard.GetStorageDevice()
        Dim dataFromFile As Byte()
        Dim newWord As String

        If sdCard.IsCardInserted Then
            ' we are OK to proceed
        Else
            ' we cannot proceed without an SD card inserted
```

```
            Debug.Print("There is a problem with your SD card")
            Debug.Print("Make sure it is inserted correctly into the SD card module")
        End If

    End Function
```

The next step is to read in the file and then convert it from a list of bytes into an actual string. These are the two lines of code that were described above in the section on reading from file. The first line looks in the file "word.txt", reads from the file and puts the contents into a variable called `dataFromFile`. The second line converts `dataFromFile` into a string variable and puts it into the variable `newWord`.

```
    Private Function ReadFromFile() As String
        Dim sdStorage As GT.StorageDevice = sdCard.GetStorageDevice()
        Dim dataFromFile As Byte()
        Dim newWord As String

        If sdCard.IsCardInserted Then
            ' we are OK to proceed
            dataFromFile = sdStorage.ReadFile("word.txt")
            newWord = System.Text.Encoding.UTF8.GetChars(dataFromFile)
        Else
            ' we cannot proceed without an SD card inserted
            Debug.Print("There is a problem with your SD card")
            Debug.Print("Make sure it is inserted correctly into the SD card module")
        End If

    End Function
```

The final step is to return the word from the file out of the function back to the procedure that called it.

```
    Private Function ReadFromFile() As String
        Dim sdStorage As GT.StorageDevice = sdCard.GetStorageDevice()
        Dim dataFromFile As Byte()
        Dim newWord As String

        If sdCard.IsCardInserted Then
            ' we are OK to proceed
            dataFromFile = sdStorage.ReadFile("word.txt")
            newWord = System.Text.Encoding.UTF8.GetChars(dataFromFile)
        Else
            ' we cannot proceed without an SD card inserted
            Debug.Print("There is a problem with your SD card")
            Debug.Print("Make sure it is inserted correctly into the SD card module")
        End If
        ReadFromFile = newWord

    End Function
```

All functions must include a line to return the value that is sent back from the function.

The function is now complete and ready to be called from your program. In ProgramStarted() call this function and store the returned text into a variable. For example, earlier we used the variable newWord = "ABC" to store the string. Now change the line to newWord = readFromFile() and it will call the function you have just written and store the returned string into the variable newWord.

```
        Public Sub ProgramStarted()
            Const dot = 300  ' dot
            Const dash = 900  ' dash is three times as long as a dot
            Const rest = 900 ' rest between letters is three times as long as a dot
            Const gap = 300 ' gap between sounds is as long as one dot
            Const frequency = 300 ' chosen frequency

            Dim melody As Melody = New Melody()
            Debug.Print("Program Started")
            Dim newWord As String
            newWord = readFromFile()

…
```

Now run your program and test that the Morse Code produced by your program does really correspond to the word you entered. When this works correctly, try the exercises below.

EXERCISES

1. Extend your program by adding a display, and displaying the converted word on the display, using dots and dashes.
2. Now enhance your program so that it can convert more than one word in the `newWord` string to morse code.
3. Extend your program by coding the `newWord` string in a different way. Rather than outputting in morse code, first encrypt the word using a shifting algorithm and then output it on to the display in code. A shifting algorithm takes each letter in the alphabet and shifts it along the ASCII table a certain number of times, which we call n. In this case, shift each letter along 2 places, so that a = c, b = d, etc.
4. Extend the previous exercise so that you can shift up or down the ASCII table. Display both the encoded and decoded strings on the display.

SUMMARY

In this chapter, you have learned to:

- read a file in from the SD card and use it in your program;
- write a user-defined function that returns a value;
- use a `Select…Case` statement inside a `For` loop.

Amazing! Well done!

CHAPTER 10. DRAWING

KEY TERMS
Array Vector graphics Procedure Parameter By Val and By Ref

MODULES YOU WILL NEED
Cerberus mainboard usbClientSP power module Display N18 module Joystick

OVERVIEW

In this chapter, you will learn about drawing on the display. There are several functions for this enabling you to draw lines, rectangles, and ellipses. The tutorial will explain how to draw lines and rectangles on the screen. You will also learn about how to use an array in Visual Basic. This is an efficient way to store data in your programs.

TUTORIAL 1: HOUSE DRAW

This first short tutorial will show you how to draw pictures on the screen using `DisplayLine()` and `DisplayRectangle()`. These pictures are known as vector graphics because the coordinates, thickness, etc. of the lines are all defined in the code and then draw as sequences of lines (or vectors) as the program runs.

Step 1: Assemble these modules

Cerberus mainboard

usbClientSP power module

Display N18 module

Figure 42: Modules needed for Drawing project

Step 2: Use the Gadgeteer Designer to link the modules together

Drag the modules on to the Designer screen as you have done previously and then link them together.

When you initially drag the modules on to the Designer screen the display module will be called “display_N18”. This can be seen as text which appears under the photo of the module. You should rename the display by clicking on the text “display_N18” underneath the picture and then changing it to read “display”. This will make your code easier to read and is renamed for all exercises in this book which use the display.

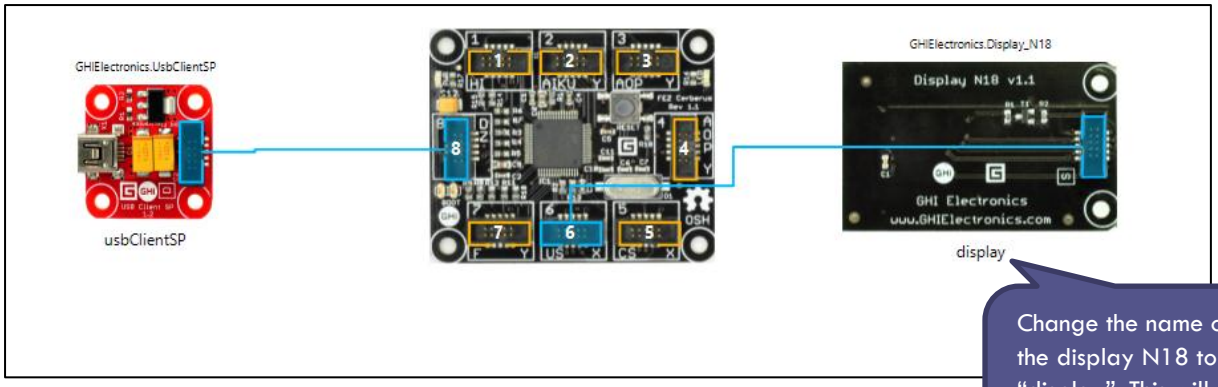


Figure 43: Modules in the Designer for the Drawing project

Step 3: Write your program

In this first program you will write the commands to draw a house. A house to us means a blue unfilled square on the screen with a triangular roof. This essentially needs `SimpleGraphics.displayRectangle()` and `SimpleGraphics.DisplayLine()` to draw these lines and shapes.

Enter the following code underneath `ProgramStarted()`:

```
Public Sub ProgramStarted()

    Debug.Print("Program Started")
    'white background
    display.SimpleGraphics.BackgroundColor = GT.Color.White
    'draw house, blue outline
    display.SimpleGraphics.DisplayRectangle(GT.Color.Blue, 4, GT.Color.White, 30, 30, 80, 80)
    ' draw roof - diagonal lines, blue colour
    display.SimpleGraphics.DisplayLine(GT.Color.Blue, 2, 30, 30, 70, 10) ' diagonal
    display.SimpleGraphics.DisplayLine(GT.Color.Blue, 2, 70, 10, 110, 30) ' diagonal
```

When you run it you should see a blue outline which is intended to be a picture of a house! See if you can add the code to draw a window and a door (and even a stick person) to your house. It should look something like Figure 44.

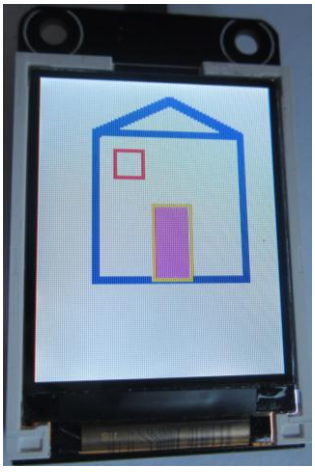


Figure 44: Does your house look like this?

NEW CONCEPTS: ARRAYS IN VISUAL BASIC

So far you have been using variables to store single items of data that you needed to know later on. However, sometimes you have lots of related variables that you need to store together and to store these we use an array. For example, you may want to record several scores, or remember a list of names or places. An array is a complex data structure in which several values can be stored and these can be referred to individually using an index. An array is declared in Visual Basic using the Dim statement. Arrays can be fixed-length or dynamic. We will be looking at fixed-length arrays only in this chapter.

The benefits of using an array can be seen in Figure 45 . When storing lots of similar data you may need to create lots of variables and then write lots of code for each variable. Instead you could store all the values that are related in one data structure called an array.

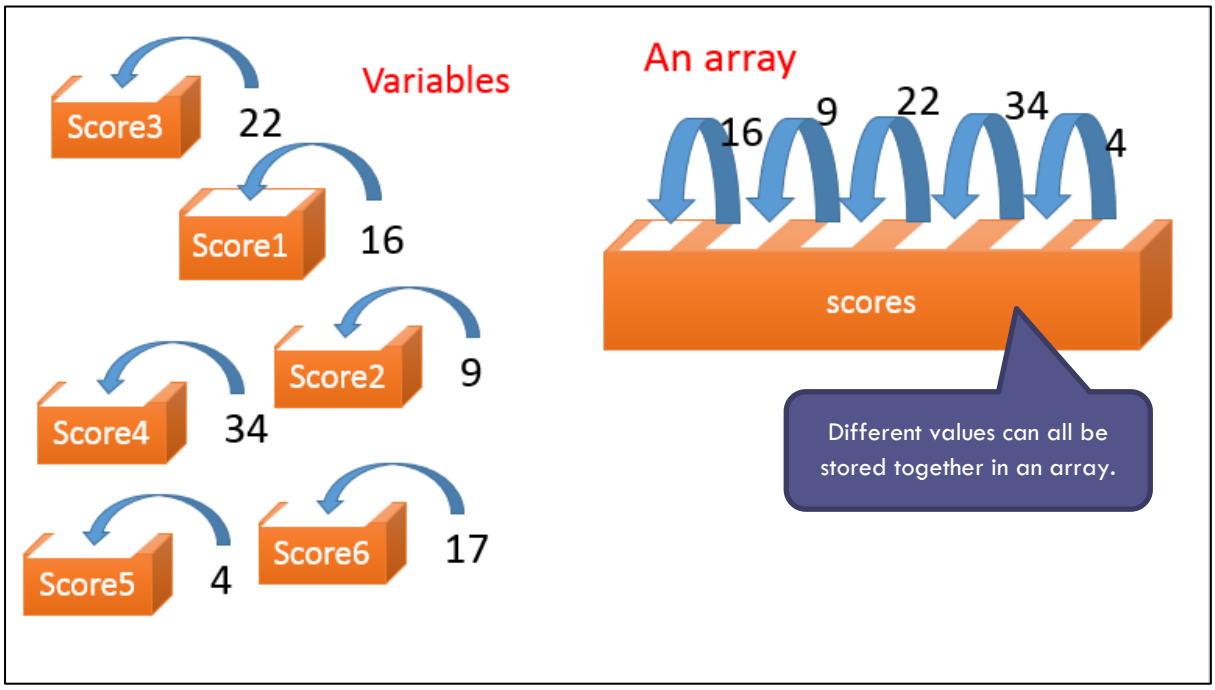


Figure 45: The difference between variables and arrays

To declare an array, use the following syntax:

```
Dim scoreValues(8) As Integer
```

This is the same as:

```
Dim scoreValues(0 To 8) As Integer
```

We could also declare an array of strings such as:

```
Dim nameArray(8) As String
```

This would be used to hold items of text.

This will define an array with 9 items of data, all of them variables, as shown in Table 14. At this point none of the data items in the array have any values.

Table 14: An empty array in Visual Basic

Referred to	Content
scoreValues(0)	
scoreValues(1)	
scoreValues(2)	
scoreValues(3)	
scoreValues(4)	
scoreValues(5)	
scoreValues(6)	
scoreValues(7)	
scoreValues(8)	

You could then put some data into the array, for example:

```
scoreValues(0) = 35
scoreValues(1) = 56
```

The values in the array would then look like Table 15.

Table 15: Assigned values in an array

Referred to	Content
scoreValues(0)	35
scoreValues(1)	56
scoreValues(2)	
scoreValues(3)	
scoreValues(4)	
scoreValues(5)	
scoreValues(6)	
scoreValues(7)	
scoreValues(8)	

Notice that there are in fact 9 variables declared because Visual Basic starts counting at 0 but finishes at 8.

A good way of looking at what is inside an array is using a `For … loop`. There are two versions of the `For` loop suitable for working with arrays. The first version is the one we have already used, where a counter variable is incremented each time the code in the loop is run. This counter case be used as an index into the array which will access each array element in turn:

```
For <stepperVariable> = 1 To 8
        <statements>
Next
```

The second version of a `For` loop is actually called a `For Each` loop. This can be used to iterate through each actual value in the array:

```
For Each <element> In <array>
        <statements>
Next
```

New concepts: Passing parameters into procedures and functions – By Val and By Ref

In the next tutorial we will create a procedure with an input parameter that we write from scratch. Up until now, the only procedures we have used with input parameters have been event handlers where the code was in part automatically generated when we selected the event in Visual Studio.

When we call a procedure with input parameters we pass some data into it for it to work with. We can do this in the two different ways shown in Table 16, `ByVal` and `ByRef`.

Table 16: The two ways of passing input parameters into a procedure, by value and by reference

`ByVal`	The value is passed into the procedure but any changes to the variable will not have an effect once the procedure has finished running.
`ByRef`	The value is passed into the procedure as before, but if the procedure makes any changes to that value in the variable then these changes will remain even after the procedure has finished running. If you need to change the value of the variable and use it again later in your code, use By Ref.

Tutorial 2: Graph draw

The second tutorial uses exactly the same modules as the first, but uses an array of values to draw a graph on the screen.

Step 1: Assemble these modules

Cerberus mainboard usbClientSP power module Display N18 module

Figure 46: Modules needed for the Graph draw project

Step 2: Use the Gadgeteer Designer to link the modules together

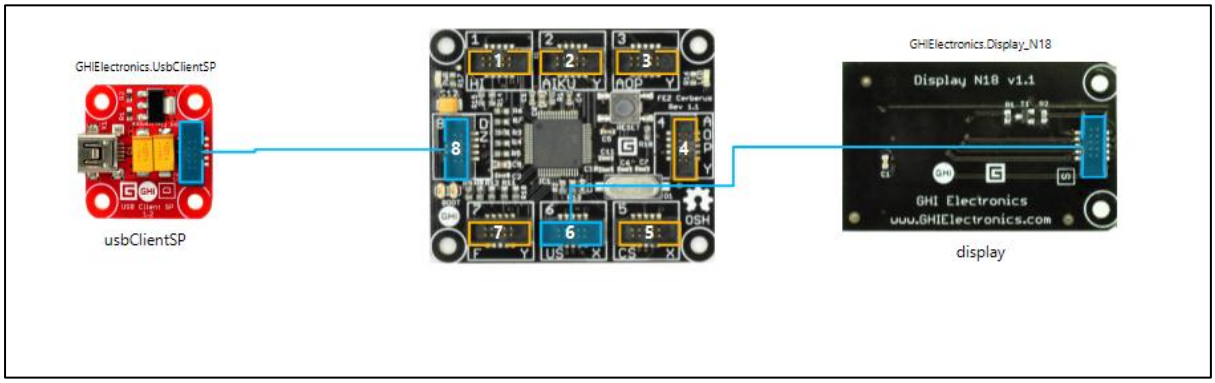


Figure 47: Modules in the Designer for the Graph draw project

Step 3: Write your program

The first step is to declare a new array in Visual Basic. This will be an array of scores gained by somebody playing a game. For the sake of this example you will decide what the scores are and write them into your program.

Table 17: The array values you should use in this tutorial

Referred to	Value
scoreValues(0)	4
scoreValues(1)	8
scoreValues(2)	1
scoreValues(3)	3
scoreValues(4)	6
scoreValues(5)	11
scoreValues(6)	5
scoreValues(7)	9

This is declared in Visual Basic using the following code:

```
Namespace GraphDraw
    Partial Public Class Program
        Const maxScore = 15
        ' This is run when the mainboard is powered up or reset.
        Public Sub ProgramStarted()
            ' Array of 8 values generated
            Dim scoreValues() As Integer = {4, 8, 1, 3, 6, 11, 5, 9}

            ' Use Debug.Print to show messages in Visual Studio's
            ' "Output" window during debugging.
            Debug.Print("Program Started")
```

In this example we are going to assume that the maximum score is 15 and we can declare that as a constant.

You then need to create a procedure to draw the array which is going to be called `displayValues` and take the array as a parameter.

We will add the following line to `ProgramStarted()` to call the new procedure:

```
Namespace GraphDraw
    Partial Public Class Program
        Const maxScore = 15
        ' This is run when the mainboard is powered up or reset.
        Public Sub ProgramStarted()
            Dim scoreValues() As Integer = {4, 8, 1, 3, 6, 11, 5, 9}

            ' Use Debug.Print to show messages in Visual Studio's
            ' "Output" window during debugging.
            Debug.Print("Program Started")
            displayValues(scoreValues)
        End Sub
```

We will then add the actual procedure using the code below which should be after the end of the `ProgramStarted()` procedure. Note that the `End Sub` instruction is automatically created for you.

```
    Private Sub displayValues(ByVal numberArray() As Integer)



    End Sub
```

Use ByVal as the array values will not be changed by this procedure.

The advantage of doing this is that we can use this procedure again and again every time we want to draw a graph like this. We want to make our `displayValues()` procedure as general as possible so that it will work for any numbers, and any width and height of display.

In this code you are stating that you will have an array of integers to be known as `numberArray()` passed into the program. It does not need to be called this when you call it from `ProgramStarted()`. When it is passed in, it takes on the name `numberArray()`.

As described above, you can use `ByVal` or `ByRef` when you declare a parameter in your own procedure. Here we do not want to change the array of numbers, just use them to draw a graph. So we can use `By Val`.

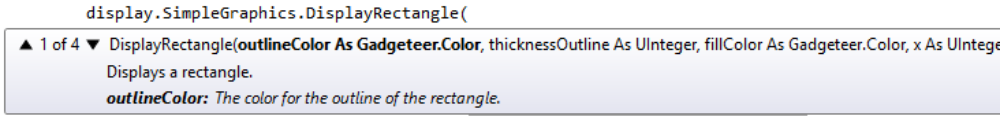


Figure 48: DisplayRectangle parameters

The next step is to declare the values we will need to draw the graph. Looking at what the `SimpleGraphics` method called `displayRectangle()` (see Figure 48) takes as its parameters gives us a clue as to what variables we need, and these are given values in Table 18.

Table 18: Setting parameter values of DisplayRectangle

DisplayRectangle parameter	Comment	Variable name
`outlineColor`	Set this to GT.Color.White	
`thicknessOutline`	Set this to 1	
`fillColor`	Set this to GT.Color.White	
`X`	Declare a variable for this as it will be different for each rectangle (i.e. each bar of the chart)	`startX`
`Y`	As above – but this will be 0	`startY`
`width`	As above	`columnWidth`
`height`	As above	`columnHeight`

We can declare the new variables we need using the code below:

```
Private Sub displayValues(ByVal numberArray() As Integer)
    ' number of numbers is the length of the array
    Dim arrayLength As Integer = numberArray.Length()
    ' width of each bar is the column width - divide display width by array length
    Dim columnWidth As Integer = CInt(display.Width / arrayLength)
    ' height of each column will depend on the number in the array
    Dim columnHeight As Integer
    ' the starting x coordinate for the rectangle (top left)
    Dim startX As Integer
    ' the starting y coordinate for the rectangle (top left)
    Dim startY As Integer
End Sub
```

To draw one bar on the display we will need to follow these steps:

1) calculate the height of the bar;
2) calculate the starting x coordinate.

The code to do this is as follows:

```
    columnHeight = CInt(display.Height / maxScore * numberArray(count))

    startX = columnWidth * count
```

CInt converts a fractional number into an Integer.

Notice that to work out the height of the column we have divided the height of the display by the maximum score to see what high a score of 1 would be then multiplied it by the actual score value. Doing calculations like this is one of the reasons you need to be comfortable with numbers when you learn programming!

Because we are dividing we could easily end up with a fractional number as the result (e.g. 4.67) and we need an integer to be stored in the `columnHeight` variable. We have used the function `CInt()` which is a very useful function and converts the answer from our calculation into an Integer before storing it in the variable `columnHeight`.

The next step is to create a loop to draw as many columns as we need. You should use a `For` loop here rather than a `While` loop because we can find out how many columns we need by looking at the length of the array. The code we end up with is highlighted in yellow below. When you enter the `For` line the corresponding `Next` instruction is automatically added for you.

```
Private Sub displayValues(ByVal numberArray() As Integer)
    ' number of numbers is the length of the array
    Dim arrayLength As Integer = numberArray.Length()
    ' width of each bar is the column width - divide display width by array length
    Dim columnWidth As Integer = CInt(display.Width / arrayLength)
    ' height of each column will depend on the number in the array
    Dim columnHeight As Integer
    ' the starting x coordinate for the rectangle (top left)
    Dim startX As Integer
    ' the starting x coordinate for the rectangle (top left)
    Dim startY As Integer

    For count As Integer = 0 To arrayLength - 1
        columnHeight = CInt(display.Height / maxScore * numberArray(count))
        startX = columnWidth * count
        startY = 0

        display.SimpleGraphics.DisplayRectangle(GT.Color.White, 1, GT.Color.White, startX,
startY, columnWidth, columnHeight)

    Next
End Sub
```

Type in the code above. When you run your program you should have a graph of 8 columns on the display. Make sure that you understand how it works before going on to try out the exercises below.

EXERCISES

1. Our current graph is actually upside down! It starts from y coordinate 0 which is at the top of the display. Edit the program so that the bars on your graph go up from the bottom of the display.
2. Edit the previous program to add the number values inside the bars on the graph.
3. Create a graph of light values. Attach a light sensor and adapt your program to take a light reading every 5 seconds (or whatever time interval you prefer). Store these readings in an array and display them in a graph.
4. Random shape drawer – create modern art! Use the random functionality in Visual Basic to create an array of 10 integers filled with random values. These will represent the radius of 10 circles. Create another array of circle x coordinate and another one of circle y coordinates. Run the program to see a variety of circles

drawn on the display. Your image should be different each time. To find out how to create a random number look ahead to the next chapter on page 75.

SUMMARY

In this chapter, you have learned to:

- declare an array with pre-defined values;
- use a `For` loop to iterate (cycle through) an array;
- pass a parameter into a function;
- use `SimpleGraphics` functions to draw lines and shapes on the display.

CHAPTER 11. REACTION GAME

KEY TERMS
Random values Using two timers Using a function to return a Boolean value Logical operators: AND, OR and NOT

MODULES YOU WILL NEED
Cerberus mainboard usbClientSP power module Display N18 module Joystick

OVERVIEW

In this chapter you will create a game using the joystick and the display. In the game you need to move the joystick in the direction of a shape that appears on the screen. If you do this quickly you get a point, if you are too slow you are out! In developing this game you will work with two timers: one to repeatedly read the joystick position at very short intervals; the other one to time out if the user is too slow or to provide the next shape. You will also learn to work with random values, as we do not want the player to know where the next shape will appear. The next section explains how to do all this in Visual Basic.

NEW CONCEPTS: WORKING WITH RANDOM VALUES

Sometimes programs need a value that is chosen randomly. All programing languages have a function which enables the creation of an integer which is different each time the program is run – this is done using a random number generator provided by the programming language. In your program you can specify the bounds of your random number – this means the range of possible values which you want it to have.

Producing a random number in Visual Basic requires four different lines of code:

1. At the top of the program, you will need to add an Import statement to tell the program to import the library with the random functions in.

```
Imports System.Random
```

2. Secondly, again towards the top of your program (before `ProgramStarted()`), declare a variable `random` as below which will allow you to create random numbers.

```
Dim random As New Random
```

3. Thirdly, you will need to also declare a variable to hold the random value you create so that you can use it in your program. I have called it `randValue` in this example, but you could use any sensible name.

```
Dim randValue As Integer
```

4. Finally, in order to create a random number you use the function `next()` as in the example below (this will create a random number between 0 and 3).

```
    randValue = random.Next(4)
```

Sometimes you want to generate random number between 1 and something (e.g. 1 to 6 to generate a random dice throw). In this case you need to generate a number between 0 and 5 and then add 1 to it:

```
    randValue = random.Next(6) + 1
```

Many games involve some sort of random element so this is a good function to be able to incorporate into your programs.

New concepts: Logical operators

So far we have used a number of arithmetic operators, for example, + (plus) and – (minus) and used operators to compare numbers such as > (greater than) and = (equals). A full list of operators is given in **Error! Reference source not found.**.

We use conditions a lot in programming to determine whether or not something should happen, and this is programmed using an if statement. When using a condition it must always be evaluated as TRUE or FALSE. For example, "It is raining" is always going to be either TRUE or FALSE. We might instead want to ask if a counter is greater than 10 – again this will either be TRUE or FALSE. Conditions are also used in some kinds of loops.

On many occasions, conditions are more complex. For example, we might want to ask if it is NOT raining. We might want to combine conditions to ask, for example, whether "It is raining" and "today is Sunday" are both true. We then need to use the logical operators AND, OR and NOT. These are described in Table 19. In the program in this chapter you will use the NOT operator when testing to see if somebody has scored (or not scored!) in the game.

Table 19: Logical operators in Visual Basic

Operator	Visual Basic example using if	Meaning
NOT	If Not timer.IsRunning then …	The condition is TRUE If the timer is not running
AND	If counter > 5 And username = "Bob"	The condition is true if the counter is greater than 5 and also the user is called "Bob"
OR	If counter > 5 Or username = "Bob"	The condition is TRUE if either the counter is greater than 5 or the user is called "Bob" or if both are the case.

More information on logical operators can be found in **Error! Reference source not found.**.

Tutorial: Building the reaction timer game

Step 1: Assemble these modules

Firstly, assemble these modules, connecting them using the letters on the modules to help you.

Cerberus mainboard

usbClientSP power module

Display N18 module

Joystick

Figure 49: Modules needed for Reaction game project

Step 2: Use the Gadgeteer Designer to link the modules together

Now use the Gadgeteer Designer to link the modules together – ensuring that your version matches the sockets you have used when linking the hardware together.

When you initially drag the modules on to the Designer screen the display module will be called "display_N18". This can be seen as text which appears under the photo of the module. You should rename the display by clicking

on the text “display_N18” underneath the picture and then changing it to read “display”. This will make your code easier to read and is renamed for all exercises in this book.

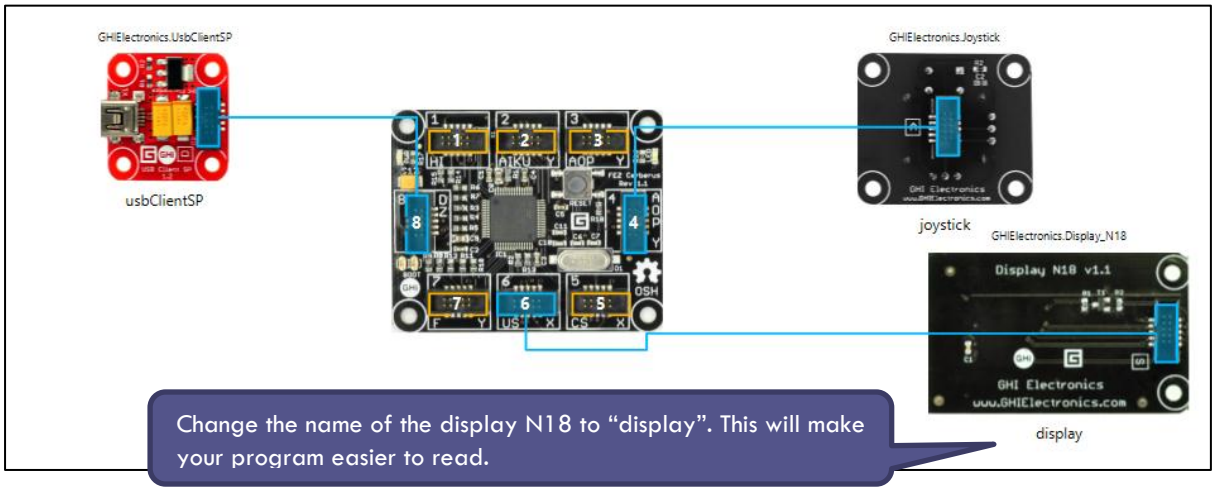


Figure 50: Modules in the Designer for the Reaction game project

Step 3: Write your program

To write your program the first stage is to understand how it will work, in terms of the sequence of events in the game.

There will be two timers. One “ticks” every second – i.e. every 1000 milliseconds – and at each tick produces a new shape. The second reads the joystick position very often, for example 50 times per second. 50 times per second is once every 1/50th of a second which equals once per 2/100th or once per 20/1000th of a second.

The way the program will work is shown below. This is the underlying algorithm for this game.

Every second:

- Check if the player scored by moving to the last shape
- If not, say “Game Over” and stop the game
- If they did then proceed to the following:
  - Choose a random value between 1 and 4
  - Draw the rectangle representing that number (e.g. 1 is yellow, 2 is red, etc.)
  - Start the fast timer (the joystick’s timer)
  - Set correct to False

Once the fast timer (joystick’s timer) is enabled it will repeatedly:

- Check if the joystick has been moved towards the shape
- If it has then set correct to True
- Once correct has been set to True, it will add one to the score
- The fast timer will then stop

If you feel confident that you can write the code for the above algorithm written in English, then go ahead and do so. If not follow these instructions to make the game.

First you will need to declare the variables that you will use throughout the game. You will need a score variable which will be an Integer, a correct variable that will be `True` or `False`, a `Random` object as described above and a `Font` so that you can write the score on the display. Since these variables will be accessed by the different procedures in our program they must be declared outside of any `Sub`, after the `Partial Public Class Program` line. Write the variable definitions as shown below:

```
        Dim font As Font = Resources.GetFont(Resources.FontResources.NinaB)
        Dim score As Integer = 0
        Dim correct As Boolean = True
        Dim random As New Random
        Dim randValue As Integer
```

The next stage is to declare the two timers:

```
      Dim WithEvents timer As GT.Timer = New GT.Timer(1000)  ' once every 1000 milliseconds
      Dim WithEvents joyTimer As GT.Timer = New GT.Timer(20) ' once every 20 milliseconds
```

Create two procedures for the `Tick()` events of these two timers – you will fill the code in for these two later. Create these as you normally would using the drop-down in Visual Studio.

```
Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick

End Sub

Private Sub joyTimer_Tick(timer As Gadgeteer.Timer) Handles joyTimer.Tick

End Sub
```

Figure 51: Four coloured rectangles act as targets in the Reaction game

Next, set up the display to look like the image in Figure 51. On the display there are four rectangles and some text written inside a rectangle shape. On a 128 x 160 pixel display, the code needed for this would be as below:

```
Public Sub ProgramStarted()
    font = Resources.GetFont(Resources.FontResources.NinaB)
     ' Display four blocks on each side of the screen and a message in
     ' the middle
    display.SimpleGraphics.DisplayRectangle(GT.Color.Blue, 1, GT.Color.Blue, 0, 50, 20, 60)
    display.SimpleGraphics.DisplayRectangle(GT.Color.Red, 1, GT.Color.Red, 40, 0, 60, 20)
    display.SimpleGraphics.DisplayRectangle(GT.Color.Green, 1, GT.Color.Green, 40, 140, 60, 20)
    display.SimpleGraphics.DisplayRectangle(GT.Color.Yellow, 1, GT.Color.Yellow, 110, 50, 20, 60)

    display.SimpleGraphics.DisplayTextInRectangle("Ready...", 40, 50, 60, 40, GT.Color.White, font)

End Sub
```

Enter this code and check that it works so far.

The next stage is to write a short procedure that draws just one of these rectangles on the screen, based on a number that is passed into the procedure as a parameter. This procedure will be used with the random number to randomly choose which rectangle to draw on the display. It uses a `Select ... Case` statement.

```
Private Sub showRectangle(number As Integer)
   Select Case number
      Case 1
         display.SimpleGraphics.DisplayRectangle(GT.Color.Blue, 1, GT.Color.Blue, 0, 50, 20, 60)
      Case 2
         display.SimpleGraphics.DisplayRectangle(GT.Color.Red, 1, GT.Color.Red, 40, 0, 60, 20)
      Case 3
         display.SimpleGraphics.DisplayRectangle(GT.Color.Green, 1, GT.Color.Green, 40, 140, 60, 20)
      Case 4
         display.SimpleGraphics.DisplayRectangle(GT.Color.Yellow, 1, GT.Color.Yellow, 110, 50, 20, 60)
    End Select
End Sub
```

This procedure will be called every second, according to the algorithm shown above.

We need a corresponding function to check that the joystick has moved to the correct rectangle. This function will return a Boolean value. The function will return `True` if the joystick movement has gone towards the correct rectangle, which it calculates by looking at the x and y coordinates of the position of the joystick. It will return `False` otherwise.

A joystick in Gadgeteer supports several events and properties, including the following:

Name	Function	What it is for?
`GetPosition.X`	Property	Returns the X coordinate of the joystick as a real number (Double data type). It ranges between -1 and 1, returning a 0 when the joystick is in the middle of is range of movement.
`GetPosition.Y`	Property	Returns the Y coordinate of the joystick as a real number (Double data type). It ranges between -1 and 1, returning a 0 when the joystick is in the middle of is range of movement.
`JoystickPressed`	Event	You can write what you want to happen when the joystick is pressed. In this case we want to start the game by starting the timer.
`JoystickReleased`	Event	As above, you can use this to write what you want to happen when the joystick is released

We need this logic to detect which shape the joystick is moving towards:

- If 1is chosen (blue rectangle) return `correct` as `True` if the `joystick.GetPosition.X < - 0.5` (blue is to the left of the display)
- If 2 is chosen (red rectangle) return `correct` as `True` if the `joystick.GetPostion.Y > 0.5` (red is to the top of the display)
- If 3 is chosen (green rectangle) return `correct` as `True` if `joystick.GetPosition.Y < -0.5` (green is to the bottom of the display.
- If 4 is chosen (yellow rectangle) return `correct` as `True` if `joystick.GetPosition.X > 0.5` (yellow is at the right of the display).

For each of these it is assumed that the joystick is used with the words “Joystick 1.2” at the bottom as shown in Figure 52.

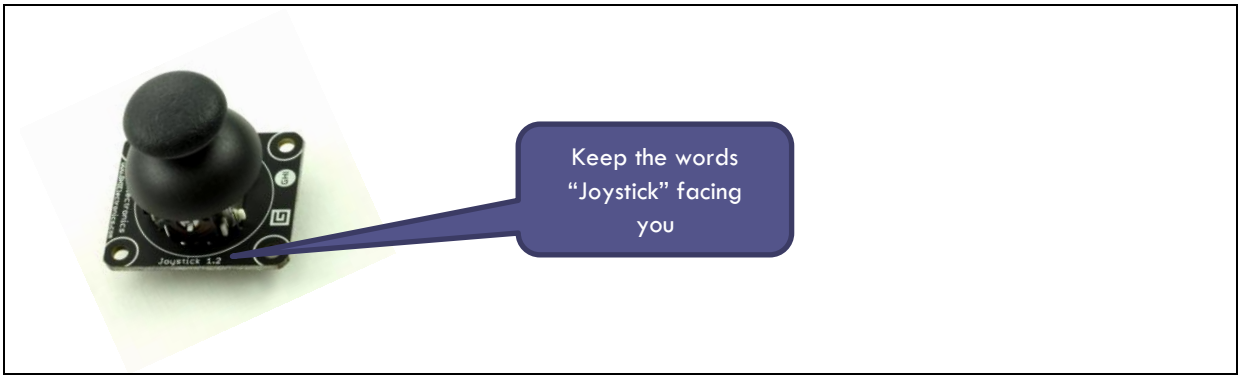


Figure 52: Orientation of joystick

This logic can be put into a function called `detectJoystickMovement()` which returns a Boolean value.

```
Private Function detectJoystickMovement(number As Integer) As Boolean
            ' function to return true or false depending on position of joystick.
            Dim x As Double = joystick.GetPosition.X
            Dim y As Double = joystick.GetPosition.Y
            Select Case number ' random colour chosen by number
                Case 1 ' blue chosen - low x value
                    If x < -0.5 Then
                        Return True
                    Else
                        Return False
                    End If

                Case 2 ' red chosen - high y value
                    If y > 0.5 Then
                        Return True
                    Else
                        Return False

                    End If
                Case 3  ' green chosen - low y value
                    If y < -0.5 Then
                        Return True
                    Else
                        Return False
                    End If
                Case 4 ' yellow chosen - high x value
                    If x > 0.5 Then
                        Return True
                    Else
                        Return False
                    End If
            End Select

            Return False
        End Function
```

The data type of the return value of the function must be included.

'number' is passed into the function as a parameter – this will be the random value for the current round.

The two helper sub-routines have now been written so the next stage is to write the code for the two timers which will call these sub-routines. Looking back to the flow of control described at the beginning of this section you can see that every second when the timer event is called the program needs to do the following:

- Check if the player scored by correctly moving to the last shape
- If not, say "Game Over" and stop the game
- If the player did score then proceed to do the following:
    - Display the score
    - Choose a new random value between 1 and 4
    - Draw the rectangle representing that number (e.g. 1 is yellow, 2 is red, etc.)
    - Start the fast timer (the joystick's timer)
    - Set `correct` to `False` to indicate that the new shape has not been completed yet

The code for each of these steps is given in Table 20 below, including some initialisation.

Table 20: The steps necessary for the reaction timer

Declare variable message
Stop the fast timer
Clear the display
	`Dim message As String`
	
	`joyTimer.Stop() ' stop the other timer`
	`display.SimpleGraphics.Clear()`

Check if the player scored by moving to the last shape	`If Not correct Then`

If not, say "Game Over" and stop the game	`message = "GAME OVER!"`
	
	`display.SimpleGraphics.DisplayText(message, font, GT.Color.White, 25, 40)`
	
	`message = "Score " & score.ToString()`
	
	`display.SimpleGraphics.DisplayText(message, font, GT.Color.White, 40, 60)`
	
	`timer.Stop()`

Otherwise (if they did score) then proceed to the following…	`Else`
Display the score	`display.SimpleGraphics.DisplayText("Score " + score.ToString(), font, GT.Color.White, 40, 60)`
Choose a random value between 1 and 4	`randValue = random.Next(4) + 1`
Draw the rectangle representing that number (e.g. 1 is blue, 2 is red, etc.)	`showRectangle(randValue)`
Set `correct` to `False`	`correct = False`
Start the fast timer (the joystick's timer)	`joyTimer.Start()`

Putting this all together gives the following code:

```
Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick

    Dim message As String
    ' should be true first time and then indicate whether last block was successful
    joyTimer.Stop() ' stop the other timer
    display.SimpleGraphics.Clear()
    If Not correct Then ' didn't get last one
        Debug.Print("Timed out")
        message = "GAME OVER!"
        display.SimpleGraphics.DisplayText(message, font, GT.Color.White, 25, 40)
        message = "Score " & score.ToString()
        display.SimpleGraphics.DisplayText(message, font, GT.Color.White, 40, 60)
        timer.Stop()
    Else
        ' select the next random block to show
        display.SimpleGraphics.DisplayText("Score " + score.ToString(), font, GT.Color.White, 40, 60)
        randValue = random.Next(4) + 1
        showRectangle(randValue)
        correct = False
        joyTimer.Start()
    End If
End Sub
```

The final step is to run the code for the fast timer. The logic behind what happens each 20 milliseconds (checking if the joystick has moved) is as follows:

- check if the joystick has been moved towards the shape;
- if it has then set `correct` to `True`;
- once correct has been set to `True`, add one to the score;
- otherwise continue checking.

This can be written in Visual Basic using the code shown against each statement in the table below.

Table 21: Further steps in the Reaction game

Stop the timer (keeping sthe timer going while checking the joystick movement creates a backlog of timer events)	`joyTimer.Stop()`
Check if the joystick has been moved towards the shape If it has then `correct` is set to `True`	`correct = detectJoystickMovement(randValue)` (`randValue` is the number of the shape passed into the function)
Once correct has been set to `True` …	`If correct Then` (has moved joystick in right direction)
… add one to the score	`score += 1`

Otherwise the fast timer should start again.	`Else`
	`    joyTimer.Start()`
	`End If`

Putting this all together gives the following code inside the `joyTimer_Tick()` function:

```
Private Sub joyTimer_Tick(timer As Gadgeteer.Timer) Handles joyTimer.Tick
    joyTimer.Stop()
    ' keep checking to see if they have moved the joystick
    correct = detectJoystickMovement(randValue)
    If correct Then ' has moved joystick in right direction
        score += 1
    Else
```

```
        joyTimer.Start()
    End If
End Sub
```

Finally, start the timer! This can be done in `programStarted()` if you want the game to start straight away. Alternatively you could use the `joystickPressed()` event to start the timer. The yellow highlighted line in the code below shows you where to add the code to start the timer in `programStarted()`.

```
Public Sub ProgramStarted()

    font = Resources.GetFont(Resources.FontResources.NinaB)
    ' display four blocks on each side of the screen and a message saying click to start

    display.SimpleGraphics.DisplayRectangle(GT.Color.Blue, 1, GT.Color.Blue, 0, 50, 20, 60)
    display.SimpleGraphics.DisplayRectangle(GT.Color.Red, 1, GT.Color.Red, 40, 0, 60, 20)
    display.SimpleGraphics.DisplayRectangle(GT.Color.Green, 1, GT.Color.Green, 40, 140, 60, 20)
    display.SimpleGraphics.DisplayRectangle(GT.Color.Yellow, 1, GT.Color.Yellow, 110, 50, 20, 60)
    display.SimpleGraphics.AutoRedraw = True
    display.SimpleGraphics.DisplayTextInRectangle("Ready...", 40, 50, 60, 40, GT.Color.White, font)

    Thread.Sleep(2000)
    timer.Start()

End Sub
```

Test that the program works correctly! Then try out the exercises below.

EXERCISES

1. Extend your program to include a button and program the button to reset the game when it is over.
2. Extend your program so that it speeds up when you reach a score of 10, 20, 30 etc.
3. Add sound effects to your game by adding a Tunes module.
4. Extend your game to save high scores to the SD card.
5. Devise a new game using random values and moving the joystick.

SUMMARY

In this chapter, you have learned to:

- produce a random number using the `random.Next()` function;
- write a function that returns a Boolean value;
- use the `not` keyword
- use two timers for different purposes.

CHAPTER 12. BUILD YOUR OWN MODULE

KEY TERMS	MODULES YOU WILL NEED	COMPONENTS YOU WILL NEED
Extending Gadgeteer Audio	Cerberus mainboard USBClientSP power module Button module Extender module	Piezo sounder

OVERVIEW

This chapter shows you how to build your own module! Using an Extender module (or a Breakout module) as a starting point you can easily connect electronic components to a Gadgeteer mainboard. Here we create a simple buzzer module which is like the GHI Electronics Tunes module. There are many extension exercises that lead from this project.

TUTORIAL 1: MAKE A NOISE!

Step1: Assemble these modules

Cerberus mainboard

usbClientSP power module

Joystick

Extender module

Figure 53: Modules needed for Build your own module project

Step2: Examine your piezo sounder

In addition to Gadgeteer modules, for this project you also need an electronic component called a "piezo sounder" or sometimes referred to as a "piezo transducer". From the outside this looks like a round piece of plastic, usually black or white. It has a small hole on one side (where the sound emanates from!). The other side will either have two metal legs or two flexible wire leads. In this project it will be easier to use a piezo which has the flexible wire leads. These components are fairly cheap – perhaps £1-£2 – and can be found in electronics shops and online. Be careful to use a sounder which is designed to be driven at 3-5V and at a range of frequencies. Suitable parts are shown in Figure 54:

Maplin FM59P | Farnell 1827943 | RS 498-2596

Figure 54: Some example piezo sounders which will work with this project with the supplier and order number for each. All are available online and Maplin also have high street retail park stores in the UK.

Step 3: Use the Gadgeteer Designer to link the modules together

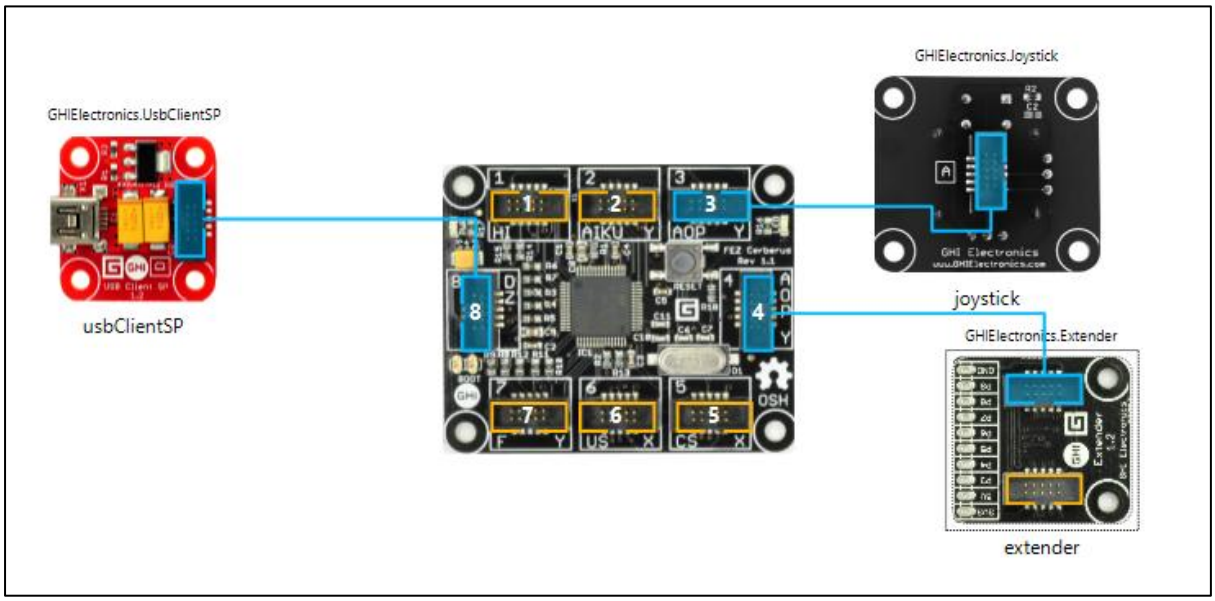


Figure 55: Modules in the Designer for the Build your own module project

Step 4: Connect the piezo sounder to the extender module

For this project once the Gadgeteer modules are connected to the mainboard you will also need to connect the flying leads on your piezo sounder to two of the electrical connection points on your Extender module. One lead needs to connect to the connection point marked 'GND' and the other lead connects to the point marked 'P9'. The sounder will work whichever way around you connect the leads, but it's good practice to connect the black lead to the 'GND' because this carries the more negative voltage. Similarly, connect the red lead to the 'P9' connection point.

Your piezo sounder is probably supplied with just 3-4mm of bare wire protruding at the end of each flying lead. This is perfect for soldering the end of each wire to the Extender module connection points, but if you don't have access to a soldering iron you can instead simply thread each bare wire end through the appropriate hole in the connection point, bend it around and twist it tightly. It's important that the bare wire makes good contact with the metal of the connection point and that the two wire leads do not touch each other. You may want to start by removing around 10-15mm of the plastic insulation on each wire because there will probably not be enough bare wire otherwise. You can do this with wire strippers, a knife or scissors if you are careful not to cut the wire, only the plastic insulation.

Soldering the piezo sounder to the Extender module is the most robust way of attaching it. Push the bare ends of the flying leads through the holes on the extender module, heat with a soldering iron and apply solder to leave a shiny joint. See for photos of what this should look like.

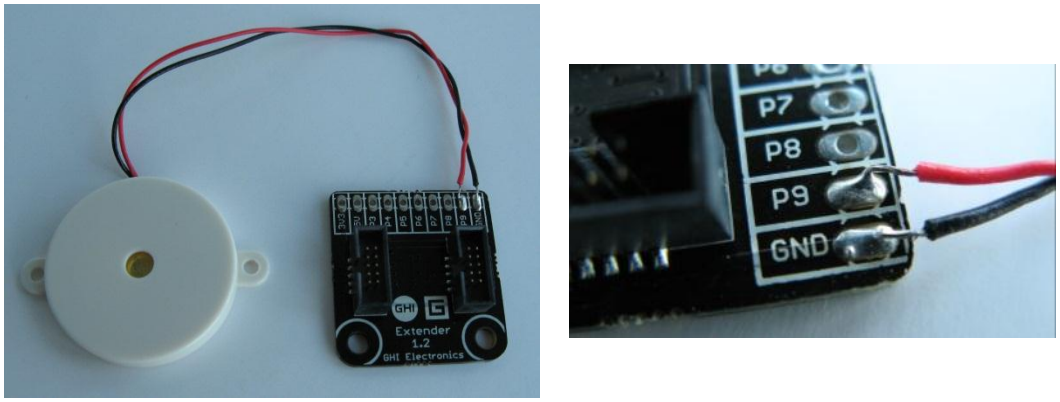


Figure 56: Soldering the piezo to the extender module

However, rather than soldering it may be easier to simply pass the bare ends of the flying leads through the holes in the Extender module connection points, bend them around and twist them tight, ensuring a good metal-to-metal contact and making sure the two leads do not touch each other.

Twisting the flying leads together is a good way to keep things neat.

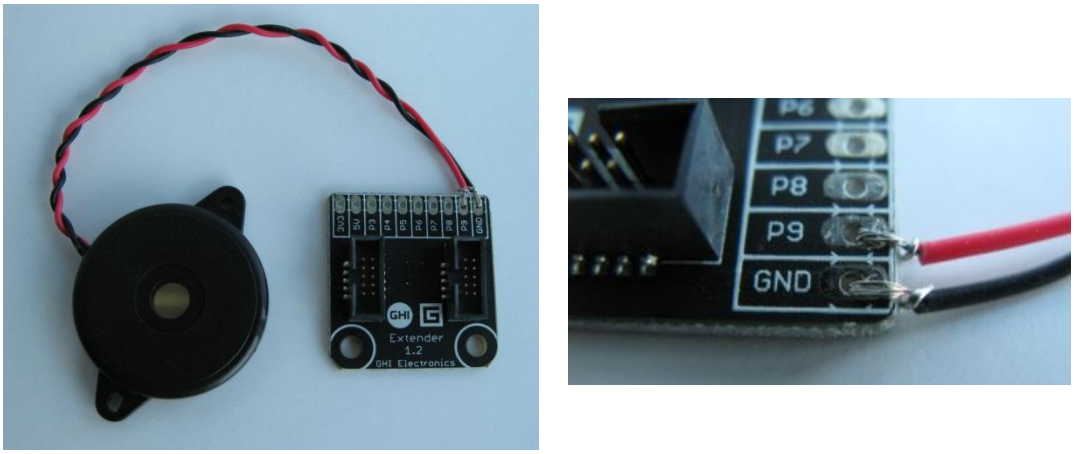


Figure 57: If you don't want to solder you can simply twist the wires in place if you are careful

Step 5: Write your program

As you may have noticed when using the designer in Step 3, the socket on the top of the extender module is compatible with *every* socket on the mainboard! This is why it doesn't have any letters written on it to indicate compatibility – it works with any socket. It is a general-purpose "building block" module with a large variety of potential uses.

For this project we are extending the functionality of Gadgeteer by creating a new type of module. Depending on where we plug the Extender into the mainboard we will be able to interface to (i.e. connect up) different types of electronic components. In this case we have chosen socket 4 on the Cerberus mainboard, and the letters 'A', 'O', 'P' and 'Y' indicate three basic ways of interfacing that we can choose between. We are going to use this socket in its 'P' mode of operation – 'P' stands for 'Pulse Width Modulation output' or more simply 'PWM output'. This lets us create a 'PWM' signal which is continually changing between two different voltages in a repetitive (or periodic) way.

When we generate a PWM signal on socket 4 and connect the piezo sounder it will turn into a sound we can hear. The frequency of the PWM signal is the same as the pitch of the note which the buzzer plays. As a reminder we used frequencies with the Tunes module in Chapter 3. To get an idea of a suitable value for this frequency, it's useful to know that the human hearing range goes from around 20 Hz to around 15,000 Hz, though the buzzer probably can't play frequencies as low as 20Hz.

The first thing we will do is generate a fixed tone:

- we need to create a buzzer object and specify that it is going to use Gadgeteer's PWM Output interface;
- the buzzer object is then set up so that it works with the Extender module we have plugged in, and we have to specify which pin of the Extender we will connect our piezo sounder to;
- finally we can set a pulse waveform going.

First, declare the buzzer:

```
Namespace GadgeteerApp1
    Partial Public Class Program
        ' Specify that the buzzer will use the Gadgeteer PWM Output interface
        Dim buzzer As GT.Interfaces.PWMOutput
```

Then add the two additional lines outlined below along with appropriate comments:

```
        Public Sub ProgramStarted()
            Debug.Print("Program Started")

            ' set up the PWM Output to use pin 9 of the Extender module
            buzzer = extender.SetupPWMOutput(GT.Socket.Pin.Nine)

            ' set a repetitive pulse going
            buzzer.Set(2000, 0.5)
        End Sub
```

Now test that the program works so far.

When you run it you should hear a constant pitch sound coming from the sounder. If it doesn't work check that the ends of the two flying leads are connected to the metal part of the Extender module connection points. Once you have it working, the buzzer won't stop until you do one of the following:

- disconnect the piezo or Extender module;
- remove power from the Gadgeteer mainboard; or
- program the mainboard to switch off the PWM Output with `buzzer.Active = False`

TUTORIAL 2: USE THE JOYSTICK TO CONTROL THE PITCH AND DURATION

Step 1: Extend your program.

The next stage is to start and stop the buzzer programmatically. We'll use the button functionality that is built into the joystick module to turn on and off the buzzer. This is done using the `JoystickPressed` event handler to toggle a state variable `buzzerOn`.

We also want to control the pitch of the sound. This is done by altering the first parameter (the frequency) of the `buzzer.Set()` method. We can use a timer every 200 milliseconds (0.2 seconds) to read the joystick and set the frequency appropriately, as long as `buzzerOn` is `True`. The joystick is read using `joystick.GetPosition().X` to get a value between -1 and 1 depending on the X position of the joystick.

The code to do this is below, and goes after `ProgramStarted`:

```
  Dim WithEvents timer As GT.Timer = New GT.Timer(200)
  Dim buzzerOn As Boolean = False

  Private Sub joystick_JoystickPressed(sender As Joystick, state As Joystick.JoystickState) Handles jo
ystick.JoystickPressed
      If (buzzerOn) Then
          buzzerOn = False
          Debug.Print("Buzzer off")
          buzzer.Active = False
      Else
          buzzerOn = True
          Debug.Print("Buzzer on")
          timer.Start()
          ' buzzer will actually be turned on by timer_Tick
      End If
  End Sub

  Private Sub timer_Tick(timer As Gadgeteer.Timer) Handles timer.Tick
      If (buzzerOn = False) Then
          timer.Stop()
          Exit Sub
      End If

      ' define useful constants for calculating the PWM frequency settings
      '
      ' the maximum frequency we want to play
      Const maxFrequency = 15000

      ' the minimum frequency we want to play
      Const minFrequency = 100

      ' the middle of the range of frequencies
      Const middleFrequency = (maxFrequency + minFrequency) / 2

      ' half of the frequency range, i.e. how much we should  deviate from the middle frequency
      Const halfRangeOfFrequency = (maxFrequency - minFrequency) / 2

      ' calculate the new frequency to use based on the joystick X position
      '
      ' X position is between -1.0 (fully left) and 1.0 (fully right)
      ' so we convert this to a number between minFrequency and maxFrequency
      Dim newFrequency As Double = middleFrequency + (joystick.GetPosition().X * halfRangeOfFrequency)

      ' set the PWM pulse output to the new frequency
      buzzer.Set(CInt(newFrequency), 0.5)

      ' and print out newFrequency
      Debug.Print(CStr(newFrequency))
  End Sub
```

In addition to adding the code above, don't forget to remove the `buzzer.Set(2000, 0.5)` line you used in the first exercise since this is no longer needed. In this project the buzzer will start to make a noise the first time that the joystick is pressed. Note that the volume of the buzzing will change depending on the frequency – the buzzer is particularly efficient at some frequencies and so it will be much louder.

EXERCISES

1. You will notice a stuttering sound to the buzzer due to the PWM being changed every 200ms (fifth of a second). Edit your program so that if the new frequency is only a tiny bit different from the old one (which you'll have to store in a variable), then the call to `buzzer.Set()` is omitted, to eliminate the stutter.
2. Edit your program so that it only plays notes from one octave on a piano keyboard. Wikipedia has a "Piano key frequencies" page. Store the frequencies in a constant array and select which to use based on the position of the joystick.
3. You may notice that the values returned by a joystick can be quite noisy – this means they vary a bit each time even if the joystick isn't being moved. Extend your program to keep a running average of the last 10 joystick values using an array and use this to control the pitch.

SUMMARY

In this chapter, you have learned to:

- use the Extender module to create a new type of module, a piezo sounder;
- generate a pulse-width modulation (PWM) signal which causes the sounder to emit a constant tone;
- control the pitch of the tone using a Joystick module.

APPENDIX A. WHERE TO BUY .NET GADGETEER

The following list is intended as a guide and is valid at the time of writing. It is not a complete listing of all manufactures and suppliers. This is because the platform is open-source and anyone can build and distribute .NET Gadgeteer hardware. Note that some suppliers only ship to certain geographical areas and so may not be applicable in your location.

NAME	MAINBOARD MANUFACTURER	MODULE MANUFACTURER	SUPPLIER
Amazon http://www.amazon.com			●
Antratek Electronics http://www.antratek.com			●
Australian Robotics http://www.australianrobotics.com.au			●
Cool components http://www.coolcomponents.co.uk			●
DFRobot http://www.dfrobot.com		●	●
Génération Robots http://www.generationrobots.com			●
GHI http://www.ghielectronics.com	●	●	●
Ingenuity Micro http://ingenuitymicro.com	●	●	●
Let Elektronik http://www.let-elektronik.dk			●
Lextronic http://www.lextronic.fr			●
Love Electronics http://www.loveelectronics.co.uk	●	●	●
Micro mint http://micromint.com	●		●
Mountaineer Boards http://www.mountaineer-boards.com	●		●
Mouser http://mouser.com			●
Proto pic http://proto-pic.co.uk			●
Robot Shop http://www.robotshop.com			●
SaveComm mfDevices http://savecomm.net			●
Seeed Studio http://www.seeedstudio.com		●	●
Solder Monkey http://www.soldermonkey.net		●	●
Sparkfun Electronics http://sparkfun.com			●
Sytech http://sytechdesigns.com	●	●	●
UAM Czech Republic http://shop.microframework.eu			●
Watterott Electronic http://www.watterott.com			●

APPENDIX B. INSTALLING THE "FEZ CERBERUS TINKER KIT"

Before you get started with your Fez Cerberus Tinker Kit, you need to install some software on your computer. The easiest way to achieve this is to visit the GHI website (http://www.ghielectronics.com) and follow the latest installation instructions. Below is an overview of the process. Note that the order of installation is important.

There are three pieces of software required:

1) In order to edit code and deploy applications to hardware you need an Interactive Development Environment (IDE), in this case **Visual Studio**. Different versions of Visual studio are supported, the most common is the latest free version called Visual Studio Express 2012 for Windows Desktop. (http://www.microsoft.com/visualstudio/eng/downloads). Once installed you should be able to select "VS Express for Desktop" from the start menu and the IDE will load as shown in Figure 58.

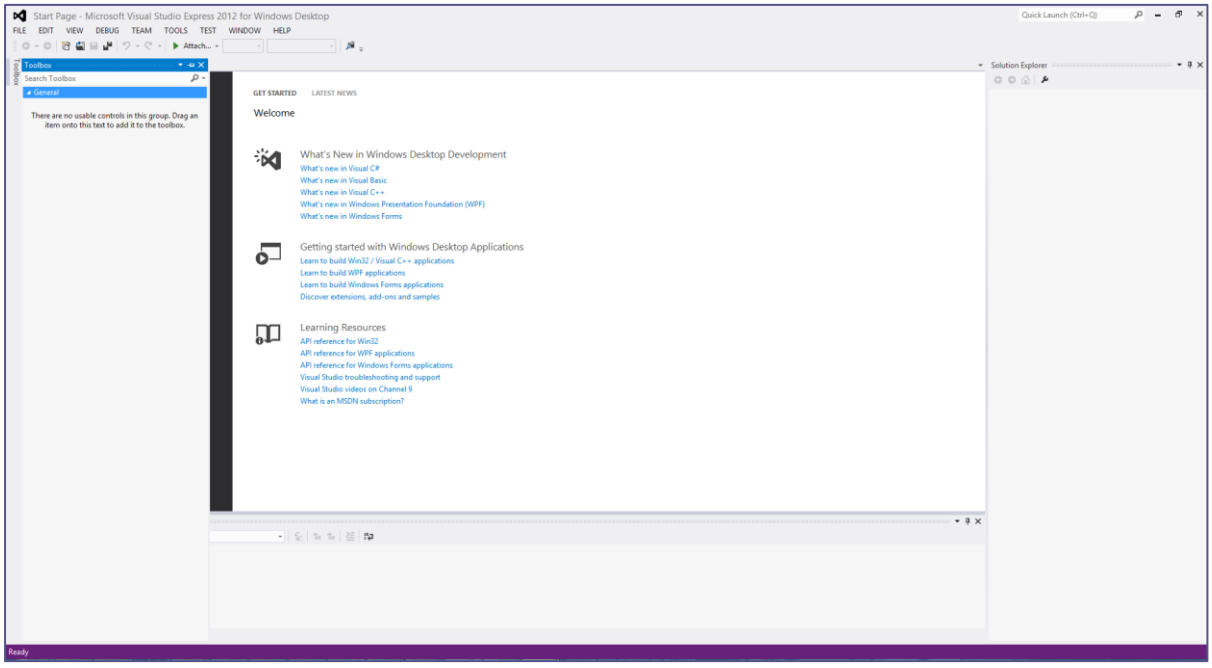


Figure 58: Visual Studio Express 2012 for Windows Desktop

2) The **.NET Micro Framework (NETMF)** is the core set of libraries upon which .NET Gadgeteer is based. It can be downloaded from https://www.ghielectronics.com/support/.net-micro-framework , but it can also be found on http://www.netmf.com. Once installed (or if you wish to check that it is installed) you can see NETMF in "Add or remove programs" as shown in Figure 59. At the time of writing the version you will have installed will depend on which version of Visual Studio you wish to use.

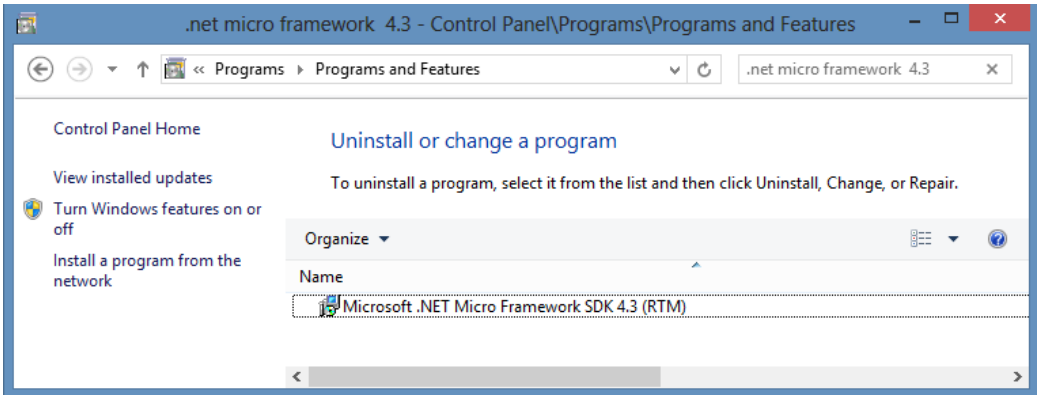


Figure 59: Check to see the NETMF is installed by selecting "Add or remove programs"

3) The final installation step is for the **Software Development Kit (SDK)** for the hardware you are using as well as for the .NET Gadgeteer platform. Fortunately GHI wrap both these installs into a single package (available at https://www.ghielectronics.com/support/.net-micro-framework). The GHI installer is downloaded as a zip file. Please right-click this file and select Extract All to a temporary folder somewhere on your computer and then run setup.exe. If you try to run setup.exe directly from the zip file it can cause problems.
Later you can make sure that the software installed correctly by checking "Add or remove programs". You should see the **GHI .NET Gadgeteer SDK** (Figure 60) and the **Microsoft .NET Gadgeteer Core** (Figure 61).

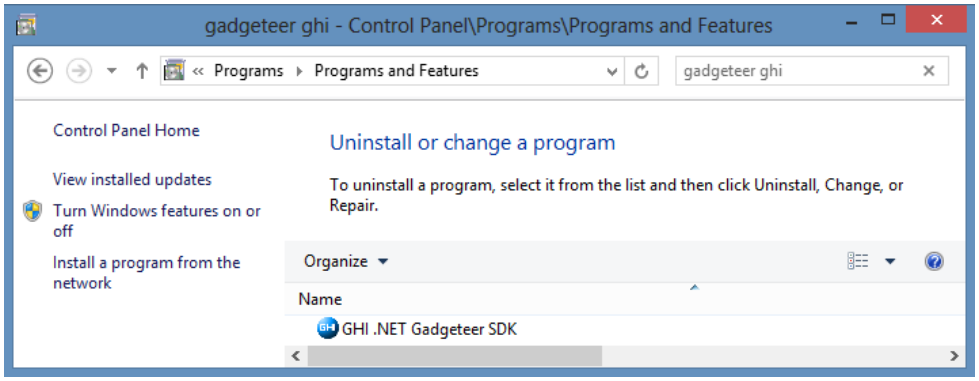


Figure 60: "Add or remove programs" shows that the GHI .NET Gadgeteer SDK is installed

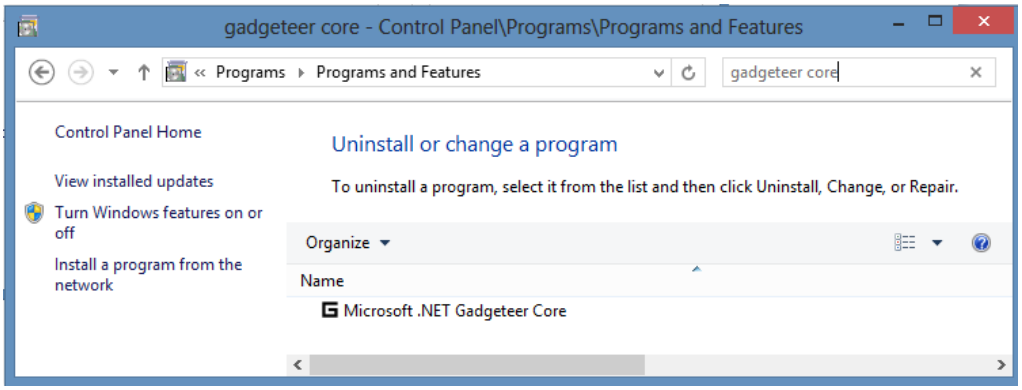


Figure 61: "Add or remove programs" shows that the Microsoft .NET Gadgeteer Core is installed

APPENDIX C. GETTING TO KNOW VISUAL STUDIO

THE SOLUTION EXPLORER WINDOW

The Solution Explorer window is on the right hand side of the Visual Studio window by default. It provides a list of all the files and resources in your solution (a collection of one or more projects). In this book, a solution is always made up of a single project, for example the Clicker project shown in Figure 62. If this window is not visible it can be enabled by selecting "Solution Explorer" from the View menu.

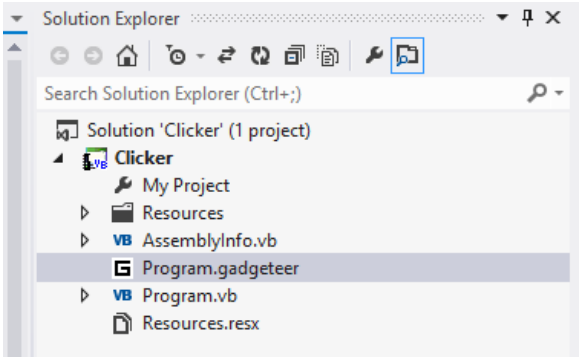


Figure 62: Solution Explorer window for the Clicker project

THE TOOLBOX WINDOW

The Toolbox window appears on the left hand side of the screen by default. When you have the Designer open, the Toolbox will display all the hardware modules that you can potentially connect to your project. The Toolbox can take a few seconds to populate but if it is empty be sure that you have the Designer open as it will be empty when the source code is visible. If you cannot find the Toolbox or it has been closed, it can be reopened by selecting Toolbox from the View menu.

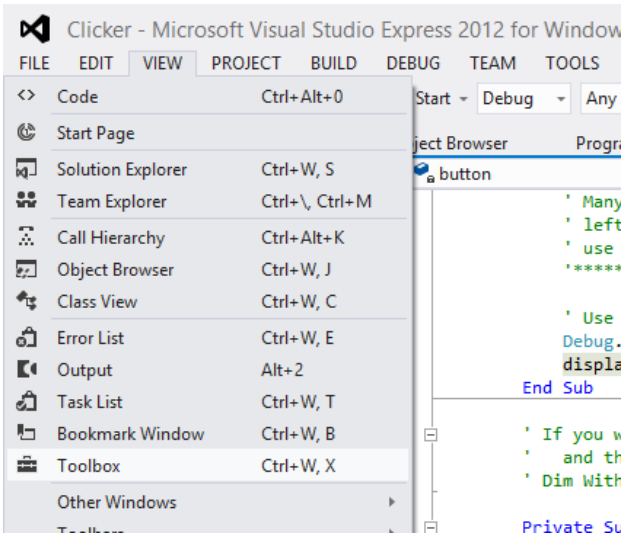


Figure 63: Opening the Toolbox from the View menu in Visual Studio

The Designer window

When you first create a .NET Gadgeteer project in Visual Studio the Designer is opened and visible. The Designer window is where you drag and drop modules to and connect your hardware. If you close this window and need to reopen it, have a look in the Solution Explorer window on the right hand side of the screen as shown in Figure 64. Clicking on "Program.gadgeteer" will re-open the Designer.

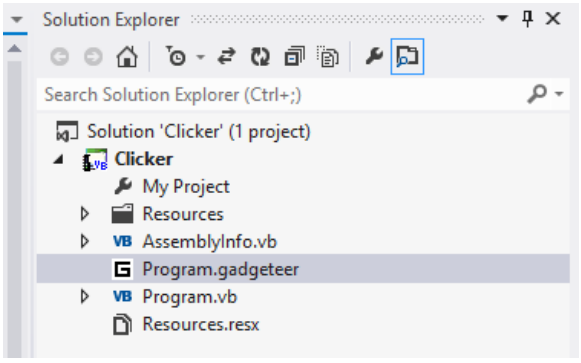


Figure 64: Solution Explorer window for the Clicker project

The Output window

It is always handy to have the Output window displayed in Visual Studio. Whilst this may appear obvious it is not on by default in some versions of Visual Studio. Select Output from the window on the Debug menu as shown in Figure 65. This window provides information about building and deploying your application to the hardware as well as displaying any debug messages you have inserted into your code.

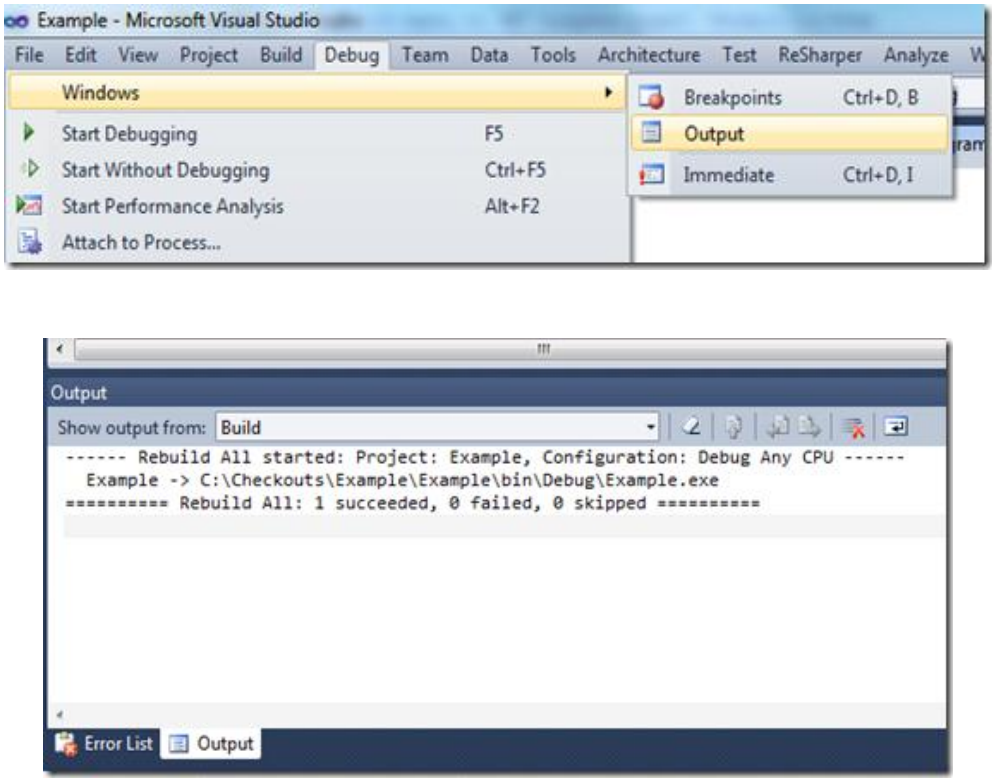


Figure 65 : Output window showing a .NET Gadgeteer project, successful build and deploy

IntelliSense in Visual Studio

IntelliSense is a collection of useful features that Visual Studio supports to assist you with coding. There are a few features that are essential to learn as they help you type faster and more accurate code as well as indicating where potential problems may arise. Below is a summary of relevant features.

Word completion

Word complete is used to automatically finish off words for you. For example, if you work on a Gadgeteer project with just a button module displayed in the Designer, then when you start typing code Visual Studio will produce a drop down menu showing a number of relevant options. In Figure 66, typing the first few letters of `button` will produce a menu with the correct option at the top. You can use the mouse to select the correct option from this menu, but it is quicker to press the Tab key on the keyboard. Tab will autocomplete the word with the top item of the displayed option list. If you ever need to see the option list again (perhaps you pressed space or clicked on the wrong option) you can press the keys 'Ctrl' and 'Space' on your keyboard.

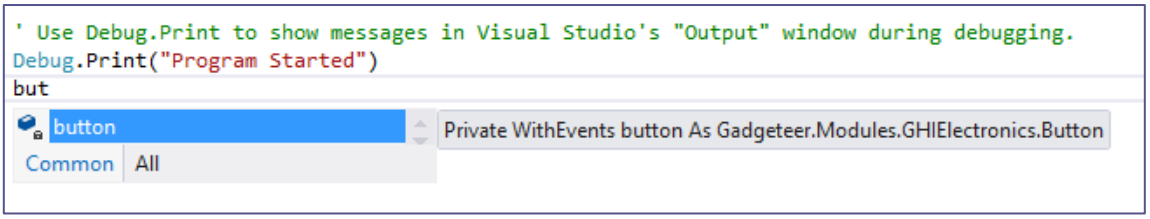


Figure 66: Word completion in Visual Studio

Listing members

There are properties and methods associated with objects, for example, a button object. By adding a dot ('.') after the object name, Visual Studio will display a drop down menu of all those properties and methods. The icon next to a member of the menu lets you know what type it is (properties are and methods in Visual Studio 2012). When a member has been selected, either by typing or by using the mouse or arrow keys, Visual Studio will show a QuickInfo description next to the dropdown. This tells you a little about the selected item and is a great way to explore the capability of hardware modules. For example, you can discover that the button module also has an LED on it which you can turn on and off and check the current status (see Figure 67).

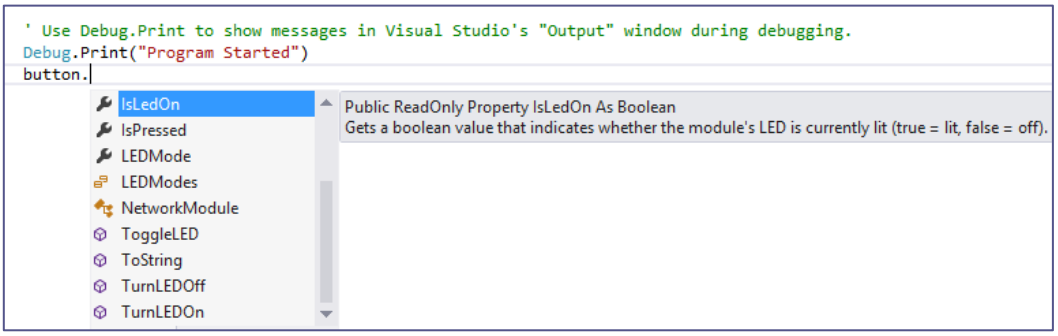


Figure 67: On the left, a list of button members, showing properties, methods and events. To the right more details of the selected member are shown.

Parameter Info

Some methods require parameters that provide some extra information needed to execute the method. For example, the `Debug.Print` method requires the text that you wish to print. If you are unsure about which parameter(s) `Print` requires, then when you type the opening bracket, Visual Studio will show more information. In Figure 68, the Print method takes a single parameter which is a string.

Figure 68: Parameter Info box showing that the Print method expects a single string parameter.

Let us look at a more complicated example, using the Tunes module. Some methods have more than one signature, meaning that you can supply different sets of parameters depending on which signature you choose. For example, to make a noise on the Tunes module you call the `Play` method. The parameter info box looks slightly different as there are 4 different ways to call this method. Figure 69 shows that you can call the `Play` method without any parameters, but that you must first add notes using the `AddNote` method.

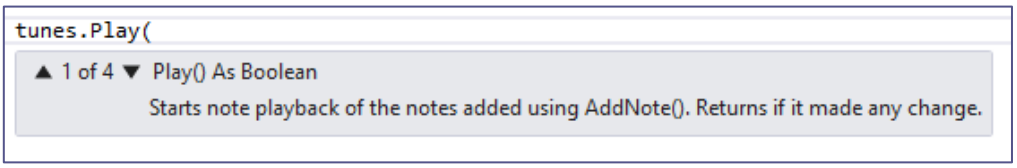


Figure 69: Parameter info for a method with multiple signatures

You can look at the different signatures by using the up/down arrows ▲ 1 of 4 ▼. For example, if you wanted to play a note based on a frequency you can simply supply a single integer (number).

Figure 70: Playing a note based on a frequency

However if you created a melody as shown in Chapter 3 then you can supply the melody to the Play method.

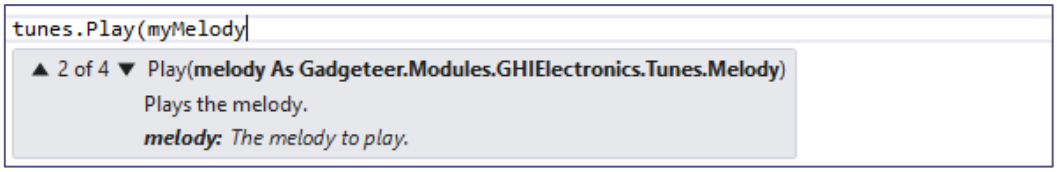


Figure 71: Playing a melody using the Play method on the tunes module

APPENDIX D. UPDATING THE FIRMWARE

The software that you install from the hardware manufacturer (SDK) must match the software that is running on the mainboard (the so-called firmware). There should not be many updates and it is best to stick to the same version of the SDK and firmware across all your devices and computers, unless there is a particular problem to solve. If the software on the computer and the firmware get out of sync you will see a *Firmware version does not match managed code version* error in the Output window. You may need to scroll to the top of the output window to see the error.

If you do not have the output window showing then this problem is harder to identify as the deploy will look as it is doing nothing, but really it has failed and put the error on the Output window and halted the deployment.

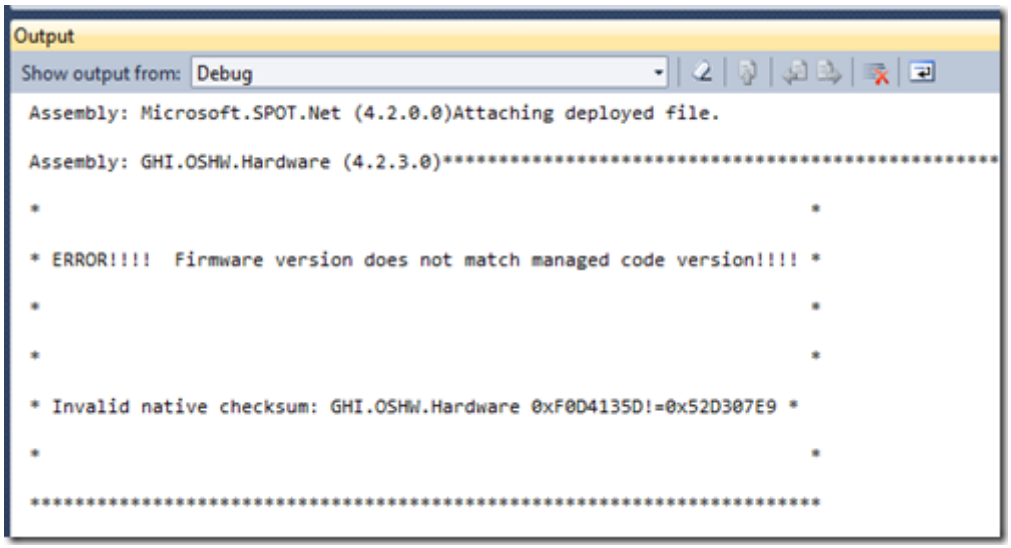


Figure 72 : Example of a firmware version mismatch error in the Output window

If you need to update the firmware to solve an issue or due to a mismatch you will need to follow the manufacturer's updating instructions. In the case of the FEZ Cerberus Tinker Kit, there is a tool called the "*FEZ Config*" that will let you update your hardware to the latest firmware version. This is the recommended option and should be used to perform any software updates to your FEZ Cerberus mainboard. A summary is shown on page 100.

For those that wish to know more or are having issues updating the firmware there are a few manual steps that can confirm firmware versioning and perform an update. The firmware comes in two parts, the TinyBooter (page 100) and the TinyCLR (Page 103). Both have to be updated separately, although the TinyBooter does not need updating as often. Before you do any update, check the version number to be sure you need to update.

CHECKING THE FIRMWARE VERSION

Depending which version of NETMF you have installed look inside the Tools folder for a tool called MFDeploy.exe, for example, `C:\Program Files (x86)\Microsoft .NET Micro Framework\v4.2\Tools` or `C:\Program Files (x86)\Microsoft .NET Micro Framework\v4.3\Tools`

Connect your mainboard to your computer via USB and load MFDeploy. Select USB and you should see your device listed. To test that everything is connected properly press the ping button and you should see the response "*TinyCLR*" as shown in Figure 73.

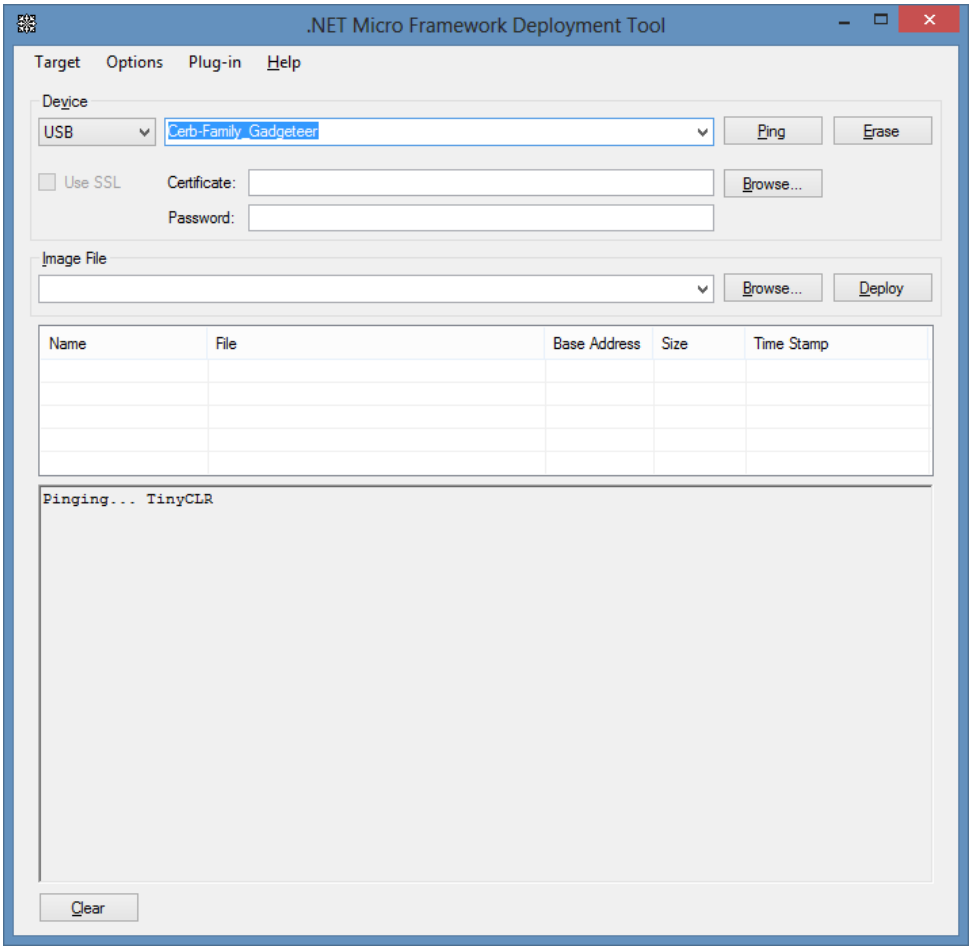


Figure 73: MFDeploy showing the FEZ Cerberus mainboard connected via USB.

Next select "*Device Capabilities*" from the Target menu. This will query your device to see which version of the firmware is currently running on your hardware as shown in Figure 74. If your firmware is out of date, you may want to consider updating it.

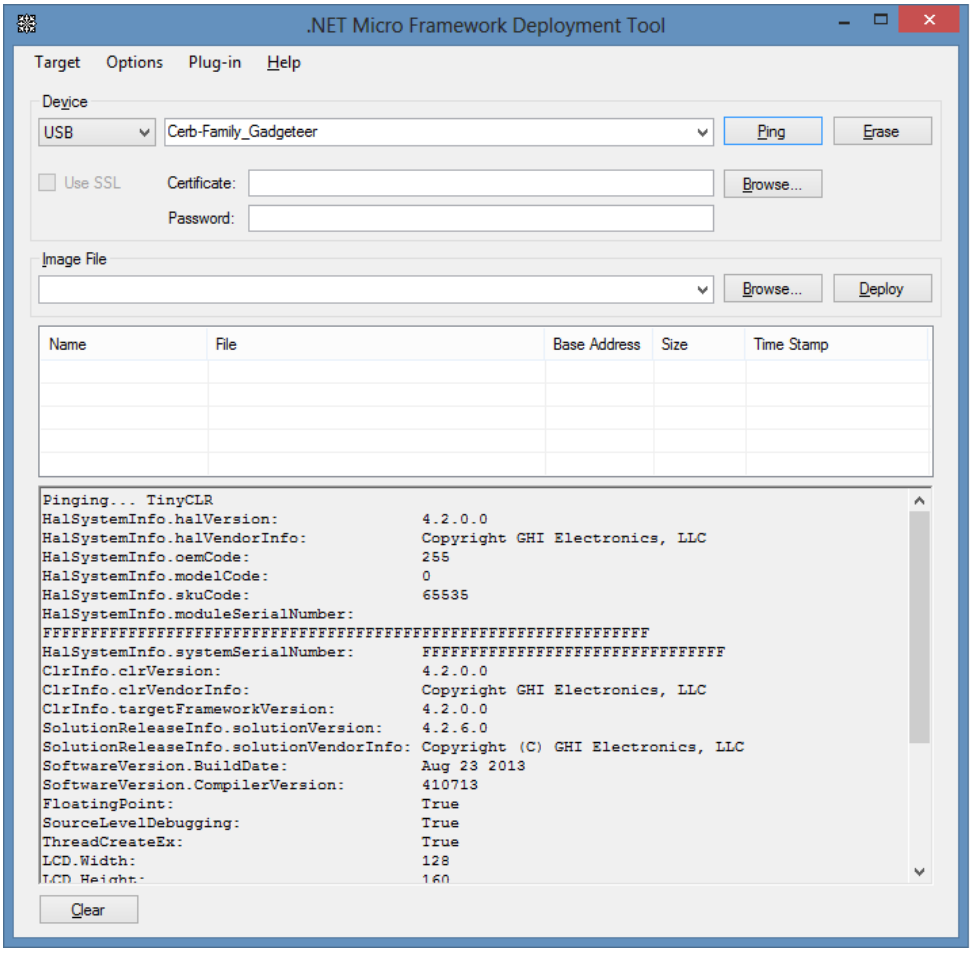


Figure 74: Fez Cerberus showing that the firmware dates from 23 August 2013 and is version 4.2.6.0

UPDATING THE FEZ CERBERUS FIRMWARE USING THE FEZ CONFIG TOOL (RECOMMENDED)

At the time of writing the FEZ Config firmware updater is NOT distributed with the GHI SDK and needs to be downloaded separately from GHI (https://www.ghielectronics.com). Once downloaded, run the config tool and you will see a screen similar to that shown in Figure 75.

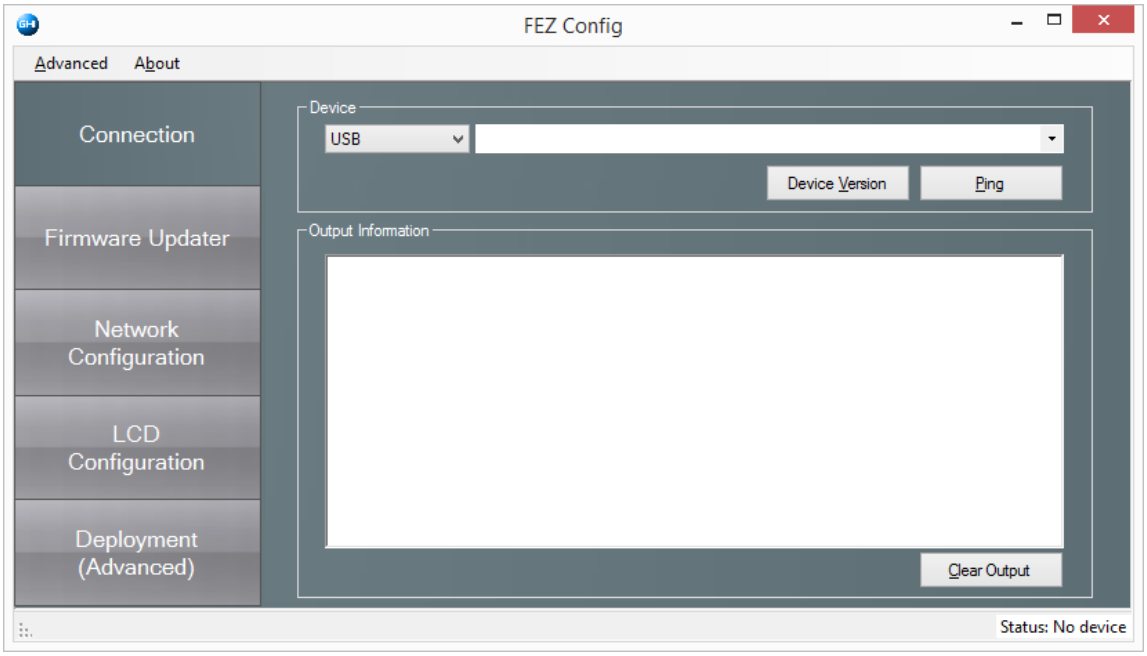


Figure 75: The FEZ Config tool without any devices attached (you need to connect your mainboard using the USB cable)

1) Connect your mainboard using the USB cable and select the device.

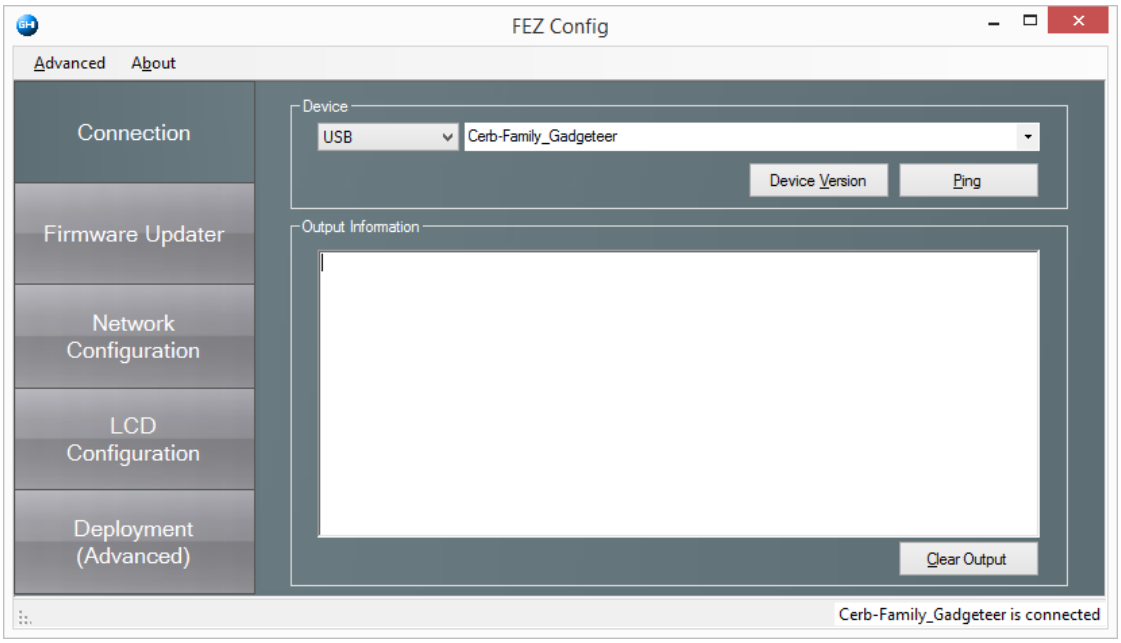


Figure 76: The FEZ Config tool with Cerberus connected as indicated in the bottom right corner

2) Next click on the "Firmware Updater" button on the left, to display a list of firmware updates for different devices. Select "FEZ Cerberus". Ticking the Ethernet option selects different firmware depending on if you want to use a network module. Default to un-ticked unless you really need networking.

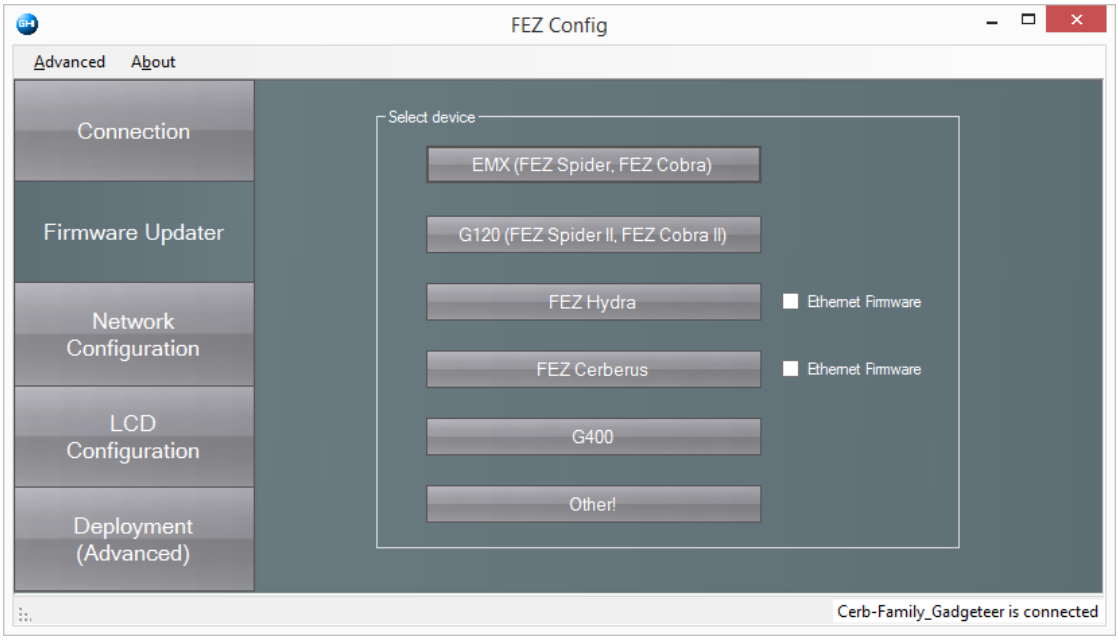


Figure 77: The FEZ Config tool can update different mainboards. Be careful to select the correct mainboard

3) The next screen confirms which files will be loaded. Leave as the default and select Next. Agree to any warnings about loss of data (it will erase the device) and follow the menus.

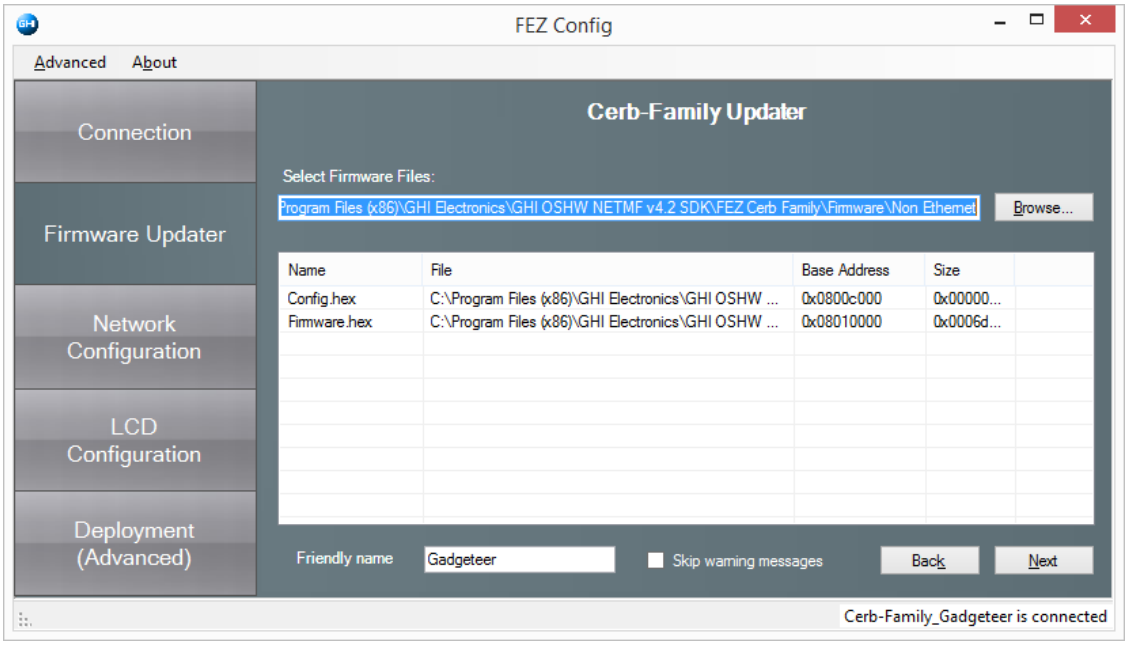


Figure 78: The FEZ Config tool showing the location of the firmware to be flashed

4) Wait while the device is being updated, you will get a progress indicator.

described on page 100. This is the recommended option and should be used to perform any software updates to your FEZ Cerberus mainboard.

To manually update the TinyBooter, you will need the latest TinyBooter (usually a DFU file) and the STDFU Tester application installed.

GHI distribute the STDFU Tester application with the firmware for example `C:\Program Files (x86)\GHI Electronics\GHI OSHW NETMF v4.2 SDK\FEZ Cerb Family\Firmware\STM_DFU.zip`. And if not already installed, you will need to run the installer and load the STDFU Tester application as shown in Figure 81.

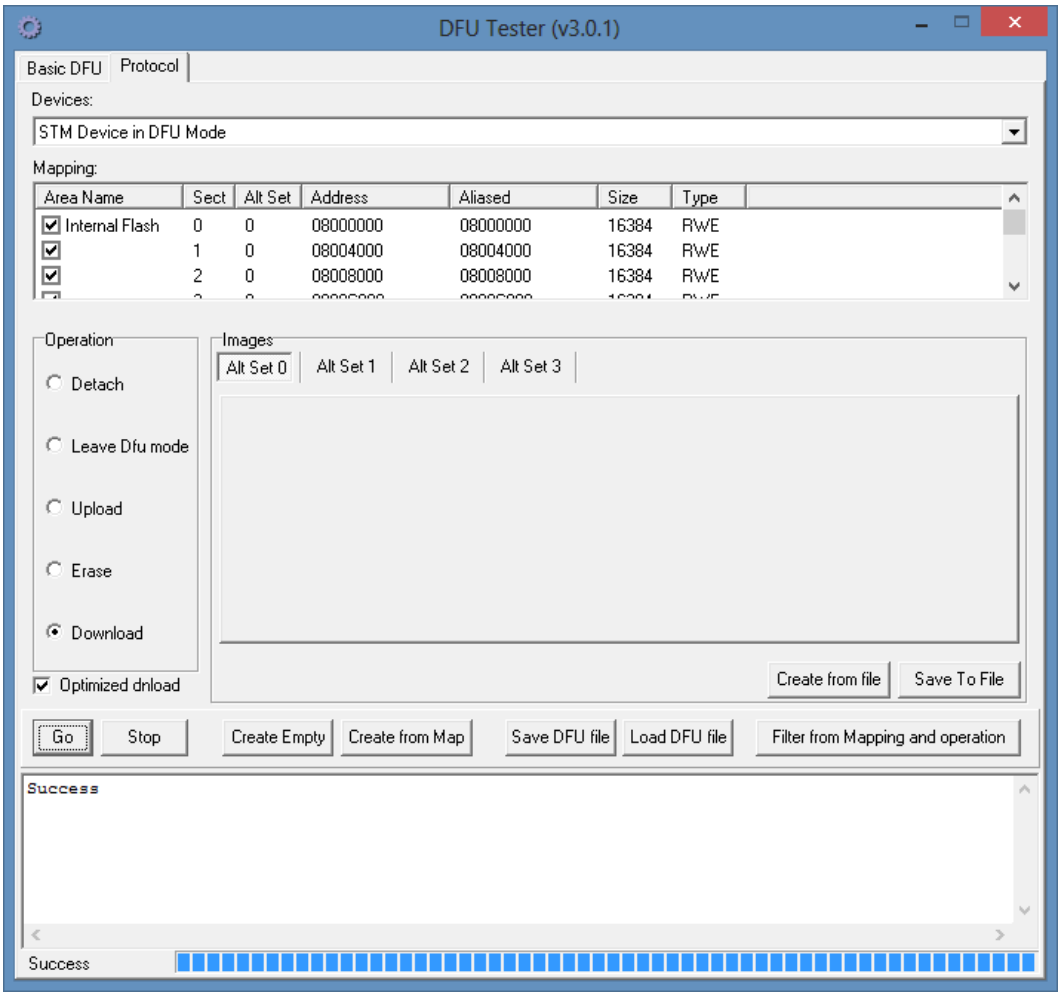


Figure 81: STDFU Tester application showing the Cerberus mainboard in boot mode

The FEZ Cerberus mainboard needs to be started in boot mode. This is achieved by starting the device (powering up) with the BOOT button on the mainboard pressed. Do not hold the BOOT button down for longer than a few seconds. If the device has restarted in boot mode it will appear in the STDFU Tester application devices menu as shown in Figure 81. Updating the TinyBooter has two steps, erase and updating. Be sure you are familiar with the process below as it can break your hardware. GHI have more complete instructions online.

Erase

1) Ensure you can see the device in the devices menu at the top of the SDTFU Tester application.
2) Select the protocol tab (top of screen).
3) Press the "Create from Map" button on the Protocol menu.
4) Tick the "Erase" radial.
5) Press Go.

Updating

1) Press the "Load DFU file" button from the protocols tab.
2) Tick the "Download" radial
3) Press Go

UPDATING THE FEZ CERBERUS TINYCLR MANUALLY

If you need to update the firmware either to solve an issue or due to a mismatch you will need to follow the manufacturer's updating instructions. In the case of the FEZ Cerberus Tinker Kit, use the FEZ Config tool as described on page 100. This is the recommended option and should be used to perform any software updates to your FEZ Cerberus mainboard.

To manually update the TinyCLR, open MFDEploy and ping your device (see page 98) to check that all is working and ensure that an update is required. Next you will need to find the latest firmware files to install, typically visit the GHI website, or check your SDK install directory, for example `C:\Program Files (x86)\GHI Electronics\GHI OSHW NETMF v4.2 SDK\FEZ Cerb Family\Firmware`. In this case there are two firmwares depending if you require Ethernet support.

The firmware comprises of two files a config and a firmware file (there could be 2 further signature files, but ignore these). You need to select these image files by selecting the browse button for the "*Image file*" selector, navigating to the directory with your new firmware and selecting both the Config.hex and the Firmware.hex files.

With both files selected press the deploy button.

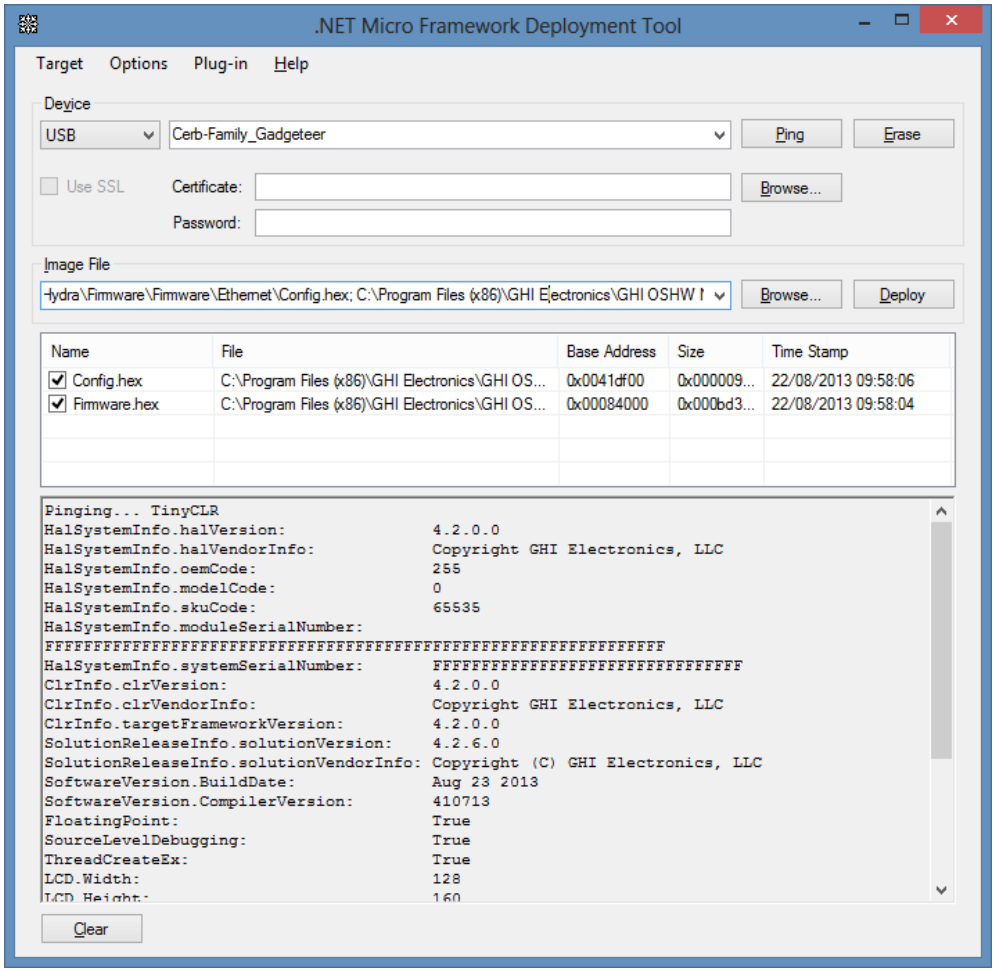


Figure 82: Selecting both the config and firmware files, pressing deploy will update the TinyCLR

APPENDIX E. HOW TO DEBUG VISUAL BASIC IN VISUAL STUDIO

The debugger is a very useful feature in Visual Studio. It lets you inspect your program while it is running and is an invaluable tool to help solve those annoying bugs where you don't understand the behaviour of your program. The debugger has a series of different capabilities. In this section we cover only the basic capabilities but as your programs become more complicated the topic is worth further investigation.

All the debugger examples in this section are based on the code for the Clicker project shown on page 23. When the example code is opened in Visual Studio make sure to check that it compiles and deploys, and every time you press the hardware button the number on the screen increments on the display.

When you run your code by pressing the play button or by pressing F5 or by selecting *Start Debugging* from the *Debug* menu (they are all the same thing), your code is deployed to the hardware with debugging enabled, and then your code is executed.

PRINTING DEBUG MESSAGES

.NET Gadgeteer projects have a line that prints `Program Started` to the debug window. This shows that your program has deployed and successfully started. You can add your own debug print statements to show the progress of your code, for example to see if a button has been pressed. In the Clicker project example we can write the count value to the debug window every time the button is pressed by simply adding a single print line as shown in Figure 83.

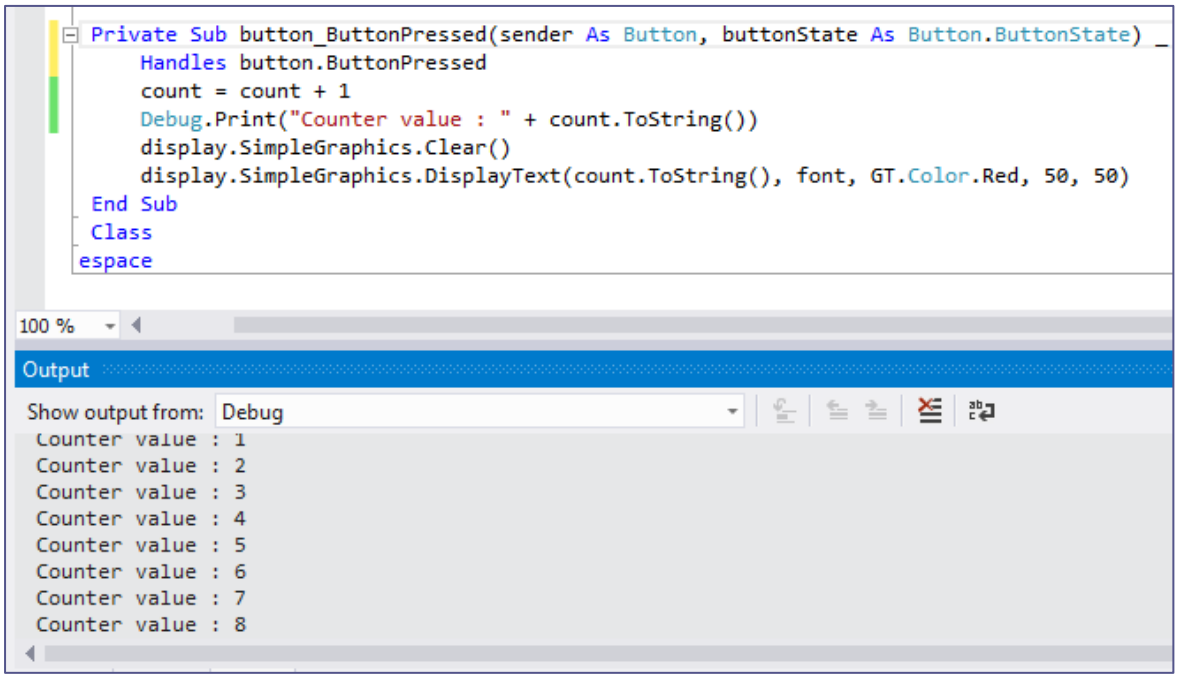


Figure 83 : Writing text and variables to the Debug window

SETTING A BREAKPOINT

A breakpoint is a way of pausing the code running on the hardware when a particular line of code is about to be executed. Once the execution has been paused you can inspect the variables to check their values. For example we can set breakpoint on the line of code that increments the counter. This can be done by placing the cursor on the line of code where you want to pause execution and selecting *Toggle Breakpoint* from the *Debug* menu (Or pressing F9).

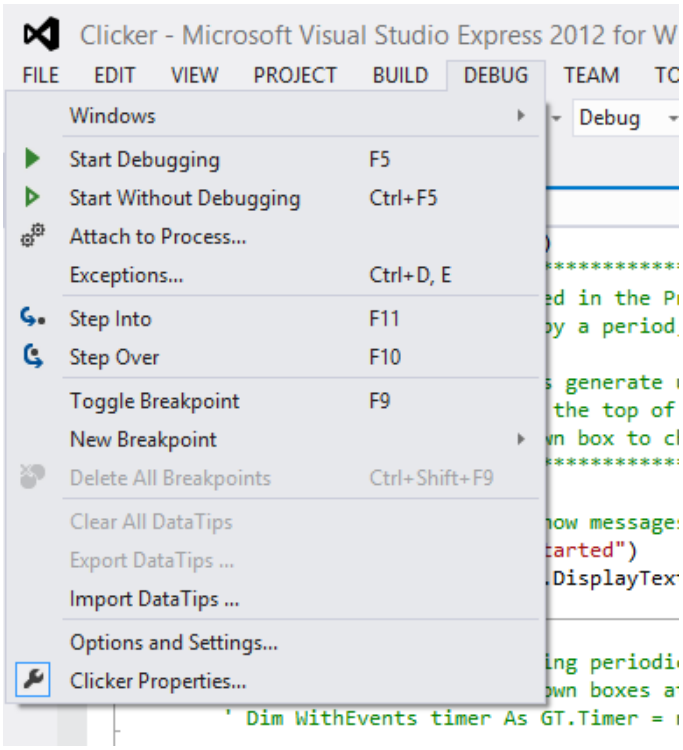


Figure 84 : Use Toggle Breakpoint to enable or disable breakpoints. Alternatively press the F9 key

You will see that a red dot appears in the left hand side margin next to the line that will pause execution, as shown in Figure 85. Note that it is not always sensible to have a breakpoint, for example on a blank line or on a line between sub-routines. If you attempt to set a breakpoint where it is not feasible, none will appear and the status bar at the bottom of the screen in Visual Studio will display the message "*This is not a valid location for a breakpoint*". You can also enable and disable breakpoints by simply clicking in the margin where the red dot has appeared.

```
Private Sub button_ButtonPressed(sender As Button, buttonState As Button.ButtonState)
    Handles button.ButtonPressed
    count = count + 1
    display.SimpleGraphics.Clear()
    display.SimpleGraphics.DisplayText(count.ToString(), font, GT.Color.Red, 50, 50)
End Sub
```

Figure 85 : Setting a breakpoint on the counter increment line

The line of code that will cause execution to pause will also turn red. Since you can have any number of breakpoints you may also want to show a list of breakpoints. This will help with enabling or disabling breakpoints in bulk. To show the breakpoint list, select DEBUG → Windows → Breakpoints or press <Alt>+<F9>. You can even add or change breakpoints once your code is being executed. Figure 86 shows the single breakpoint that has been set in the code shown above.

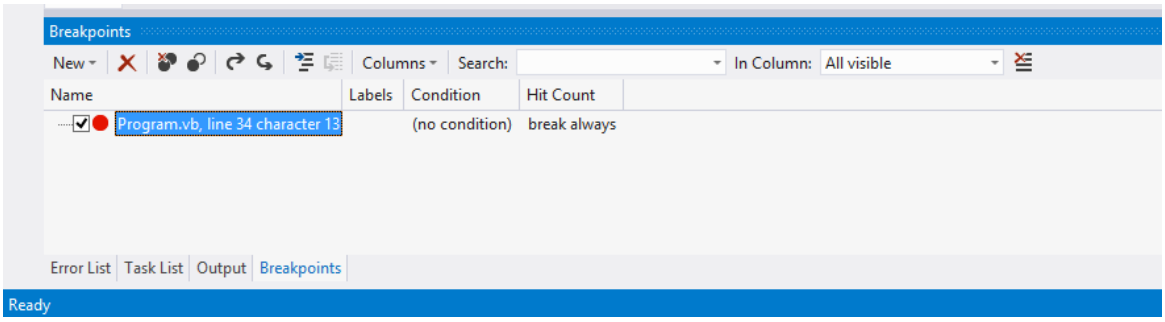


Figure 86 : Breakpoint window listing all breakpoints. You can multi-select breakpoints and enable/disable them in bulk if required. This is useful if you want to simply run your code wihtout pausing.

Once you have set the breakpoint press the play button as normal, execution will only pause when a breakpoint gets hit. In our example case we need to press the hardware button in order for the routine to be executed, thus triggering the breakpoint. You will know when a breakpoint has been reached because Visual Studio will automatically highlight the breakpoint in yellow and bring the code into view. The code has now paused on the mainboard, so the screen will not update with the latest value.

If you hold your mouse over any variable that is in scope (for example variables in the current sub-routine, and global variables) a little window will pop up showing the actual value that is stored in memory. If the variable was more complicated such as an array, you would be able to see all the values stored in the array.

```
Private Sub button_ButtonPressed(sender As Button, buttonState As Button.ButtonState) Handles button.ButtonPressed
    count = count + 1
    display.Simp count 0 lear()
    display.SimpleGraphics.DisplayText(count.ToString(), font, GT.Color.Red, 50, 50)
End Sub
```

Figure 87 : Visual Studio highlights the current active breakpoint in yellow. You can then inspect variables by holding the mouse over to see their contents.

It is important to note that the execution is paused before the line with the break point is executed. It is for this reason that the `count` variable is still set to 0. Once you have hit a breakpoint see page 110 to see how you can step through your code one line at a time. Alternatively select *Continue* from the Debug menu (<F5>) to carry on running your program.

THE CONDITIONAL BREAKPOINT

When you set a breakpoint, execution will pause every time execution reaches that line of code. This can be inconvenient, for example, if you have a long loop that iterates say 100 times, and you know something fails on the last iteration of the loop. You could set a breakpoint at the beginning of the loop and press continue 99 times until you get to the last iteration. There is an easier way: the conditional breakpoint. This will only pause at a particular line of code if a condition is met or if the value has changed, for example when the variable counter is equal to 99.

In the Clicker example we can set the condition on the counter breakpoint by right-clicking and setting the condition to only pause when the counter is equal to 5, as show in Figure 88 and Figure 89.

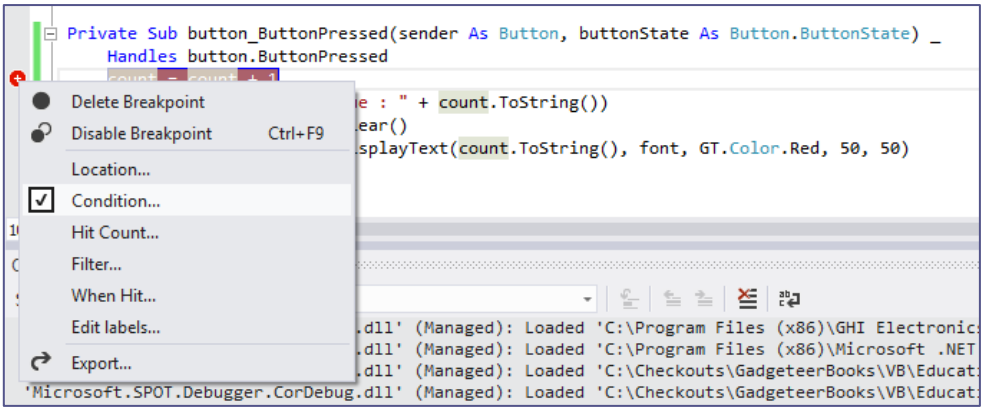


Figure 88 : To create a conditional breakpoint, right click a breakpoint and select condition

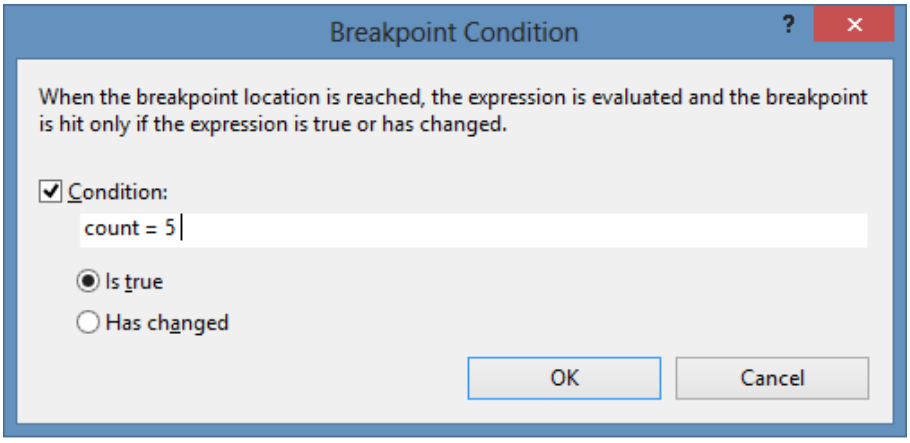


Figure 89 : Set the condition that when true will pause execution. Keep expressions simple here to avoid errors.

There are a few other conditional break capabilities that are handy (see Figure 88).

Another useful feature that could be used in this example is the hit count option shown in Figure 88. Every time a breakpoint is hit this counter is incremented. You can set it to only pause when it has been hit a certain number of times, a multiple of a certain number, or greater than a certain number. In this example we could use it to pause when the hit count is equal to 5.

WALKING THROUGH YOUR CODE

Once you have paused on a breakpoint you will notice that there are now a few more options on the debug menu (see Figure 90). These options let you choose how to resume execution of your program, for example by letting you step line by line through your code. Remember you can hold your mouse over any in-scope variables to see their contents. It can be helpful to remember the shortcut keys for these different step options, particularly the Step Over (F10) option, but the options are also shown as the following icons on the toolbar at the top of Visual Studio.

Figure 90 shows a brief description of the various options:

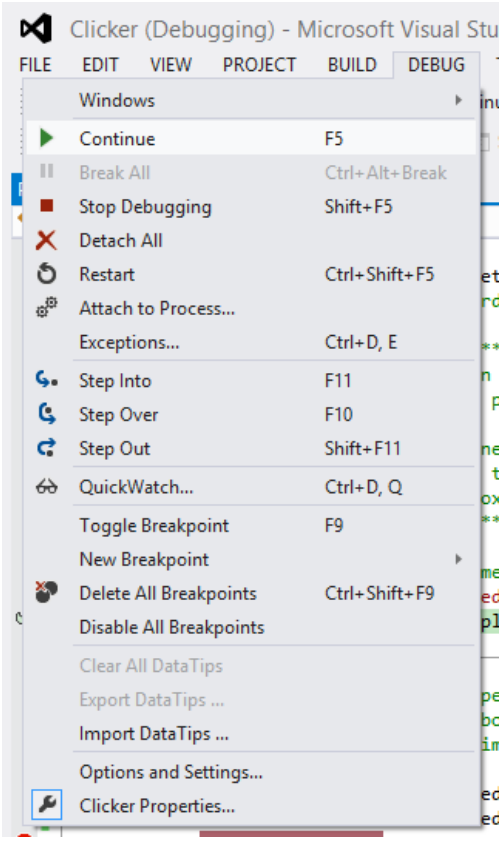


Figure 90 : The Debug menu showing the options once a breakpoint has been hit. There are various step options to step through your code.

Continue (F5)

Once you have paused on a breakpoint and have finished inspecting the variables, you can simply press Continue or <F5> and your program will continue running. It will only pause again if it hits another breakpoint.

In the example shown in Figure 87, inserting a second breakpoint on the last line of the sub-routine and deploying your project will cause it to pause on the first breakpoint. Pressing Continue or <F5> will then advance execution to the second breakpoint as shown in Figure 91. The counter will be incremented and the screen cleared but the new text will not yet be displayed as the current line has not been executed.

```
Private Sub button_ButtonPressed(sender As Button, buttonState As Button.ButtonState) Handles button.ButtonPressed
    count = count + 1
    display.SimpleGraphics.Clear()
    display.SimpleGraphics.DisplayText(count.ToString(), font, GT.Color.Red, 50, 50)
End Sub
```

Figure 91: Pressing Continue will advance to the next breakpoint

Step Over (F10)

This is probably the most useful option and will let you step to the next line of code executing the line that was shown in yellow. You can then hold your mouse over any variable to see how it has changed. In the example shown in Figure 87 when you step over the current line, it is executed and the next line of code is displayed in yellow. You can then hold the mouse over the counter variable to see that it has been incremented from 0 to 1, as shown in Figure 92.

Note that execution has paused on the clear display line of code. The display will not yet have cleared as the current line (shown in yellow) has not been executed. Stepping over this line will execute this line and you will see

the display clear and stepping over yet again will then put the text on the display. You will then probably want to press Continue or <F5> to resume your program as normal.

If you press the hardware button again, it will halt on the breakpoint again and you can step through the next increment of the counter.

```
Private Sub button_ButtonPressed(sender As Button, buttonState As Button.ButtonState)
    count = count + 1
    dis count 1 raphics.Clear()
    display.SimpleGraphics.DisplayText(count.ToString(), font, GT.Color.Red, 50, 50)
End Sub
```

Figure 92 : Using the Step Over option will advance execution to the next line of code. In this case we can see that the counter has been incremented.

Step Into (<F11>) and Step Out (<SHIFT>+<F11>)

These two options are used for getting into and out of sub-routines. If execution has been paused on a line of code that calls a sub-routine (for example other code that you have written) then you can use the Step Into option to view the sub-routine and then step line by line. If you had used the Step Over option then the sub-routine would be executed in a single step and you would not be able to step through each line of code in the sub-routine.

DEBUGGING TIPS

- Remember that your code is paused when it hits a breakpoint. You may reach one, not realise and then wonder why nothing is working. An easy way to tell is to look at the Visual Studio title bar at the top of the screen. It will tell you if it is running or debugging, for example see Figure 93.

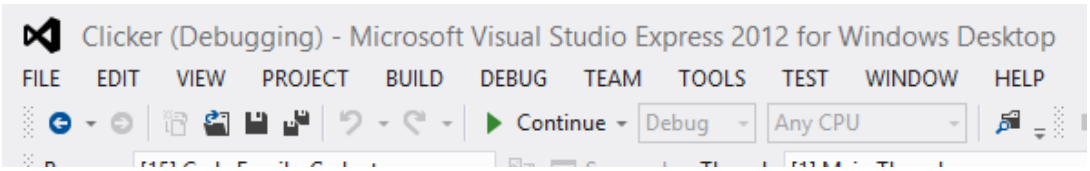


Figure 93 : The title of the Visual Studio window shows the solution name as well as the current status, Running or Debugging if a breakpoint has been reached.

- You may find that you end up with many breakpoints as you debug issues. It can be useful to disable a breakpoint rather than delete it, as you may want to use it later on, for example in a loop. Use the debug view window to list all breakpoints and then enable/disable them in bulk.
- Always keep an eye on the status bar at the bottom of the screen, while you may be waiting for the deployment and the first breakpoint to be hit, Visual Studio may be telling you something important. For example, it may indicate a failed deployment (Figure 94).

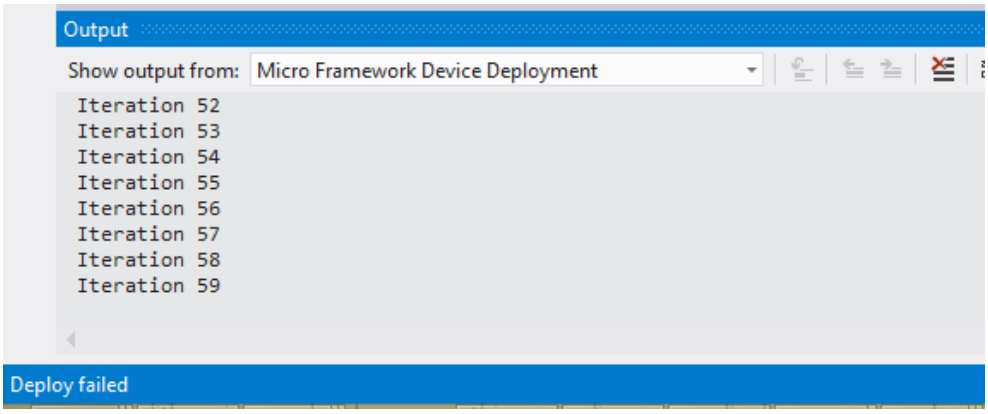


Figure 94 : Visual Studio status bar showing that the deployment failed

APPENDIX F. COPING WITH OUT-OF-MEMORY EXCEPTIONS

The Cerberus mainboard (used in this book) is a low-cost, low-power processor making it great for a wide range of applications. However it has limitations. It cannot do everything at the same time and certainly will not compete with your latest laptop. For example, reading files from an SD card and displaying graphic at the same time can stretch the limits of the hardware. You notice this when you deploy (F5) your code and it just hangs (try it a few times to be sure there is no other issue) displaying `Failed allocation for xxx  blocks, xxxx bytes` in the Output window, more than once.

If this is the case, try reducing the amount of tasks you have asked the hardware to perform. It is also easier to start with a simple project, check it works and then extend it slowly, testing that each stage works.

You may also get a memory exception window as shown in Figure 95. If your program ran once and then you get a memory error, it is likely that it is very close to the maximum limit. Try cleaning the project (Build -> Clean Solution) then redeploy your program (F5).

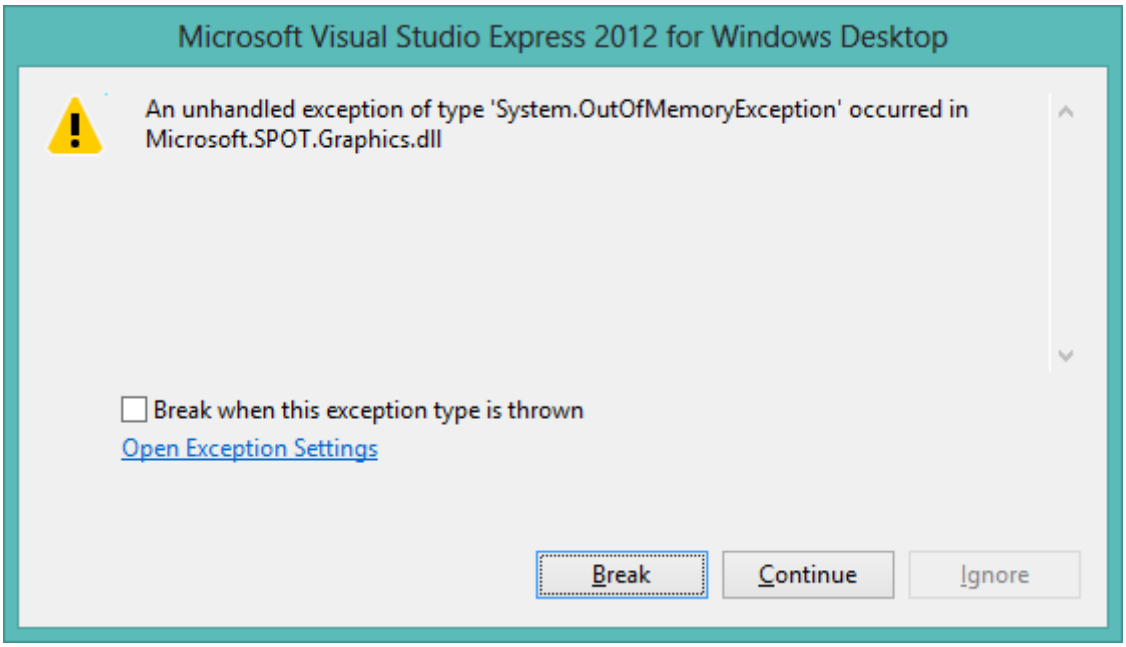


Figure 95: Visual Studio Out Of Memory exception

APPENDIX G. TROUBLESHOOTING: VISUAL STUDIO WILL NOT DEPLOY

When Visual Studio complains that there are deployment issues yet everything is plugged in, there are a few things you need to check.

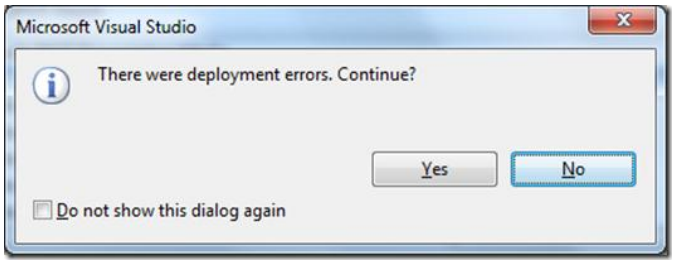


Figure 96 : Visual Studio deployment error

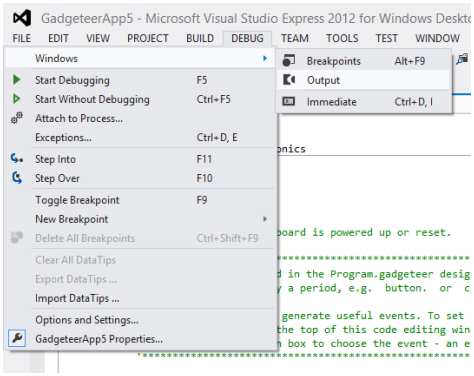


Figure 97 : To view the Output window in Visual Studio Express 2012 select Debug → Windows → Output

The Output window will show you errors and is usually located at the bottom of the main Visual Studio window (you may need to select the tab). Below is the output of a blank project once it has been successfully rebuilt.

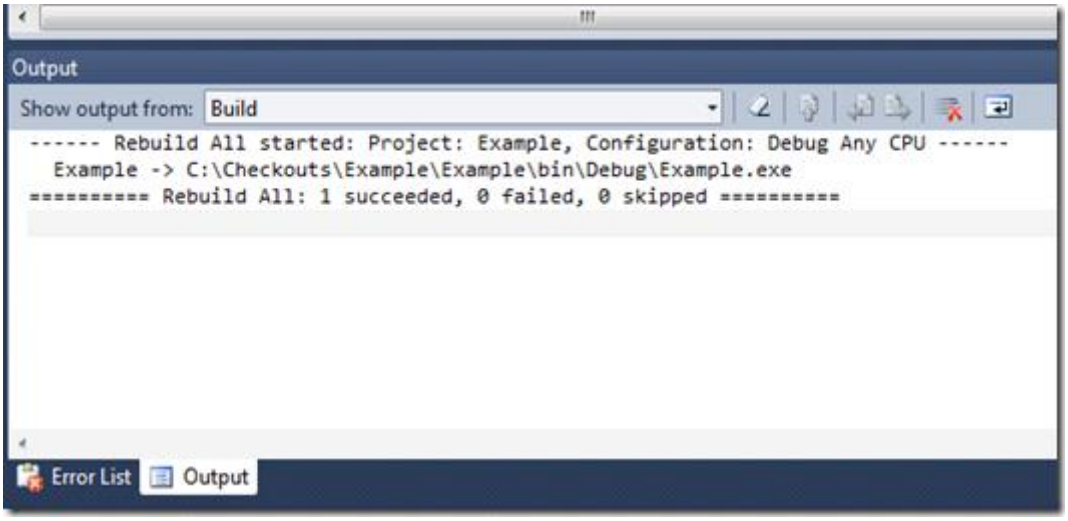


Figure 98 : Blank .NET Gadgeteer project, successful build and deploy

Before you go any further it is worth doing a few sanity checks:

- Did the project build without errors? Have a look at the output window to see if there are errors. You can always open a new project in Visual Studio and quickly try a blank Gadgeteer project. The default template for a new project will build and deploy without errors.
- Check that the hardware is plugged in properly, try another USB port on the computer or swap the USB cable to check it is not faulty.
- Don't plug into any USB3.0 ports on your computer if possible. There can be compatibility issues with these. They tend to have blue plastic showing and/or be labelled with the text 'USB 3.0'.
- Does hitting the reset switch on the mainboard help? Wait about 20 seconds before trying to deploy from Visual Studio.

If you are satisfied that everything should work or it was just working but is not anymore, the first step is to check that your computer has found the mainboard. Plug the mainboard into a USB port, go to Device manager and see if the device has been found. To load Device Manager click Start, then in the Run box type the following command `mmc compmgmt.msc` or select Device Manager from the *hardware and sound* section of the control panel as shown in Figure 99.

Figure 99 : Locating the Device Manager in the Control Panel

Have a look in the Device Manager for the .NET Gadgeteer mainboard. Not all mainboards appear in the same section in the Device Manager. The FEZ Cerberus is located under "Universal Serial Bus controllers" and is called "GHI NETMF Debugging Interface" as shown in Figure 100. Note that future firmware updates may rename or move this in the Device Manager. You can easily check if you have the correct device on the list by unplugging the USB cable. It should disappear when you remove it and reappear when you reconnect the USB cable.

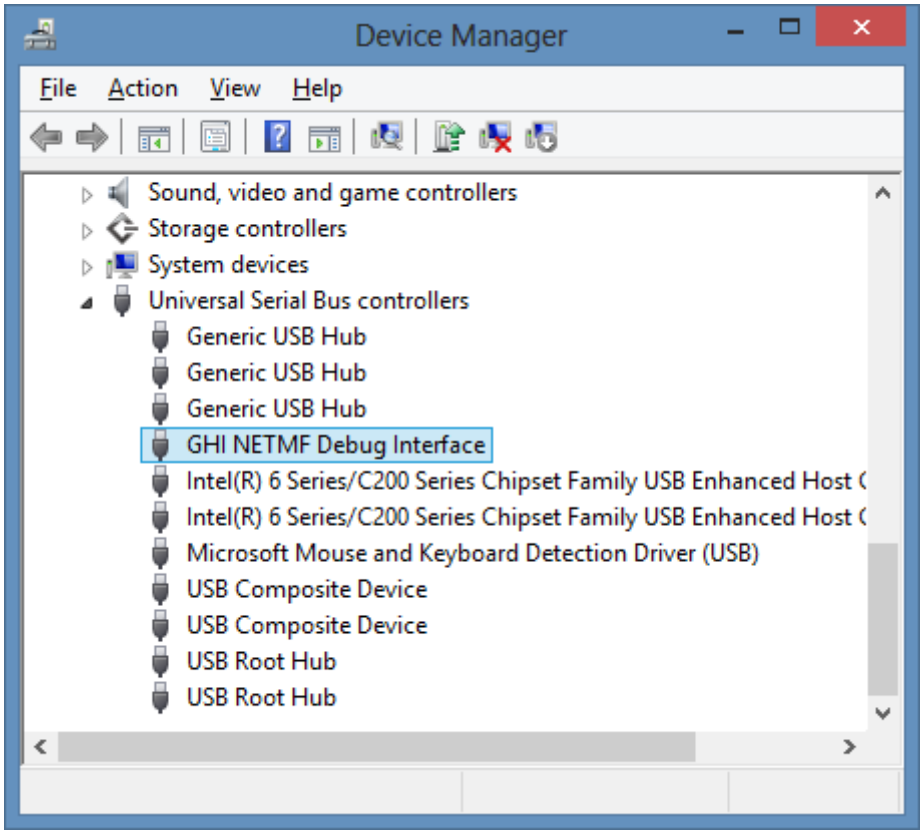


Figure 100: The FEZ Cerberus mainboard shown in Device Manager

If you can see your hardware listed in the Device Manager and there are no warnings then you can move on and assume that you hardware has been recognised correctly by your computer. If you get a yellow exclamation mark on the mainboard in Device Manager then the driver did not load and the hardware will not work until you get the driver fixed. Simply try a different USB port on the computer, particularly if it worked before. Failing that it is best to reinstall the SDK provided by the manufacturer for the kit you are using.

If you got a deployment issue and your hardware is displayed in the Device Manager, the next step is to check that Visual Studio can see the hardware and is attempting to deploy to the correct device. In the Solution Explorer right click on the project (not solution) and select Properties. To open the Solution Explorer if it is not on the left of the screen, select it from the Visual Studio menu, VIEW → Other Windows → Solution Explorer.

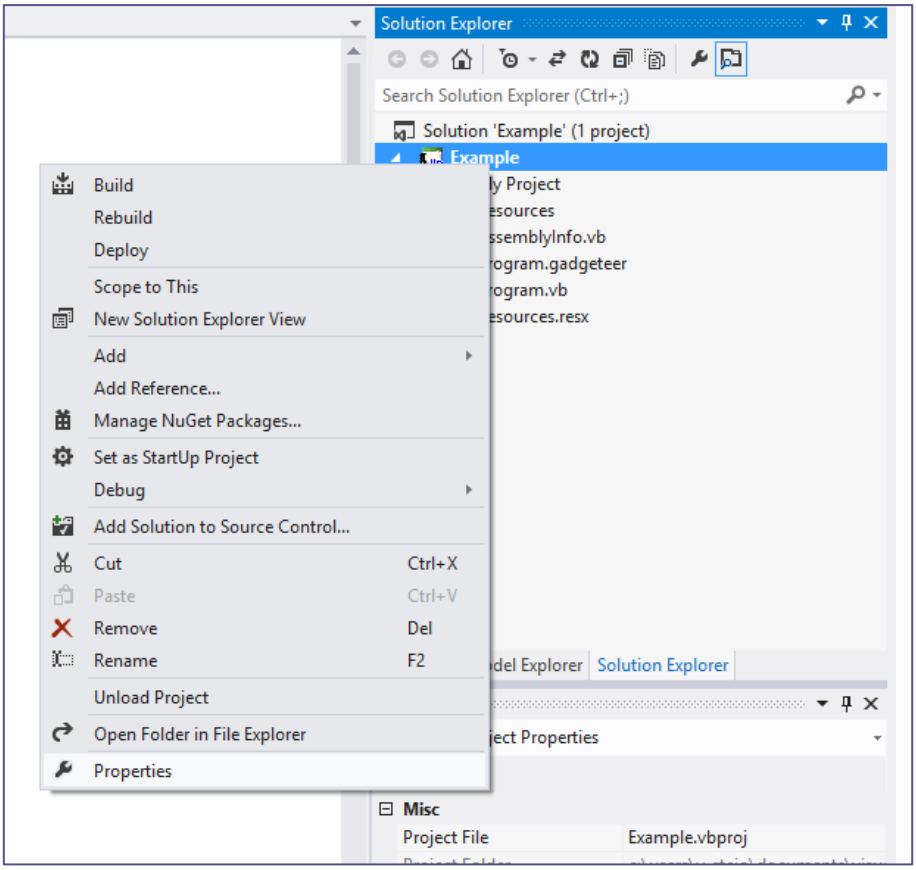


Figure 101 : Selecting the project Properties in Visual Studio

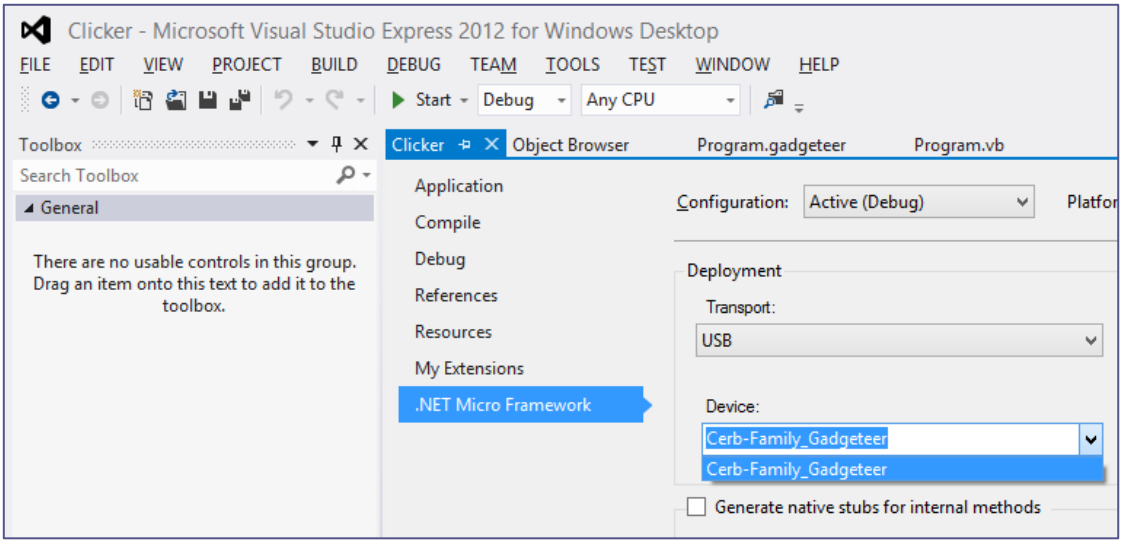


Figure 102: FEZ Cerberus mainboard recognised in Visual Studio

The properties window has a .NET Micro Framework tab which will show you which device Visual Studio is attempting to deploy. The transport should be set to USB (if you are using a USB cable to connect the mainboard to the computer) and you should see the device listed. If your device is not listed here then it was not recognised by your computer. Check to be sure it is listed in the Device Manager. If you change this setting you need to save for the change to take effect.

If your hardware is recognised and Visual Studio can see the device, you should not get a deployment error. Be sure your project compiles (test with a blank new Gadgeteer project). Check the Output window for errors and check to see if it is a firmware version issue.

APPENDIX H. LIST OF CURRENT SOCKET TYPES

A	Three analogue inputs, with pins number 3 and 4 doubling as general-purpose input/output. In addition, pin number 6 is a general-purpose input/output, and pin number 3 supports interrupt capabilities.
B	Display (LCD) interface, carrying the blue component on pins 3 to 7, as well as the LCD enable line on pin 8 and the clock signal on pin 9.
C	Controller-area network (CAN, or CAN-Bus). Pins number 4 and 5 serve as the CAN transmit (TD) and receive (RD) pins, and double as general-purpose input/outputs. In addition, pins number 3 and 6 are general-purpose input/outputs, and pin number 3 supports interrupt capabilities.
D	A USB device interface for the mainboard to connect to a PC, usually for programming. Pins 4 and 5 are used as the dedicated USB data pins (D- and D+). In addition, pins 3, 6 and 7 are general-purpose input/outputs, with pin 3 supporting interrupt capabilities.
E	Ethernet PHY connection. Pins 6 to 9 are the dedicated transmit/receive lines to an Ethernet connector with integrated magnetics. Pins 4 and 5 are optional connections to the LEDs on the Ethernet connector.
F	Secure Digital Card (SD) or MMC (Multi Media Card) interface. Pins 4 to 9 are the dedicated data and control lines for this interface. In addition, pin 3 is a general-purpose digital input/output with interrupt capabilities.
G	Display (LCD) interface, carrying the green component on pins 3 to 8, as well as the backlight control line on pin 9.
H	USB host interface to connect USB peripherals to the mainboard. Pins 4 and 5 are used as the dedicated USB data pins (D- and D+). In addition, pin 3 is a general-purpose input/output with interrupt capabilities.
I	I2C interface. Pins 8 and 9 are the dedicated I2C data (SDA) and clock (SCL) lines. Note that a mainboard should include pull-up resistors for these pins, in the region of 2.2K Ohms. Modules must not include their own pull-ups on these lines. In addition, pins 3 and 6 are general-purpose input/outputs, with pin 3 supporting interrupt capabilities.
K	UART (serial line) interface operating at TTL levels, with hardware flow control capabilities. Pin 4 (TX) is data from the mainboard to the module, and pin 5 (RX) is data from the module to the mainboard. These lines are idle high (3.3V), and can double as general-purpose input/outputs. Pin 6 (RTS) is an output from the mainboard to the module, indicating that the module may send data. Pin 7 (CTS) is an output from the module to the mainboard indicating that the mainboard may send data. The RTS/CTS are 'not ready' if high (3.3V) and 'ready' if low (0V). In addition, pin 3 is a general-purpose input/output, supporting interrupt capabilities.
O	Analog output on pin 5. In addition, pins 3 and 4 are general-purpose input/outputs, and pin 3 includes interrupt capabilities.
P	Three pulse-with modulated (PWM) outputs on pins 7, 8 and 9. Pins 7 and 9 double as GPIOs. In addition, pin 3 is an interrupt-capable GPIO, and pin 6 is a GPIO.

R	Display (LCD) interface, carrying the red component on pins 3 to 7, as well as the VSYNC line on pin 8 and the HSYNC line on pin 9.
S	Serial peripheral interface (SPI). Pin 7 is the master-out/slave-in (MOSI) line, pin 8 is the master-in/slave-out (MISO) line, and pin 9 is the clock (SCK) line. In addition, pins 3, 4 and 5 are general-purpose input/outputs, with pin 3 supporting interrupt capabilities.
T	Four-wire touch screen interface.
U	UART (serial line) interface operating at TTL levels. Pin 4 (TX) is data from the mainboard to the module, and pin 5 (RX) is data from the module to the mainboard. These lines are idle high (3.3V), and can double as general-purpose input/outputs. In addition, pins 3 and 6 are general-purpose input/outputs, with pin 3 supporting interrupt capabilities.
X	Three general-purpose input/output (GPIO) pins, with pin number 3 supporting interrupt capabilities.
Y	Seven general-purpose input/output (GPIO) pins, with pin number 3 supporting interrupt capabilities.
Z	Manufacturer specific. The pinout for this socket will vary between mainboards. Please refer to the individual mainboard's documentation for details.
*(DaisyLink)	DaisyLink downstream interface. Pin 3 is used for the DaisyLink neighbour bus, pin 4 is used for I2C SDA, pin 5 is used for I2C SCL. Note that this socket type should not appear on a mainboard, only on DaisyLink modules. The [MS] pins on this socket type can optionally support reflashing the firmware on the module.

Socket Type	Pin 1	Pin 2	Pin 3	Pin 4	Pin 5	Pin 6	Pin 7	Pin 8	Pin 9	Pin 10	Socket Letter
3 GPIO	+3.3V	+5V	GPIO!	GPIO	GPIO	[NC]	[NC]	[NC]	[NC]	GND	X
7 GPIO	+3.3V	+5V	GPIO!	GPIO	GPIO	GPIO	GPIO	GPIO	GPIO	GND	Y
Analog In	+3.3V	+5V	AIN (G!)	AIN (G)	AIN	GPIO	[NC]	[NC]	[NC]	GND	A
CAN	+3.3V	+5V	GPIO!	TD (G)	RD (G)	GPIO	[NC]	[NC]	[NC]	GND	C
USB Device	+3.3V	+5V	GPIO!	D-	D+	GPIO	GPIO	[NC]	[NC]	GND	D
Ethernet	+3.3V	+5V	[NC]	LED1 (OPT)	LED2 (OPT)	TX D-	TX D+	RX D-	RX D+	GND	E
SD Card	+3.3V	+5V	GPIO!	DAT0	DAT1	CMD	DAT2	DAT3	CLK	GND	F
USB Host	+3.3V	+5V	GPIO!	D-	D+	[NC]	[NC]	[NC]	[NC]	GND	H
I²C	+3.3V	+5V	GPIO!	[NC]	[NC]	GPIO	[NC]	SDA	SCL	GND	I
UART+Handshaking	+3.3V	+5V	GPIO!	TX (G)	RX (G)	RTS	CTS	[NC]	[NC]	GND	K
Analog Out	+3.3V	+5V	GPIO!	GPIO	AOUT	[NC]	[NC]	[NC]	[NC]	GND	O
PWM	+3.3V	+5V	GPIO!	[NC]	[NC]	GPIO	PWM (G)	PWM (G)	PWM	GND	P
SPI	+3.3V	+5V	GPIO!	GPIO	GPIO	GPIO	MOSI	MISO	SCK	GND	S
Touch	+3.3V	+5V	[NC]	YU	XL	YD	XR	[NC]	[NC]	GND	T
UART	+3.3V	+5V	GPIO!	TX (G)	RX (G)	GPIO	[NC]	[NC]	[NC]	GND	U
LCD 1	+3.3V	+5V	LCD R0	LCD R1	LCD R2	LCD R3	LCD R4	LCD VSYNC	LCD HSYNC	GND	R
LCD 2	+3.3V	+5V	LCD G0	LCD G1	LCD G2	LCD G3	LCD G4	LCD G5	BACKLIGHT	GND	G
LCD 3	+3.3V	+5V	LCD B0	LCD B1	LCD B2	LCD B3	LCD B4	LCD EN	LCD CLK	GND	B
Manufacturer Specific	+3.3V	+5V	[MS]	[MS]	[MS]	[MS]	[MS]	[MS]	[MS]	GND	Z
DaisyLink Downstream*	+3.3V	+5V	GPIO!	GPIO	GPIO	[MS]	[MS]	[MS]	[MS]	GND	*

APPENDIX I. OPERATORS IN VISUAL BASIC

Operator	Description
=	Is equal to
<=	Is less than or equal to
>=	Is greater than or equal to
<>	Is not equal to
>	Is greater than
<	Is less than
And	Performs a logical conjunction on two Boolean expressions
Not	Performs logical negation on a Boolean expression
Or	Performs a logical disjunction on two Boolean expressions
Xor	Performs a logical exclusion on two Boolean expressions
AndAlso	Performs short-circuiting logical conjunction on two expressions
OrElse	Performs short-circuiting inclusive logical disjunction on two expressions

APPENDIX J. TYPICAL LUX VALUES

Lux (lx) is an international standard unit for ambient light and is often used in light sensors. It represents the amount of light incident per unit area – it's actually calculated as lumens per square metre. Below is a table of typical Lux ranges for different scenarios.

Table 22: Typical Lux values

Lighting condition	Typical Lux (lx) values
Pitch black	0 – 10 lx
Very dark	11 – 50 lx
Dark indoors	51 – 200 lx
Dim indoors	201 – 400 lx
Normal indoors	401 – 1000 lx
Bright indoors	1001 – 5000 lx
Dim outdoors	5001 – 10,000 lx
Cloudy outdoors	10,001 – 30,000 lx
Direct sunlight	30,001 – 100,000 lx

APPENDIX K. RESOURCES AVAILABLE ONLINE

- Main Microsoft site for .NET Gadgeteer: http://www.netmf.com/gadgeteer
- Resources to learn about .NET Gadgeteer: http://gadgeteering.net
- Source code: http://gadgeteer.codeplex.com
- Consider searching http://www.codeplex.com for either 'Gadgeteer' or 'NETMF' to discover alternative drivers for modules or helpful libraries. For example, StringBuilder: http://netmfcommonext.codeplex.com
- Books using other languages
    - *Microsoft .NET Gadgeteer : Electronics Projects for Hobbyists and Inventors*
    - *Getting Started with .NET Gadgeteer*