WITH BASIC FOR ANDROID – B4A

ANDROID APP DEVELOPMENT

FOR ELECTRONICS DESIGNERS

Dogan Ibrahim

elektor

LEARN DESIGN SHARE

Android App Development for Electronics Designers

Dogan Ibrahim

elektor

LEARN DESIGN SHARE

● This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
78 York Street, London W1H 1DP, UK
Phone: (+44) (0)20 7692 8344

● Declaration

The author and publisher have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, or hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident or any other cause.

● British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

● **ISBN 978-1-907920-71-4**

Prepress Production: D-Vision, Julian van den Berg
First published in the United Kingdom 2018
Printed in the Netherlands by Wilco

This book is about developing apps for the Android mobile devices (mobile phones and tablets) using the **Basic For Android (B4A)** programming language and the Integrated Development Environment (B4A IDE). The book includes many tested and working projects, where most hardware based projects have the following sub-headings:

- Title of the project
- Description of the project
- Aim of the project
- Block diagram
- Circuit diagram
- Construction
- Complete program listing of the project
- Full description of the program

The book is aimed for students, hobbyists, and anyone else interested in developing apps for the Android mobile devices. First parts of the book describe how to install the B4A on your PC. The syntax and the programming features of the B4A are then described step-by-step by giving simple projects.

One of the nice features of this book is that it describes with many projects and step-by-step how an Android mobile device can communicate with a Raspberry Pi, or Arduino, or the ESP32 processor over a Wi-Fi link or by using SMS text messages.

The example projects describe how an Android mobile device can send commands using the UDP protocol or SMS messages in order to control devices connected to a Raspberry Pi, Arduino, or to the ESP32 processor. Additionally, some of the projects show how data packets can be sent from a Raspberry Pi, Arduino, or from the ESP32 processor to an Android mobile phone and then displayed on the mobile phone.

All of the Android side of the projects given in the book have been developed using the B4A programming language. The Raspberry Pi projects have been developed using the Python language. Arduino and the ESP32 processor projects make use of the popular Arduino IDE. Full program listings of all the projects as well as the detailed program descriptions are given in the book. Users should be able to use the projects as they are presented, or modify them to suit to their own needs.

• 12

Preface

Worldwide statistics show that the number of smartphones sold to end users in the last decade from 2007 to 2017 has been increasing steadily. In 2016, around 1.5 billion smartphones were sold worldwide to end users. In 2017 this number increased to around 1.54 billion, which is a significant increase over just one year. In the fourth quarter of 2016, 81.7% of all smartphones sold to end users were phones with the Android operating system. This number has increased to 85.9% in the first quarter of 2018 (source: https://www.statista.com).

Developing apps for mobile phones is not an easy task and requires extensive knowledge of programming skills. The program development also takes considerable amount of time. Android based apps are available in the Google Play Store. Most of these apps are free of charge and can easily be downloaded to your mobile device. The problem with most of these apps is that they are not tested by any authority and therefore are available as you see them. In addition, most of such applications include advertisements which can make it annoying for the users. It is however possible to purchase some more professional apps without any built-in advertisements.

This book is about developing apps for the Android mobile phones or tablets using the **Basic For Android (B4A**) language and the integrated development environment. B4A syntax and the development environment are similar to the Visual Basic program development environment, and thus it is very easy to develop apps very quickly and without much knowledge of previous programming languages.

B4A includes all the features of the Android mobile devices and is therefore an ideal platform for developing apps for the Android mobile phones. Features such as Wi-FI, SMS, NFC, graphics, serial ports, phone utilities, and many more features are supported by B4A. Thousands of developers around the world, including famous companies such as HP, IBM, and NASA use the B4A.

Unfortunately not many books are available in the market place on B4A. One of the most comprehensive books that the author came across is by *Wyken Seagrave* and is entitled *B4A Rapid Android App Development Using BASIC*.

This book gives an introduction to the syntax of B4A, and gives example projects showing how to develop simple apps using the B4A. One of the interesting and strong points of this book is that it describes how an Android mobile phone can communicate with the Raspberry Pi, Arduino, or the ESP32 based development boards remotely, using Wi-Fi and SMS text messaging. Full program listings of the B4A, Raspberry Pi, Arduino, and the ESP32 processors are given in the book with detailed hardware and software descriptions of each project. I hope you like reading the book and find it useful for your next Android based apps project.

Prof Dr Dogan Ibrahim
London, 2018

Chapter 1 • Android mobile phones

1.1 Overview

In this Chapter we shall be looking at the history of the mobile phones very briefly and also learn about the different versions of the Android operating systems.

1.2 Mobile Phone Generations

Currently, we can investigate the mobile phone development in 4 generations.

1.2.1 First Generation (1G)

The first 1G mobile phone was commercially available analogue cellular system, initially developed and launched by Motorola Inc in 1983, and was given the name DynaTAC 800X. These phones were very large in size (see Figure 1.1), had only 35 minutes of talk time, required long charging times, and the phone could store up to 30 numbers for quick recall. The phone was 33cm high, and weighted about 0.8Kg. Despite these limitations there was very high demand by the consumers. This phone was priced around $4,000 in 1984 (equivalent to $9,500 in 2017).

Figure 1.1 Early DynaTAC 800x mobile phone

1.2.2 Second Generation (2G)

In the second generation, phone conversations were digitally encrypted. These systems were much more efficient than the early systems, allowing greater penetration levels to be achieved. These phones were mainly developed for voice, although 2G also has introduced the very important concept of data services, such as text based SMS messaging and early picture messaging (MMS), and multimedia services (MMS). Nokia mobile phones were probably the most popular phones used during the second generation (see Figure 1.2). These phones had small sizes (115 x 51 x 12 mm), low weight (79g), small gray-scale screens, large memories for storing contact details, and single-core CPUs. These mobile phones dominated the consumer market in the early and late 1990s.

Figure 1.2 Second generation Nokia mobile phone

1.2.3 Third generation (3G)

3G technology was the result of research and development work carried out by the International Telecommunication Union in the early 1980s. Third generation mobile phones became available in early 2000s, where the first 3G phone was developed in Japan. In the third generation, the data speed increased from 144kbps to 2Mbps, and also packet switching Internet services and video communications have been available on these phones. These phones were also called Smartphones. We can see many manufacturers such as Samsung, Apple, HTC, Alcatel, Nokia, Blackberry and others offering 3G phones. By the year 2007 over 200 million 3G subscribers were available. 3G offered additional services such as GPS, mobile TV, video conferencing, telemedicine etc.

1.2.4 Fourth Generation (4G)

4G technology has become available around 2010 and it offered true mobile broadband, higher speeds (100Mbps to 1Gbps), higher security, and higher capacity.

1.2.5 Fifth Generation (5G)

5G mobile phones are currently under development. It is expected that these phones will have higher data rates than 4G, support for many simultaneous connections to be made, better coverage and signalling efficiency, and more energy efficiency.

1.3 Android Versions

The topic of this book is the development of apps for the Android mobile phones. It is therefore worthwhile to look at briefly the history of the Android operating system and also to the mobile phones that use this operating system.

The first commercial Android operating system (version 1) was first introduced in 2008. Since then the operating system has been developed by Google and the Open Handset Alliance and many updated versions have been released. Android version names have been chosen to be confectionary names (except the first two releases) and also they have been introduced in alphabetical order. Table 1.1 gives a list of the various Android version names and version numbers together with their initial release dates. Interested readers can find

detailed information on the Internet about each version. Note that the early versions up to and including Lollipop are not currently supported. Also, although not announced yet, a new version (version 9.0) called Android P is currently under development.

Android Code Name	Version Number	Release Date
Petit Four	1.1	2009
Cup Cake	1.5	2009
Donut	1.6	2009
Eclair	2.0 – 2.1	2009
Froyo	2.2	2010
Gingerbread	2.3	2010
Honeycomb	3.0 – 3.2	2011
Ice Cream Sandwich	4.0	2011
Jelly Bean	4.1 – 4.3	2012
Kit Kat	4.4	2013
Lollipop	5.0 – 5.1	2014
Marshmallow	6.0	2015
Nougat	7.0 – 7.1	2016
Oreo	8.0 – 8.1	2017

Table 1.1 Android operating system versions

1.4 Smartphones Using the Android Operating System

At the time of writing this book there were several companies offering Smartphones based on the Android operating system. Some details about the popular ones are given below. Interested readers can get full details of these smartphones from the manufacturers' web sites:

Samsung

Probably the most popular Android smartphones are offered by Samsung. Samsung is continually developing new smartphones with added features. Currently, some of the popular smartphones available from Samsung are: Galaxy S9, Galaxy S9 Plus, Galaxy S8, and Galaxy Note 8.

Huawei

Huawei offers Android based smartphones at lower cost and they are one of the main competitors of Samsung. Currently, some of the popular smartphones available from Huawei are: P20, P20 Pro, Mate 10 Pro, Nova 2i, and P Smart.

LG

LG also offers cheaper Android based smartphones than Samsung. Currently, some of the smartphones offered by LG are: G7 ThinQ, V30, Q6, G6, and Stylus 3.

OnePlus 6

OnePlus 6 offers Android smartphones at budget prices. These are fast, low-cost (below £500) smartphones with large displays (6.28 inches). The phone is equipped with two cameras with 16 and 20 megapixel resolutions. Three levels of memory are available from 64GB, 128GB and 256GB. Being a lo-cost smartphone, currently OnePlus 6 is a competitor to some of the expensive Android smartphones such as the Galaxy series.

1.5 Summary

In this Chapter we have briefly had a look at the history of the mobile phones. Additionally, various versions of the Android operating system is given as a table and the names and brief features of the currently available popular Android based mobile phones are given.

In the next Chapter we shall be seeing how to install the Basic For Android (B4A) IDE on our PC.

Chapter 2 • Getting started

2.1 Overview

This Chapter is about installing and starting to use the B4A software on a Windows based computer. B4A can be purchased from the **Anywhere Software** (https://www.b4x.com/), where four different versions are available depending upon the upgrade support and license requirements:

- Trial Version
- Standard Version
- Enterprise Version
- Site License

The Standard and Enterprise licenses are per developer only where each developer requires a separate single license.

The Trial Version is free of charge and as shown in the next section it can be downloaded from the web. This version is time limited to 30 days and also the program size is limited. The other versions have perpetual licenses.

The Standard Version costs $59 at the time of writing this book. This version includes 2 months of free upgrades and full access to the forums.

The Enterprise Version costs $119 and it includes 2 years of free upgrades in addition to full access to the forums.

The Site License costs $599 and it includes 30 licenses for the same organization. In addition, as in the Enterprise Version, this version includes 2 years of free upgrades and full access to the forums.

B4A academic licenses (for students, teachers, and researchers) are available for half the price, where full details can be obtained from the developers' web site:

https://www.b4x.com/store.html

2.2 Installing the Trial Version of B4A

The trial version can be run in **Remote Compilation Mode** or in **Local Compilation Mode**. The Remote Compilation Mode requires the installation of very few components, but it has limited features. In this section we will assume that the reader will use the trial version for a short time and then upgrade to the full version later on. Therefore we will install all the required components so that the trial version (and later the full version) runs in Local Compilation Mode with added features such as using an emulator, full debugging (called **Rapid Debugger**), and so on.

Before installing the B4A it is necessary to install the **Java SDK** and **Android SDK**. We will then install the **B4A-Bridge** on our Android device after installing the B4A. The B4A-Bridge

allows the IDE to connect to our Android device on a wireless link so that we can debug our application (we could also connect through the USB link as we shall see in a later Chapter).

The steps to install the trial version are given below (it is assumed that you will be installing on a Windows based PC):

Go to the Java SDK download page:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

- Click **Accept License Agreement** button
- Select the **Windows x86** (see Figure 2.1) even if you have a 64-bit machine by clicking the appropriate **Download** file
- Download and install the file (at the time of writing this book the filename was **jdk-8u172-windows-i586.exe** and the size of this file was 199MB)

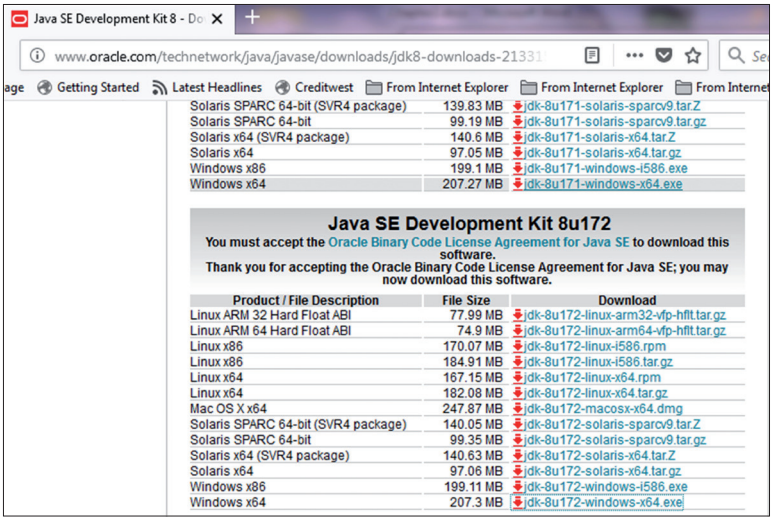


Figure 2.1 Select the product compatible with your PC

- Download the Android SDK from the following link (see Figure 2.2) and install in a folder, e.g. C:\Android (do not install in a folder with spaces in the folder name such as the program Files)

https://dl.google.com/android/installer_r23.0.2-windows.exe

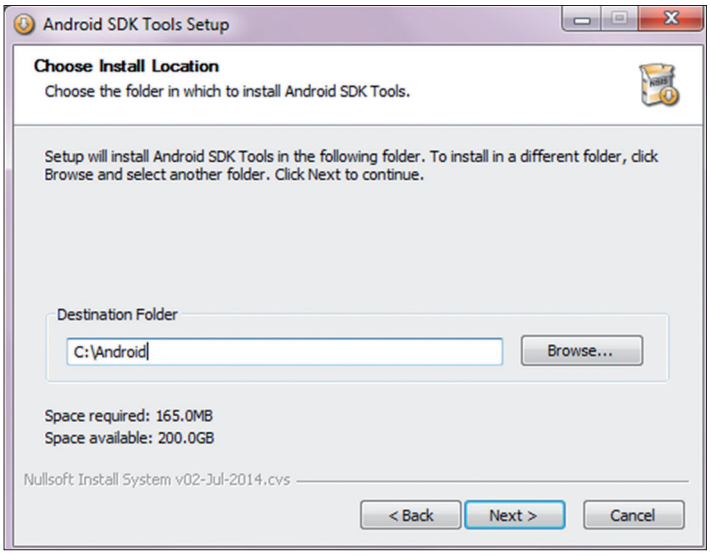


Figure 2.2 Download Android SDK

- Start the SDK manager during the installation (see Figure 2.3)

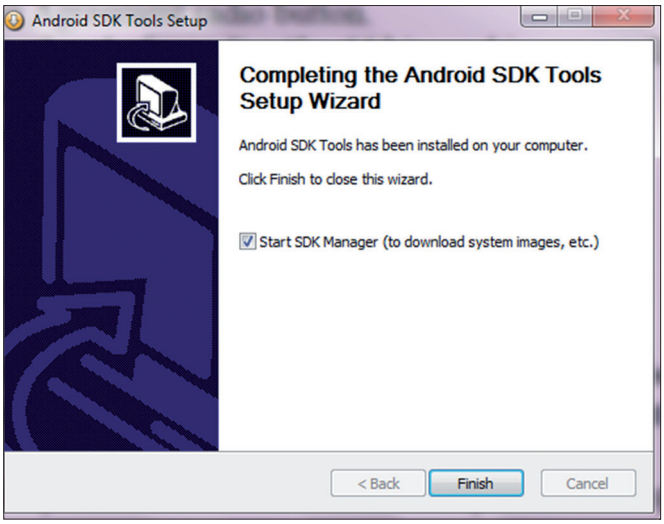


Figure 2.3 Start the SDK Manager

- We need the Android SDK Tools and SDK Platform-tools and the latest API. This is shown in Figures 2.4 and 2.5. Click **Install Selected** and then in the next form click **Accept License** and **Install** to install all the selected packages. Wait until the installation is complete.

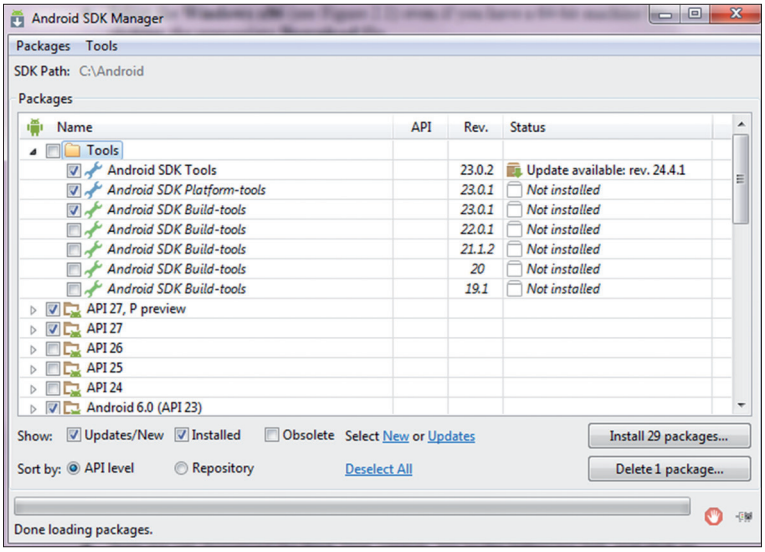


Figure 2.4 Android SDK Manager

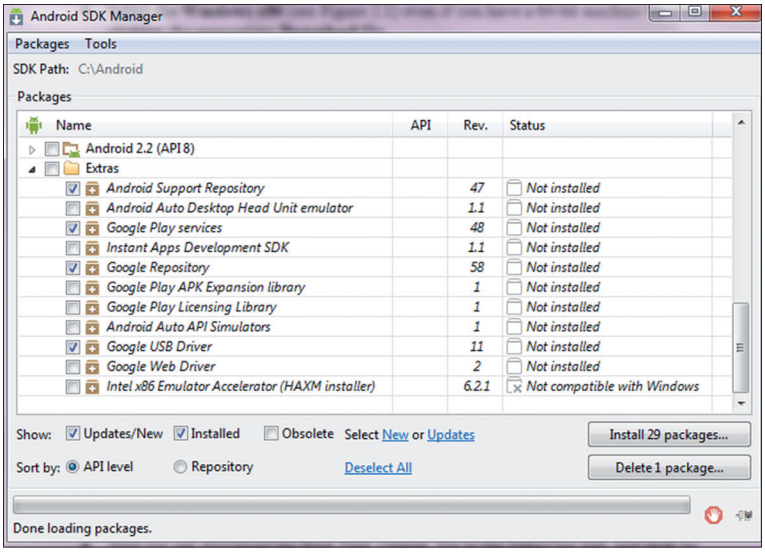


Figure 2.5 SDK Manager Extras

- Now we can download the B4A Trial version. Go to the following link and click to download and install the Trial Version (file **b4a-trial.exe**):

  https://www.b4x.com/b4a.html

- Start the B4A. You should see a Welcome message saying that you are running the Trial Version of B4A. We will now have to configure our environment (see Figure 2.6). Click **Tools -> Configure Paths** and check to make sure that file

javac.exe is in the correct path. If not, click Browse to include this file in the path. Also, check to make sure that file **android.jar** is in the correct path and if not click Browse to include it. Two types of libraries are used by B4A: Standard libraries which are distributed with B4A, and Additional libraries which are written by users and they are not part of B4A distribution. It is recommended to create the folder Additional libraries under B4A for future use. Shared module files can be shared between different projects and it is recommended to create a folder for these files.

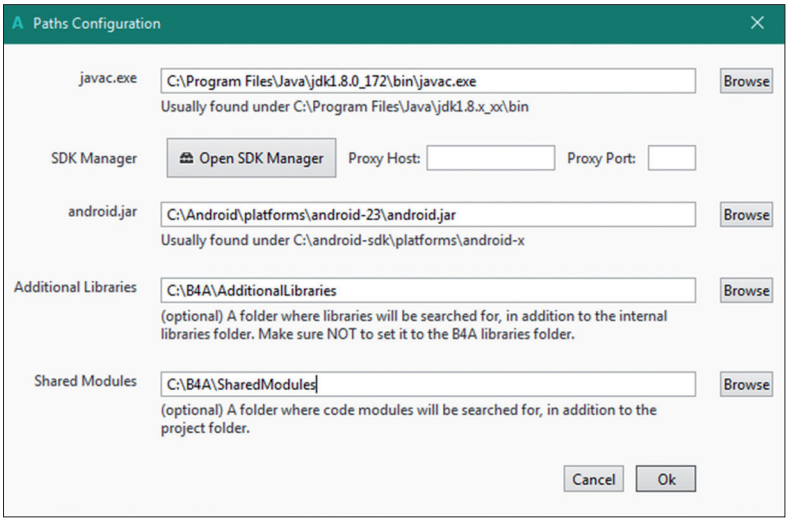


Figure 2.6 Configuring the B4A environment

- We should now install the **B4A-Bridge** so that we can communicate with our Android device over a wireless link. For this step, you should download the B4A-Bridge apps from Google Play to your Android device. Figure 2.7 shows the apps activated on an Android mobile phone.

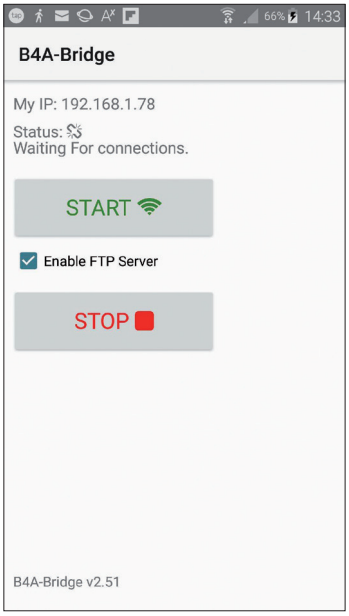


Figure 2.7 The B4A-Bridge apps

- We should now be ready to develop and run our first application which is described in the next Chapter.

2.3 Installing the Standard Version of B4A

The Standard version (or the other non-trial versions) can be purchased from the following link:

https://www.b4x.com/store.html

On the first run B4A will ask you to first locate the license file and afterwards it will ask you for the email address you have used when you have purchased B4A. You should copy the license file (named: b4a-license.txt) to a convenient place like the desktop. It is important to note that the license file is not a text file and it should not be opened with a text editor.

The Standard (and other non-trial versions) version overwrites the Trial version automatically. The steps to install the Standard version are given below:

- After purchasing the B4A, download it and install by clicking file **b4a.exe**
- Activate B4A and locate the folder of the license file (e.g. Desktop)
- Enter the e-mail address that you used when you purchased the product
- It is recommended that you register to the B4A forum to get information and help on various topics. The link of the forum is:

  https://www.b4x.com/android/forum/

At the time of writing this book the latest version of B4A was 8.00.

2.4 Getting Help

There are many sources that the readers can get help on B4A. The links to some such sources are given below.

Video Tutorials

There are several video tutorials on various B4A topics at the following link:

https://www.b4x.com/etp.html

Booklets and Guides

Various B4A booklets and guides are available at the following link:

https://www.b4x.com/android/forum/threads/b4a-beginners-guide.9578/

On-line Tutorials and Examples

Tutorials on various B4A topics are available at the following link:

https://www.b4x.com/android/forum/forums/tutorials-examples.27/

On-line Help

On-line help is available inside B4A by clicking the Help menu.

Beginners Guide

B4A Beginners Guide is available at the following link:

http://d1.amobbs.com/bbs_upload782111/files_53/ourdev_724596SVV4X2.pdf

Books

You may also find the following B4A books useful as it covers all aspects of B4A in detail:

B4A: Rapid Android App Development using BASIC, by: Wyken Seagrave
B4A: Ultra-fast Android App Development using BASIC, by: Wyken Seagrave

Additional Libraries

Additional libraries are available at the following link:

https://www.b4x.com/android/forum/forums/
additional-libraries-classes-and-official-updates.29/

2.5 Summary

In this Chapter we have seen how to install the Trial version and the Standard version of the B4A software and the B4A-Bridge apps. In the next Chapter we shall be looking at developing a very simple program to make ourselves familiar with using the B4A.

.

Chapter 3 • My first B4A program

3.1 Overview

In the last Chapter we have seen how to install the Trial and Standard Versions of the B4A on a Windows based PC. In this Chapter we shall be developing a very simple program and look at the basic features of the B4A Integrated Development Environment. In this Chapter we shall display the message **Hello from the B4A...** on our Android mobile phone.

3.2 Running the B4A IDE

After starting the B4A program you should see the Welcome screen. Click **Close** to see the B4A main screen as shown in Figure 3.1.

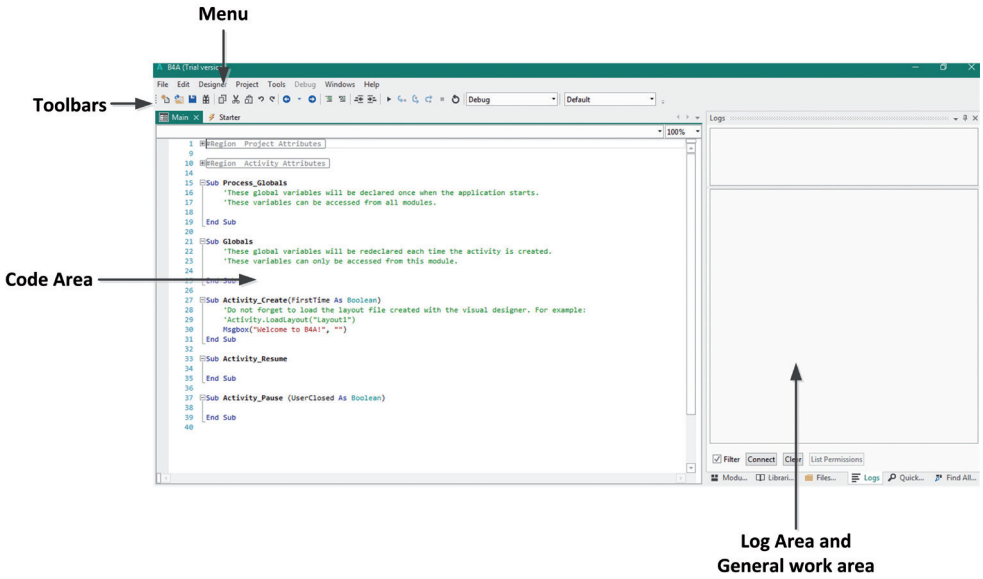


Figure 3.1 B4A main screen

The Code Area is where we will be writing and editing our code. When the B4A is started a sample template is displayed on the screen to help the users to start writing their programs. The Code Area is divided into regions where the contents of these regions can be expanded by clicking the "+" signs at the left of the regions. Clicking the "-" sign at the left of a region collapses the region so that its content are not displayed. The following regions are displayed by default:

- Project Attributes
- Activity Attributes
- Process_Globals
- Globals
- Activity_Create
- Activity_Resume
- Activity_Pause

The Project Attributes and the Activity Attributes determine the features of the current project and the current activity respectively. Normally you will not need to change the contents of these two regions. The default contents of these two regions are shown in Figure 3.2.

```
#Region  Project Attributes
    #ApplicationLabel: B4A Example
    #VersionCode: 1
    #VersionName:
    'SupportedOrientations possible values: unspecified, landscape or portrait.
    #SupportedOrientations: unspecified
    #CanInstallToExternalStorage: False
#End Region

#Region  Activity Attributes
    #FullScreen: False
    #IncludeTitle: True
#End Region
```

Figure 3.2 Default contents of Project and Activity Attributes

The other regions are in the form of subroutines. A B4A application is made of one or more activities where an activity is similar to Windows forms. An activity can be killed if it is not in the foreground in order to preserve memory. **Process_Globals** is used to store global variables which are declared once when the application starts and these variables can be accessed from all modules in the program. **Globals** can only be accessed from this module and they will be re-declared each time the activity is created. **Activity_Create** is where our main program code is and the code in this region is called after the program starts. i.e. an activity is created. The FirstTime parameter tells us whether or not this is the first time that this activity has been created. This parameter can be used to initialize certain variables if for example it is True. **Activity_Pause** is called when an activity moves from the foreground to the background. This can happen for example when a different activity starts, the Home or the Back buttons are pressed etc. **Activity_Resume** is called after **Activity_Create** finishes, or after resuming a paused activity so that the activity returns from the background to the foreground.

All projects contain a module called **Starter**. This module can be expanded by clicking it at the top left hand side of the screen next to Main. This is the first module to be executed (if it exists) when the app begins. By default the subroutines inside the Starter are blank and therefore the Main activity is the first code to be executed when the app begins. The Starter only starts once and is then retained in memory until the device is powered off. It is recommended to declare the **Process_Globals** here since the Starter is retained in memory.

3.3 Displaying a Message On the Android Mobile Device

The steps to display the message **Hello from the B4A...** are given below. In this example we will be displaying the message on the actual Android mobile phone:

- Insert the message inside the **Activity_Create** subroutine as shown in Figure 3.3.

```
Sub Activity_Create(FirstTime As Boolean)
    'Do not forget to load the layout file created with the visual designer. For example:
    'Activity.LoadLayout("Layout1")
    Msgbox("Hello from the B4A...", "")
End Sub
```

Figure 3.3 The Activity_Create subroutine

- Enable the compiler **Release** mode by clicking on the **Debug** window as shown in Figure 3.4.

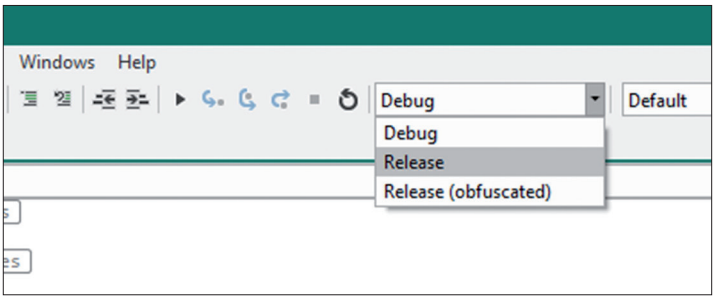


Figure 3.4 Enable the Release mode

- Make sure that your Wi-Fi is up and running and both your PC and the mobile phone are connected to the Wi-Fi.

- Start the B4A-Bridge apps on your mobile phone and enter the IP address (e.g. 192.168.1.78)

- Click the **START** button on the apps on your mobile phone. You should see the message **Waiting For connections** on your mobile phone.

- Click **Tools -> B4A Bridge** and then **Connect.** You will be prompted with **New IP** to connect to your mobile phone. Enter the IP address displayed on your mobile phone.

- Click **Project -> Compile & Run** (or press **F5**) to compile the program. You will have to give a name to your project at this stage. You should see the success message on your PC as shown in Figure 3.5.

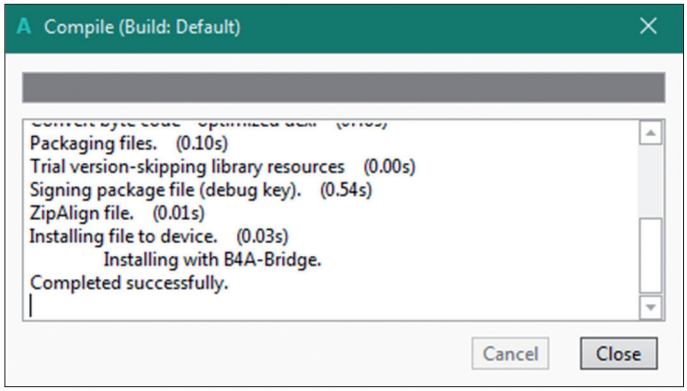


Figure 3.5 Successful compilation and run

- When prompted, click **INSTALL** to install the apps on your mobile phone and click **OPEN** to open it. You should see the message **Hello from the B4A...** displayed on your mobile phone as shown in Figure 3.6.

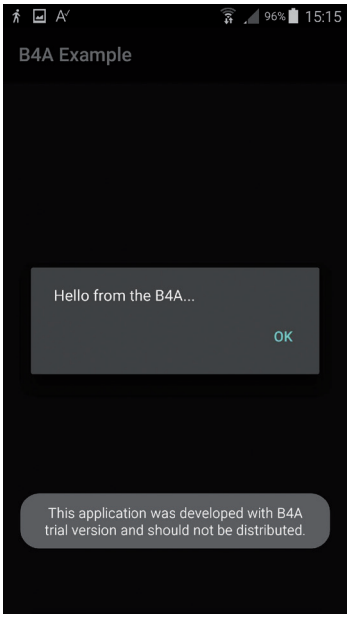


Figure 3.6 Message on the mobile phone

3.4 Summary

In this Chapter we have seen how to create a very simple program that displays a message on our Android mobile phone. In the next Chapter we shall be looking at the development of a program that also uses two buttons to display two different messages when the buttons are pressed.

Chapter 4 • My second B4A program

4.1 Overview

In the last Chapter we have seen how to create a very simple program that displays a message on our Android mobile phone. In this Chapter we shall be learning how to create two buttons called ON and OFF in our apps using the B4A Designer tool. Pressing ON will display the message **Hello from B4A** on our Android device, and pressing OFF will remove this message.

4.2 Running the Designer

The Designer tool is used to create user friendly applications on our Android device with buttons, textboxes, checkboxes, labels, etc. To start the Designer tool, click **Designer -> Open Designer** in the main menu at the top of the screen. The Designer screen will be displayed as in Figure 4.1. In B4A, a page that can be displayed is called an **Activity**, and a control that can be added to the page is called a **View**.

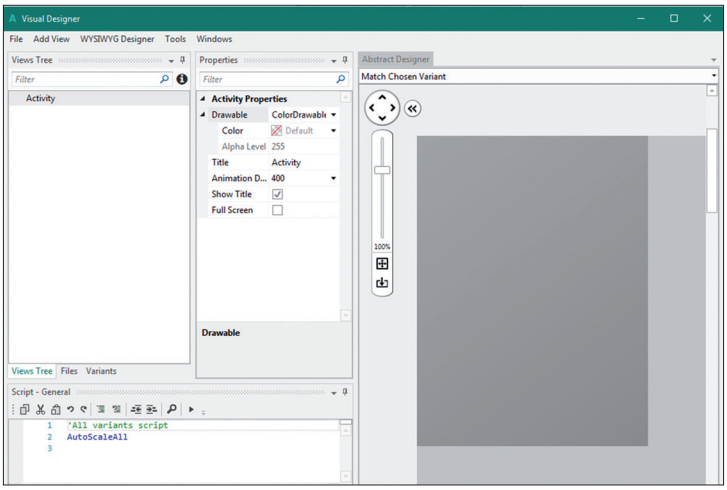


Figure 4.1 Designer screen

The right hand side of the screen is called the **Abstract Designer** and this is where we place the various controls (or the views). Notice that what we see on the Abstract Designer screen is not same as what will be displayed on the actual Android device. As we shall see later, we can connect to the actual device to see the screen in WYSIWYG (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et). The middle part of the screen shows the properties of the activity or the selected control. The left part of the screen shows the **Activity**.

First of all, change the background colour of the activity to white in the Activity Properties so that the buttons can be seen clearly (notice that the Designer screen is not WYSIWYG and you will not see the correct colours). We now want to add a button to the Abstract Designer screen. Click **Add View -> Button** and a button control will be added to the screen. We now have to configure the properties of this button. Set the Name of this button to **btnON** and its Text to **ON** as shown in Figure 4.2 and adjust the button size as required.

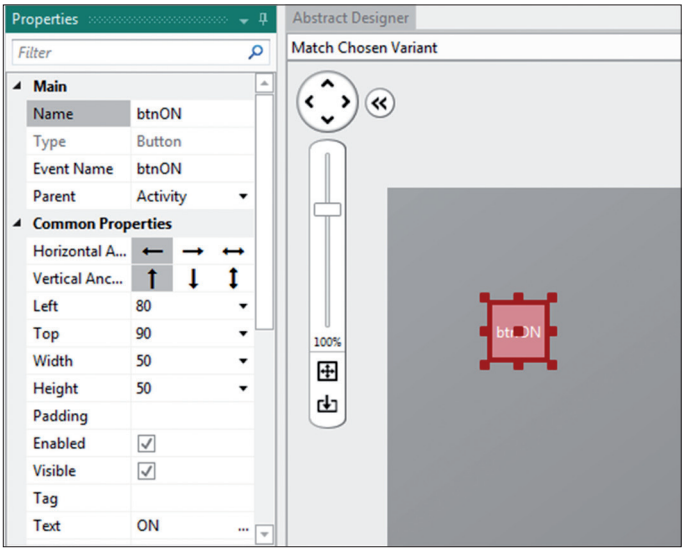


Figure 4.2 Configure the button

Now, create another button, name it as **btnOFF**, set its Text to **OFF**, and place it close to the first button as shown in Figure 4.3. You should now save your design by clicking **File -> Save**. In this example the design is saved with the name **Hello**.

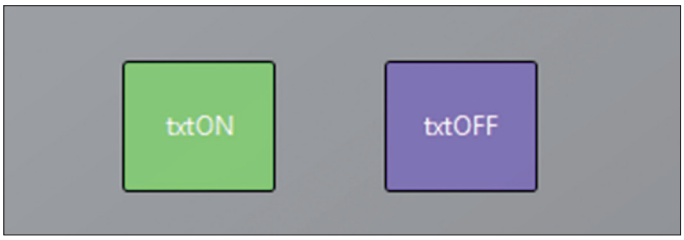


Figure 4.3 Abstract Designer with two buttons

We now have to generate code for our two buttons. Click **Tools -> Generate Members**. Expand **txtON** and select **Click**. Similarly, expand **txtOFF** and select **Click** so that the buttons will respond to clicks (see Figure 4.4). Click **Generate Members** to generate code, and then close the Generate Members screen.

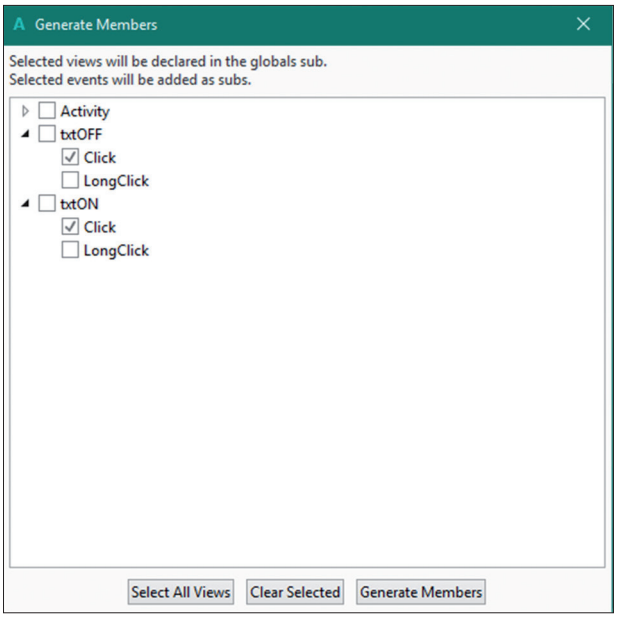


Figure 4.4 Generate members

Notice that the following two empty subroutines are created in the main program with the word **Click** added to the button names. This indicates that the code inside the subroutines will be activated when the corresponding buttons are clicked:

```
Sub btnON_Click

End Sub

Sub btnOFF_Click

End Sub
```

Now, we have to add code inside these subroutines to tell the program what action to take when the button is clicked. The message **Hello from B4A** will be displayed when button **ON** is clicked and this message will be removed (replaced by spaces) when button **OFF** is clicked. Here, the message is displayed using the **MsgBox** statement. The require code for the subroutines are:

```
Sub btnON_Click
   MsgBox("Hello from B4A", "")
End Sub

Sub btnOFF_Click
   MsgBox("                          ", "")
End Sub
```

Notice that we also have to load the layout created by the Abstract Designer. This is done inside the subroutine **Activity_Create** by the statement **Activity.LoadLayout("Hello")** where **Hello** is the name of the saved layout.

You should now activate the B4A-Bridge apps on your mobile phone and connect to the device by setting **Tools -> B4A Bridge -> Connect** and give the IP address as in the previous project example. Compile and run the program and click **INSTALL** to install it on the mobile phone. **OPEN** the application and run it as shown in Figure 4.5. The complete program listing is shown in Figure 4.6.

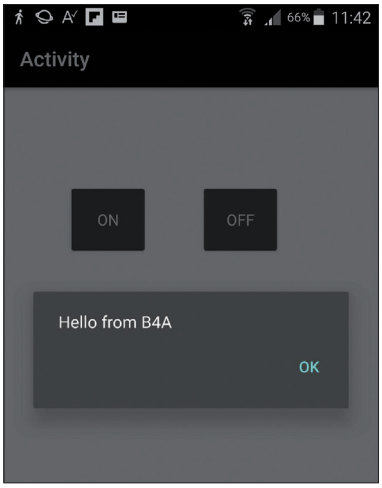


Figure 4.5 Application on the mobile phone

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
End Sub

Sub Activity_Create(FirstTime As Boolean)
  'Do not forget to load the layout file created with the visual designer.
For example:
```

```
  Activity.LoadLayout("Hello")
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub


Sub txtON_Click
  Msgbox("Hello from B4A", "")
End Sub

Sub txtOFF_Click
  Msgbox("              ", "")
End Sub
```

Figure 4.6 Program listing

Notice that during the Abstract Designer process the positions, shapes, and colours of the controls on the actual Android device can be seen interactively by activating the B4A-Bridge on the mobile device and then clicking WYSIWYG -> Connect on the Designer menu. This is shown in Figure 4.7 with the heading **Activity**.

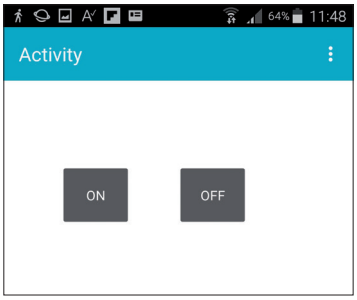


Figure 4.7 Positions and colours of the controls on the actual device

After finishing the design and testing you should disconnect the B4A-Bridge by clicking **Tools -> B4A -> Disconnect** on your PC and also **STOP** the B4A-Bridge on your Android device.

4.3 Summary

In this Chapter we have seen how to create a simple application with two buttons and a label. In the next Chapter we shall create a simple calculator program and also learn how to use the debugger and also how to communicate with the Android device over the USB link.

Chapter 5 • My third B4A program

5.1 Overview

In the last Chapter we have seen how to create a simple application with two buttons. In this Chapter we shall create a simple calculator program and also learn how to use the debugger and also how to communicate with the Android device over the USB link.

5.2 Simple Calculator Program

The aim of this example is to show how to use some other controls (views) of the B4A. This is a simple calculator program that can do addition, multiplication, division, and subtraction. The user enters two numbers and then clicks one of four buttons to select the required operation. The answer of the calculation is displayed by a MessageBox.

The steps to design the calculator program are given below (You can connect to the Android mobile device via the Designer WYSIWYG menu option to see the design on the actual device):

- Click **Designer -> Open Designer** to open the Designer screen
- Set the background of the **Activity Properties** to white (colour: #FFFFF)
- Add a Label by clicking **Add View -> Label**. Name this label as **txtLabel** and set its Text to **SIMPLE CALCULATOR**, set the size of the Text to 18, and its colour to Black (color: #000000)
- Add a Label, name it as **txtLabelNo1** and set its Text to **Enter First Number:**, and set its colour to Black (color: #000000)
- Add an **EditText** box next to the label and name this box as **txtNo1.** Set the Input Type to NUMBERS, and set the Text Color to Black (color: #000000)
- Add another Label, name it as **txtLabelNo2** and set its Text to **Enter Second Number:**, and set its colour to Black (color: #000000)
- Add an **EditText** box next to the label and name it as **txtNo2**. Set the Input Type to NUMBERS, and set the text Color to Black (color: #000000)
- Add 4 buttons with names **txtButtonPlus**, **txtButtonMinus**, **txtButtonMult,** and **txtButtonDiv**. Set the Texts of these buttons to **+**, **-**, **x**, **/** respectively, and the text sizes to 20

Figure 5.1 shows the design layout for this example. Click **File** to save the design. In this example the design is given the name **Calculator**.

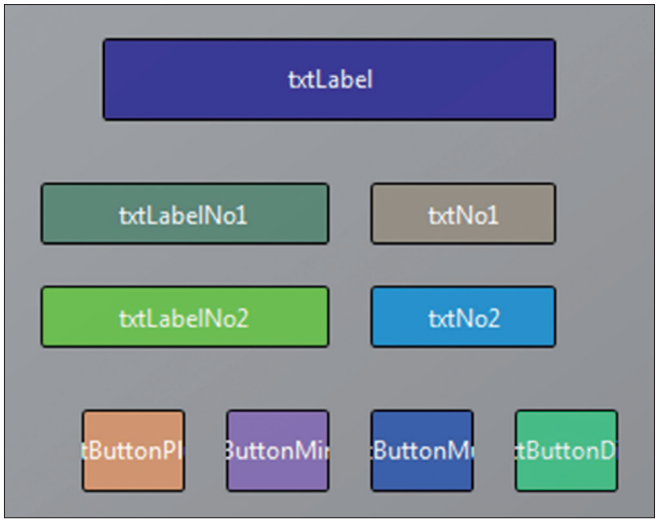


Figure 5.1 The design layout

Here, the **EditText** boxes store the two numbers that will be operated on. The result of the operation will be displayed by a MessageBox.

Now, we can see how the layout will look like on the actual device. Start the B4A-Bridge on your Android device and connect to it by clicking **Tools -> B4A -> Connect**. Click **WYSIWYG** in the Designer screen and then click **Connect**. You should see the screen layout as in Figure 5.2. **Notice that once connected, you can adjust the location and size of the controls on the actual device screen by manipulating them on the designer menu**.

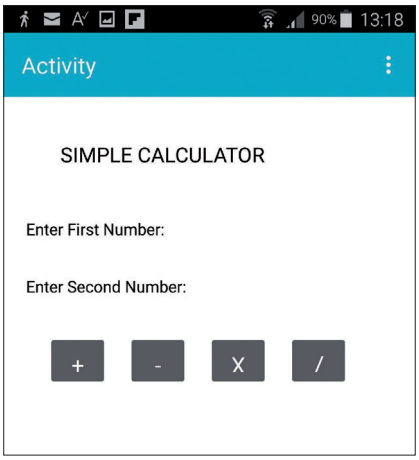


Figure 5.2 Screen layout on the actual Android device

Now, we have to generate code for the various controls on the screen. As in the previous project, click **Tools -> Generate Members** and set the button modes to **Click** so that the buttons respond when they are clicked, **EditText** boxes to **EnterPressed** so that we press

the Enter key after entering a number, and click also **Select All Views** so that we have the views defined in the **Globals** list so that their elements can be accessed (see Figure 5.3).

Click **Generate Members** to generate the code for the buttons.

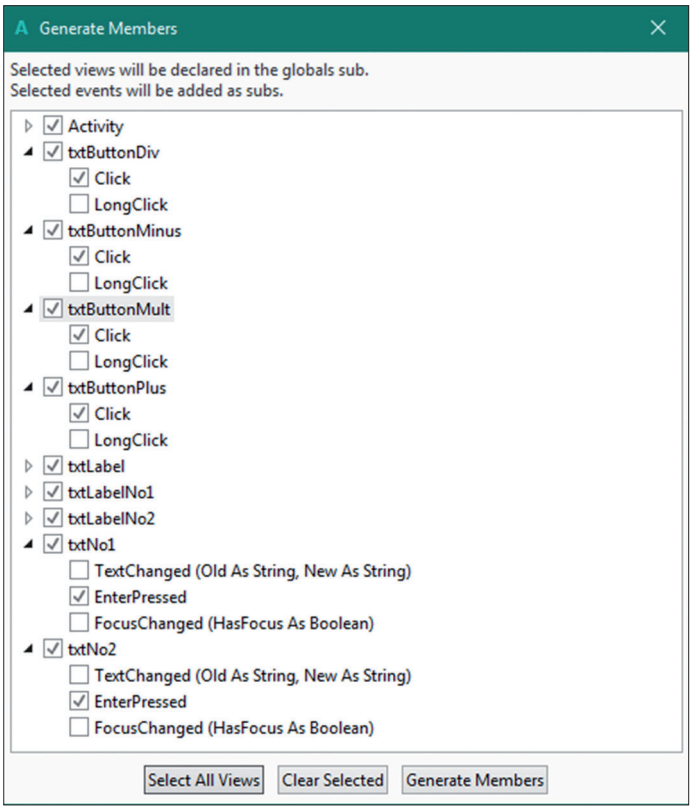


Figure 5.3 Generate Members

The following 4 subroutines are generated in the code:

```
Sub txtButtonPlus_Click
End Sub

Sub txtButtonMult_Click
End Sub

Sub txtButtonMinus_Click
End Sub

Sub txtButtonDiv_Click
End Sub
```

```
Sub txtNo1_EnterPressed
End Sub

Sub txtNo2_EnterPressed
End Sub
```

In addition, the following Globals are defined:

- Private txtNo1 As EditText
- Private txtNo2 As EditText
- Private txtButtonDiv As Button
- Private txtButtonMinus As Button
- Private txtButtonMult As Button
- Private txtButtonPlus As Button
- Private txtLabel As Label
- Private txtLabelNo1 As Label
- Private txtLabelNo2 As Label

Now, we have to write the code for these subroutines. As an example, when the **txtButtonPlus** button is clicked we want to add the two numbers and display their result in a MsgBox. Similarly, we have to do the required operations for the **txtButtonMult**, **txtButtonMinus**, and **txtButtonDiv** buttons. The required code for each subroutine is given below:

```
Sub txtButtonPlus_Click
    result = N1+N2
    Display_Result
End Sub

Sub txtButtonMult_Click
    result = N1*N2
    Display_Result
End Sub

Sub txtButtonMinus_Click
    result = N1 - N2
    Display_Result
End Sub

Sub txtButtonDiv_Click
    result = N1 / N2
    Display_Result
End Sub

Sub txtNo1_EnterPressed
    N1=txtNo1.Text
End Sub
```

```
Sub txtNo2_EnterPressed
   N2=txtNo2.Text
End Sub
```

Variables **N1**, **N2** and **result** are declared as integers in **Process_Globals**:
Dim result As Int

```
Dim N1 As Int
Dim N2 As Int
```

Variable **result** stores the result of the required mathematical operation. Subroutine **Display_Result** is then called to display the result in a MsgBox. The code of this subroutine is as follows. After displaying the result focus is set to EditText box **txtNo1** so that the user can carry out a new calculation by entering the first number. At the same time the contents of the two EditText boxes are cleared:

```
Sub Display_Result
     MsgBox(result, "RESULT")
     txtNo1.RequestFocus
   txtNo1.Text=""
     txtNo2.Text=""
End Sub
```

The designed layout is loaded to the mobile device by the statement: **Activity.LoadLayout("Calculator").** Notice that the designed file was saved with the name **Calculator**.

As described in the previous Chapter, activate the B4A-Bridge on your mobile phone, Click **Tools -> B4A Bridge** and then **Connect** to the mobile phone. Compile and run your code so that it is uploaded to the mobile phone. Figure 5.4 shows a typical run of the apps on the Android mobile phone where two numbers are multiplied. In this example, number 12 is types as the first number and Enter key is pressed, then number 5 is typed as the second number and the Enter key is pressed. Then the multiplication sign is clicked which then displays the result in a MsgBox as shown in Figure 5.5. Clicking the **OK** button in MsgBox re-starts the program. Notice that a numeric keyboard is displayed by the apps since the EditText box **Input** types were set to be **NUMERIC**.

Figure 5.4 Typical run on the mobile device

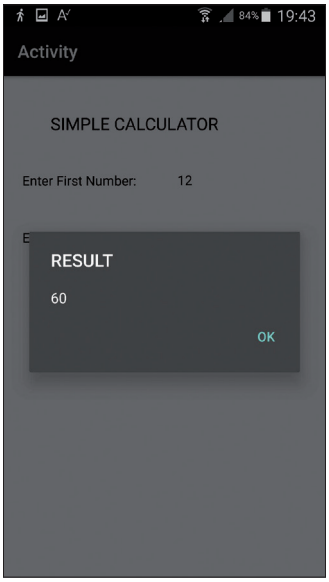


Figure 5.5 The result is displayed in the MsgBox

Figure 5.6 shows the complete program listing.

```
#Region  Project Attributes

#Region  Activity Attributes
```

```
Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
  Dim result As Int
  Dim N1 As Int
  Dim N2 As Int
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private txtNo1 As EditText
  Private txtNo2 As EditText
  Private txtButtonDiv As Button
  Private txtButtonMinus As Button
  Private txtButtonMult As Button
  Private txtButtonPlus As Button
  Private txtLabel As Label
  Private txtLabelNo1 As Label
  Private txtLabelNo2 As Label
End Sub

Sub Activity_Create(FirstTime As Boolean)
  'Do not forget to load the layout file created with the visual designer.
For example:
  Activity.LoadLayout("Calculator")
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub Display_Result
   Msgbox(result, "RESULT")
   txtNo1.RequestFocus
   txtNo1.Text=""
   txtNo2.Text=""
End Sub

Sub txtButtonPlus_Click
  result = N1+N2
  Display_Result
```

```
End Sub

Sub txtButtonMult_Click
 result = N1*N2
 Display_Result
End Sub

Sub txtButtonMinus_Click
 result = N1 - N2
 Display_Result
End Sub

Sub txtButtonDiv_Click
 result = N1 / N2
 Display_Result
End Sub

Sub txtNo1_EnterPressed
 N1=txtNo1.Text
End Sub

Sub txtNo2_EnterPressed
 N2=txtNo2.Text
End Sub
```

Figure 5.6 Program listing

5.3 Debugging

Debugging is the process of finding errors in a program. Using the debugger we can single step through a program and examine the values of the variables. B4A supports two debugging modes: **Legacy Debugging** and **Rapid Debugging**. The **Debug** compilation option must be enabled at the top of the main screen before a program can be debugged as shown in Figure 5.7

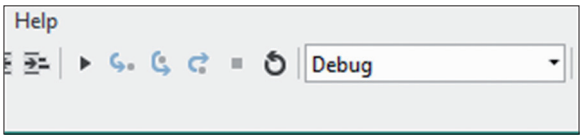


Figure 5.7 Enabling the Debug option

Legacy Debugger is simple and does not require the installation of the Java JDK. In order to use the Legacy Debugger we have to enable it as shown in Figure 5.8 by clicking **Tools -> IDE Options -> Use Legacy Debugger**.

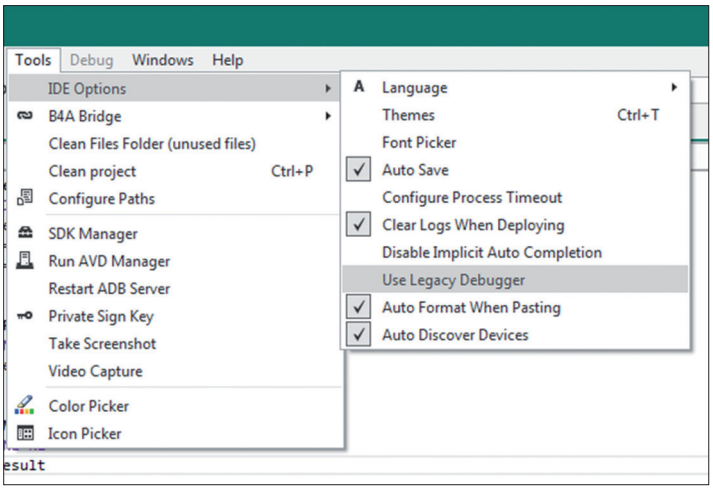


Figure 5.8 Enabling the Legacy Debugger

By default the Rapid Debugging is enabled and used. This debugger is very powerful as it allows the code to be modified while the apps is running without having to re-install it. When using a debugger we usually create Breakpoints and allow the program to run up to these breakpoints and then stop. At this point we can examine the variables if we wish. The steps of using the Rapid Debugger for the program given in Figure 5.6 are given below:

- Make sure that the **Debug** option is enabled (Figure 5.7)
- Activate the B4A-Bridge on your mobile device. Click **Tools -> B4A Bridge** and **Connect** to the mobile device as before
- Click at the left hand side (grey margin) of the statement where you wish to insert a breakpoint (clicking at the same point removes the created breakpoint). The code at the breakpoint will be highlighted in red. In Figure 5.9 the breakpoint is at statement **result = N1*N2**

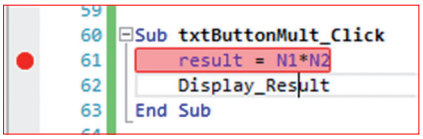


Figure 5.9 Breakpoint at the statement

- Compile and run the program. Enter two numbers (e.g. 5 and 2) on your mobile device and click **X** to multiply them
- The program will run and then stop at the breakpoint statement. This is highlighted with yellow colour. At the same time the mobile device will show that the program is paused, in this example at line 61 as shown in Figure 5.10

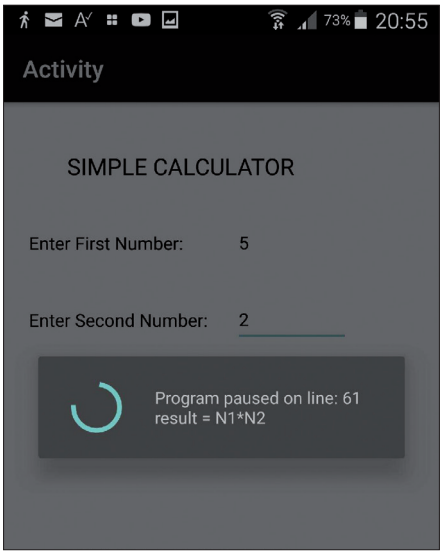


Figure 5.10 Program paused at the breakpoint

- Press key F8 once on your PC to step through the program. You will notice that the breakpoint is still highlighted with red and the program is paused at the next statement, highlighted in yellow.

- At this point you can examine the value of variable **result** by hovering the mouse over **result**. As shown in Figure 5.11 the value of result is 10 as expected.

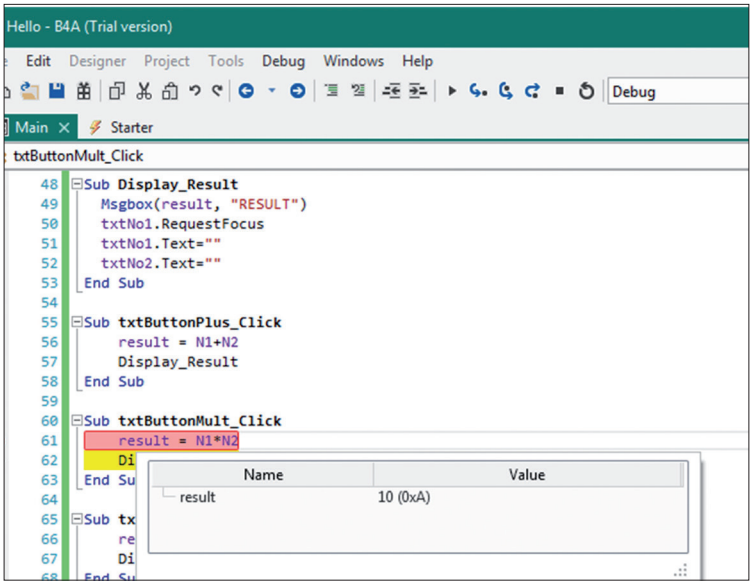


Figure 5.11 Examining the value of result

- We can also examine the value of a variable in the **Watch** window. All we have to do is type the name of the variable to be examined and press the Enter key, or click the **Add Watch Expression** icon at the top right hand side of the Watch window as shown in Figure 5.12.

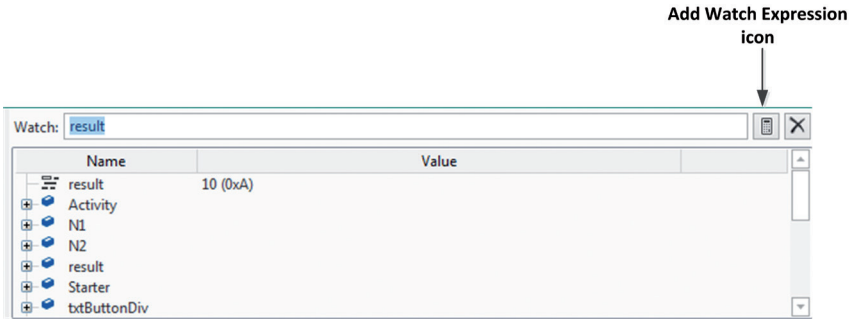


Figure 5.12 Add variable result to the watch window

Figure 5.13 shows the control commands available in the Debug menu and supported by the debugger. Function key F5 is used to continue program execution from the point it stopped. Function key F8 steps into the next statement and this is useful when we want to single step through our program. Function key F9 steps into the next line without entering any subroutine call. Function key F10 executes the program until it comes out of a sub-routine and then it pauses. The debugging is stopped by clicking the STOP button in the Debug menu.

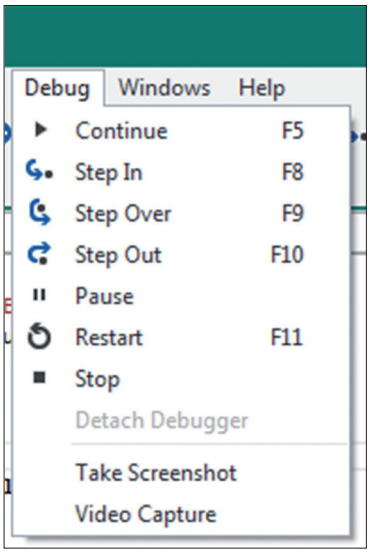


Figure 5.13 Debugger control commands

We can also use the **Log** statement with the debugger inside our program to display the value of a variable as the program is running. An example is shown in Figure 5.14 where

variable **result** is set to be logged. The results of logging are shown at the right hand side of the debug screen as shown in Figure 5.15.

```
Sub txtButtonMult_Click
    result = N1*N2
    Log("result = " & result)
    Display_Result
End Sub
```

Figure 5.14 Using the Log statement

```
Logger connected to:  samsung SM-N910F
--------- beginning of main
--------- beginning of system
*** Service (starter) Create ***
** Service (starter) Start **
** Activity (main) Create, isFirst = true **
** Activity (main) Resume **
** Activity (main) Resume **
result = 25
** Activity (main) Pause, UserClosed = false **
```

Figure 5.15 Displaying Log result

5.4 Using the USB Connection

There are three methods that we can use to debug a program:

- Connecting to the physical device using B4A-Bridge
- Connecting to the physical device using USB cable
- Using the built-in emulator

Connecting to the physical device using a USB cable has the advantage that the application development can take place even if there is no Wi-Fi (the B4A-Bridge connection requires Wi-Fi connection). Another feature of the USB connection is that you can take screen shots and capture video when using the full version of B4A with the USB connection. Click on **Tools -> Tale Screenshot** to take the screen shot from the mobile device (or from the emulator). The slider can be used to re-size the image, the image can be rotated, and saved to the **Clipboard** if required. Notice that the screen image of a mobile device can be taken on the actual physical device. For example, on the Samsung Galaxy Note 4, press the power key and the menu keys together for a few seconds to take a screenshot of the current screen. It is also possible to capture video when the device is connected through the USB port. Click **Tools -> Video Capture** to capture video from your mobile device. This can be useful for demonstrating your application as a video.

The steps to make connection to the physical device using a USB cable are given below:

- Enable USB Debugging on your Android device. On a mobile phone (Samsung Galaxy Note 4 is used in this example) click **Settings -> About device** and then click on the **Build number** 7 times. You should see a message saying that the developer mode has been enabled.

- On the Android device click **Settings -> Developer options** and tick to enable **USB debugging** (see Figure 5.16)

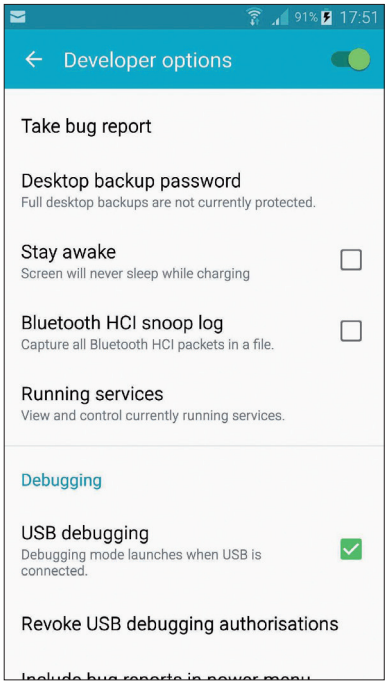


Figure 5.16 Enable USB debugging

- You will also need to install the Google USB driver for debugging. For the Samsung mobile phones this was located at the following link at the time of writing this book (you may have to look at the Internet if you have a different type of mobile phone):

  https://developer.samsung.com/galaxy/others/android-usb-driver-for-windows

- As a result of a security change in Android 4.2.2 related to USB debugging, it was necessary to install the new version of the Android SDK. This can be installed from the following link (you should also install at least one platform):

  http://dl.google.com/android/installer_r21.1-windows.exe

- Connect your Android mobile phone to the USB port of the PC. You should be prompted with a message asking you to allow USB debugging as shown in Figure 5.17. Click **OK** to accept.

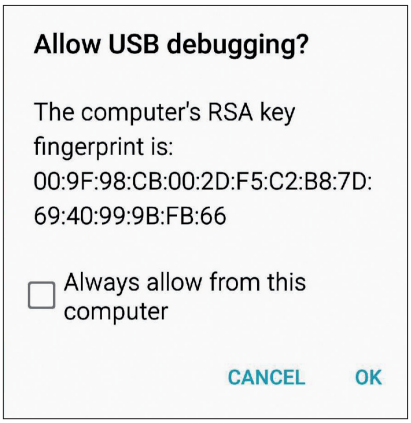


Figure 5.17 Allow USB debugging

- You can now compile and run your application where the compiled code will be uploaded to your mobile device automatically (no need to activate the B4A-Bridge or any other apps on your mobile device).

- You can also use the Designer screen to design a layout with views on it and then Connect to the mobile device using the WYSIWYG menu option to see your design interactively in real-time.

5.5 Summary

In this Chapter we have designed a simple calculator program. In addition, we have seen how to use the debugger to debug a program. Connection to an Android mobile device through the USB cable has also been described with the advantages of the USB connection. Next Chapter is about the language reference of B4A where we shall be learning about the variables, statements, functions, etc of B4A.

Chapter 6 • B4A language reference

6.1 Overview

In this Chapter we shall be looking at the B4A language in greater detail and learn about the variables, comment lines, conditional statements, arrays etc.

6.2 Comments

Comment lines in B4A start with a single colon character. An example is given below:

```
' This is a B4A comment line
' Comment lines are for information only and they are not executed.
' Comment lines are useful for program maintenance
```

We can also use Block Comments in a program where part of a program can be commented. Block comments are inserted and removed (Uncommented) using the icons shown in Figure 6.1. You can highlight the lines to be commented and then click the Block Comment icon to comment all the selected lines.

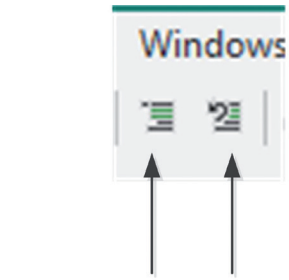


Figure 6.1 Block Comment icons

6.3 Indentation

It is a good practice to indent your code to make it easier to follow. You can either indent the code during writing, or click the indentation button to indent the code automatically after writing it. The indentation icon is shown in Figure 6.2. The Outdent icon removes the indentation.

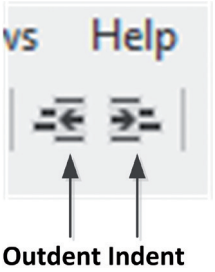


Figure 6.2 Indentation icon

Figure 6.3 shows part of a code which is not indented. Select the code to be indented and then click the indentation icon. The resultant code is shown in Figure 6.3.

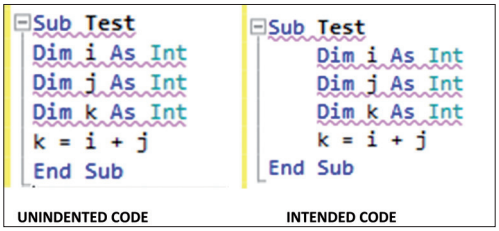


Figure 6.3 Indenting code

6.4 Case Sensitivity and Statement Separation

B4A is case insensitive where the editor automatically changes the case of a keyword. It is allowed to use more than one statement on a line by separating the statements by semi-colons. An example is shown below:

```
Dim i as Int: Dim j as Int: Dim k as Int
```

6.5 Constants

Constants are declared using the keyword **Const**. The value of a constant cannot be changed inside a program. An example is given below:

```
Dim Const MyPi As Double = 3.14159
```

6.6 Variables

Variables are identified by giving them names. Variable names must begin with letters and must use the characters A-Z, a-z, numbers 0-9, and the underscore character. Spaces, brackets, mathematical symbols, or any other symbols cannot be used as variable names. In addition, reserved B4A language names such as Int, Double, Float, While, Sub, If, For etc cannot be used as variable names.

There are two types of variables in B4A: Primitive Types, and Non-Primitive Types.

Primitive Types

These variables pass their values when they are passed to a Sub, or when assigned to other variables. Table 6.1 shows the primitive variable types allowed in B4A:

Primitive Type	Type	No of bits	Min value	Max value
Boolean	boolean	1	FALSE	TRUE
Byte	byte	8	-128	127
Short	integer (signed)	16	-32768	32767
Int	integer (signed)	32	-231	231 – 1

Long	integer (signed)	64	-263	263 - 1
Float	floating point	32	-2-149	(2-2-23)*2127
Double	double precision	64	-2-1074	(2-2-52)*21023
Char	character	16	0	65535
String	string	array of chars		

Table 6.1 B4A primitive types

Variables in B4A are declared using the **Dim** statement. Although it is not necessary to declare a variable before it is used, it is good practise to declare them before use. Any undeclared variables are assumed to be of type **String**. The editor generates a warning message when an undeclared variable is used in a program statement. The advantage of declaring all the variables is that if an attempt is made to assign wrong data type to a variable then this will generate a run time error. A compiler error is generated if an unassigned variable (whether declared or not) is used in a statement.

The format of the Dim statement is as follows:

```
Dim variable-name As variable-type
```

Some examples variable declarations are given below:

```
Dim Cnt As Int
Dim Total = 0 As Int
Dim Sum As Double
Dim i, j, k, m As Int
Dim v = 12.35 As Double
Dim Sum as Double, Cnt as Int, p As Int
Dim i = 0, j = 1, k = 2, m = 10 As Int
Dim MyText = "Computer" As String
```

In addition to **Dim**, we can also use the keywords **Public** or **Private** to declare variables. In **Sub Process_Globals** we can define variables using **Public** and these variables can be accessed from other modules in the program. Declaring variables with the keyword **Private** in **Sub Process_Globals** hides them from other modules in the program. Variables declared as **Private** in **Sub Globals** are always private to the module where they are defined. In B4A variable types are automatically converted from one type to another one as required. In the following example variable **Cnt** is declared as an integer and is assigned number 25 to it. String variable **Txt** is then assigned to **Cnt** where the integer is converted into string automatically and is assigned to variable **Txt**:

```
Dim Cnt As Int, Txt as String
Cnt = 25
Txt = Cnt
```

We can similarly convert string variables to integer. In the following example, string variable **Txt** stores string **50** which is converted into an integer and stored in integer variable **Cnt**:

```
Dim Cnt as Int, Txt as String
Txt = "50"
Cnt = Txt
```

6.7 Arrays

Arrays in B4A can have multiple dimensions, although one dimensional arrays are commonly used in most programming projects.

A one dimensional array is declared by giving its name, its size, and its type. If the type of the array is not declared then it defaults to **String**. In the following example, array called **Sum** holds 10 integers indexed as Sum(0) to Sum(9):

```
Dim Sum(10) As Int
```

The above one dimensional array holds a row of 10 integer numbers indexed 0 to 9 as follows:

0	1	2	3	4	5	6	7	8	9

Similarly, for example, a two dimensions integer array called **Cnt** having 4 elements can be declared as follows:

```
Dim Cnt(2, 2) As Int
```

The above two dimensional array has two rows and two columns each indexed from 0 to 1 as follows:

0	1
1	

Data is assigned to array elements by simply using the index. In the example below, the 3rd element of array Sum is assigned number 25:

```
Sum(2) = 25
```

The length of an array, i.e. how many objects there are in the array can be found using the keyword **Length**. In the following example, the length of array **Sum** is returned as 10 and stored in variable **L**:

```
L = Sum.Length
```

Arrays can be initialized at the time of declaration, firstly without specifying their sizes. In the following example, array **Cnt** is initialized with even numbers from 2 to 10:

```
Dim Cnt() As int
Cnt = Array As Int(2,4,6,8,10)
```

6.8 Lists

The sizes of arrays are fixed and cannot be changed during run time. **Lists** on the other hand are similar to arrays, but have the advantages that their sizes can be changed by adding or removing items from them. Lists however must be initialized before items are added to them. An example **List** is given below where **List names** is initialized and four names are added to it. Then, item 2 (Michael) is assigned to string variable **MyName** (notice that the first index is 0 as in arrays):

```
Dim names As List
Names.Initialize
Dim MyName As String
Names.Add("John")
Names.Add("Mary")
Names.Add("Michael")
Names.Add("Smith")
MyName = Names.Get(2)
```

We can remove an item from a List by specifying its index number. For example, to remove Mary from the list we can write the following statement:

```
Names.RemoveAt(1)
```

To add an item at a given index we can write (adding an item will shift down all the items with greater index numbers) the following which will add **Susan** to index number 2:

```
Names.InsertAt(2, "Susan")
```

The size of a List can be obtained as:

```
Names.Size
```

The value of an item can be changed by specifying its index number and the new item. In the following example the item at index 2 is changed to Harry:

```
Names.set(2, "Harry")
```

The contents of a List can be cleared by:

```
Names.Clear
```

It is also possible to sort the contents of a List in ascending or descending order. Examples are:

`Names.Sort(True)`	sorts in ascending order
`Names.Sort(False)`	sorts in descending order

In addition, the statement **SortCaseInsensitive(True/false)** can be used to sort the List ignoring the case of the characters.

The index of an item in a List can be obtained using the **IndexOf** keyword. In the following example, variable **k** is assigned number 3 which is the index of **Smith** in the List:
Dim names As List

```
Dim k As Int
Names.Initialize
Dim MyName As String
Names.Add("John")
Names.Add("Mary")
Names.Add("Michael")
Names.Add("Smith")
k = Names.IndexOf("Smith")
```

6.9 Maps

Maps are similar to Lists but their elements can be accessed using keys as well as index numbers. Maps must be initialized before they are used. An example use of the **Map** is shown below. In this example, **Student** is declared as a **MAP** and his Name, Surname, Age, and Gender are declared. The **Age** and Name of the **Student** are then retrieved using the **Get** keyword and stored in variables **sAge** and **sName** respectively:

```
Dim Student As Map
Student.Initialize
Dim sAge As Int
Dim sName As String
Student.put("Name","John")
Student.Put("Surname", "Harris")
Student.Put("Age", 25)
Student.Put("Gender", "Male")
sAge = Student.Get("Age")
sName = Student.Get("Name")
```

The size of a MAP (number of items in it) is returned using the keyword **Size**:

```
Student.Size
```

The items inside a MAP can all be cleared using the keyword **Clear**:

```
Student.Clear
```

An item can be removed from a MAP by specifying its key:

```
Student.Remove("Name")
```

The keyword ContainsKey can be used to test if the MAP contains the specified key. A TRUE is returned if the key is found. In the following example, variable **k** is initialized to 0. The code then sets **k** to 1 since the MAP contains the key **Surname**:

```
Dim Student As Map
Dim k = 0 As Int
Student.Initialize
Dim Age As Int
Dim sName As String
Student.put("Name","John")
Student.Put("Surname", "Harris")
Student.Put("Age", 25)
Student.Put("Gender", "Male")
If Student.ContainsKey("Surname") Then k=1
```

6.10 Mathematical Operators

The mathematical operators supported by B4A are shown in Table 6.2. **Power** has the highest precedence, followed by **Mod**, **multiplication** and **division**, and finally **addition** and **subtraction**.

Operator	Operation
Power	Power of, e.g. Power(x , y)
Mod	Modulo, e.g. x Mod y
*	Multiplication, e.g. x * y
/	Division, e.g. x / y
+	Addition, e.g. x + y
-	Subtraction, e.g. x – y

Table 6.2 Mathematical operators

6.11 Logical Operators

A list of the logical operators supported by B4A is shown in Table 6.3

Operator	Example
And	X And Y, returns TRUE if both x and y are TRUE
Or	X Or Y, return TRUE if either X or Y or both are TRUE
Not	Not(X), returns TRUE if X is FALSE

Table 6.3 Logical operators

6.12 Relational Operators

Relational operators are used to compare two variables in conditional statements. A list of the relational operators supported by B4A I shown in Table 6.4.

Operator	Example
=	Equal to, e.g. x = y
<>	Not equal to, e.g x <> y
>	Greater than, e.g. x > y
<	Less than, e.g. x < y
>=	Greater than or equal to, e.g. x >= y
<=	Less than or equal to, e.g. x <= y

Table 6.4 Relational operators

6.13 Changing the Program Flow

There are two types of statements that can be used to change the flow of control in a program: Conditional statements, and Iteration statements.

6.13.1 Conditional Statements

Conditional statements compare two or more variables and take some action depending upon the result of this comparison. In this section we shall be looking at the use of the conditional statements with some examples.

If-Then-Else-End If

The **If** statement is used to test a condition and take some action if the condition is True. Some examples are given below:

Single line comparison:

```
If x = 10 Then y = 0
```

Multiple line comparison (It is recommended to indent the code inside the **If-End If** block):

```
If x = 10 Then
   a = 1
   b = 2
```

```
    c = 3
End If
```

Using the **Else** keyword:

```
If x = 10 Then
    a = 1
Else
    a = 0
End If
```

Select-Case-End Select

The **Select-Case-End Select** statements are used when it is required to make comparisons with a number of different expressions. An example is given below. In this example, the value of variable **xyz** is compared with several numbers. If **xyz =1** then **a=1**, if **xyz = 3** then **a = 10,** if **xyz = 5** then **a = 20**, if **xyz = 7,9,** or **11** then **a = 100**, otherwise (i.e. xyz is not equal to any of the above) then a is cleared to 0:

```
Select xyz
    Case 1
        a = 1
    Case 3
        a = 10
    Case 5
        A = 20
    Case 7, 9, 11
        a = 100
    Case Else
        a = 0
End Select
```

6.13.2 Iterations

Several iteration (or loop) statements are supported by B4A.

For-Next

The For-Next statement is found nearly in all high level programming languages. Here, the code inside a loop formed by the **For-Next** statements is executed a number of times. In the following example the loop is executed 10 times. Variable **p** takes values from 1 to 10 in steps of 1. The code calculates the sum of numbers from 1 to 10:

```
Dim Sum As Int
Sum = 0
For p = 1 To 10
    Sum = Sum + p
Next
```

By default the loop step count is 1, but it can be changed to any value using the **Step** keyword:

```
For p = 1 To 10 Step 2
   'some code here
Next
```

The **Continue** statement can be used in a **For-Next** loop to skip iterations. In the following example the sum of numbers from 1 to 10 except number 5 is calculated:

```
Dim Sum as Int
Sum  = 0
For p = 1 To 10
     If p = 5 Then Continue
     Sum = Sum + p
Next
```

The Exit statement can be used to terminate a For-Next loop. In the following example the sum of numbers from 1 to 5 are calculated even though the loop is set to run from 1 to 10:

```
Dim Sum as Int
Sum  = 0
For p = 1 To 10
     Sum = Sum + p
     If p = 5 Then Exit
Next
```

Do-While-Loop

The **Do-While-Loop** statements can be used to establish a loop where the loop carries on until a condition becomes True. The following example calculates the sum of numbers from 1 to 10 using **Do-While-Loop** statements:

```
Dim Sum, i As Int
Sum = 0
i = 1
Do While i <= 10
     Sum = Sum + i
     i = i + 1
Loop
```

Care must be taken when using the **Do-While-Loop** statements to make sure that the loop condition changes inside the loop so that the loop terminates. As shown below, an infinite loop will be formed if the condition does not change inside the loop. In this example variable **i** is always equal to 1 and the loop never terminates:

```
Dim Sum, i As Int
```

```
Sum = 0
i = 1
Do While i <= 10
     Sum = Sum + i
Loop
```

Also, if the condition is not satisfied at the beginning of the **Do-While-Loop** statements then the loop will never execute. This is shown below where variable **i** is never greater than 1 and therefore the loop never executes:

```
i = 0
Do While i > 1
     'code here
Loop
```

Do-Until-Loop

The Do-Until-Loop statement is similar to the **Do-While**-**Loop** statement but here the loop is repeated until a condition becomes True. In the following example the loop repeats until variable **i** becomes 10, and it calculates the sum of numbers from 1 to 10:

```
Dim Sum, i As Int
Sum = 0
i = 0
Do Until i = 10
     i = i + 1
     Sum = Sum + i
Loop
```

The **Exit** keyword can be used in a **Do-While-Loop** and **Do-Until-Loop** statements as in a **For-Next** statement to terminate a loop.

For-Each-Next

This statement is similar to the **For-Next** statement, but here we can use any kind of object in the loop and not just numbers. In the example given below, **Fruits** is declared as an array of strings. When the program runs in Debug mode the following will be displayed in the debug window:

apple
orange
banana
pear

```
Dim MyFruit As String
Dim Fruits() As String
Fruits = Array As String("apple", "orange", "banana", "pear")
```

```
For Each Txt As String In Fruits
   MyFruit = Txt
   Log(MyFruit)
Next
```

6.14 Subroutines

Subroutines in B4A are declared with the keyword **Sub**. They have a name, possible arguments with their variable types, and the return type. After the body of the subroutine the keyword **End Sub** is used to terminate a subroutine. It is recommended to use comments to describe what the subroutine does. An example subroutine declaration is given below. This subroutine is given the name **Sum_Of_Numbers**, it returns an integer number, and it has one integer argument called **Num**. The subroutine calculates the sum of numbers from 1 to **Num** and returns the result to the calling program:

```
'
' This subroutine calculates the sum of numbers from 1 to Num
'
Sub Sum_Of_Numbers(Num As Int)As Int
     Dim i As Int
     Dim Sum As Int
     For i = 1 To Num
       Sum = Sum + i
     Next
   Return Sum
End Sub
```

The above subroutine can be called by specifying the value of **Num**. For example, to calculate the sum of numbers from 1 to 10 we can call the subroutine as follows where the value of variable **k** will be set to 55 after the call:

```
k = Sum_Of_Numbers(10)
```

6.15 Error Handling in Programs

Errors in a program could either be compile time errors or run time errors. Some of the compile time errors such as syntax errors can easily be detected by the compiler and corrected by the programmer. Run time errors can only be detected while the program is running.

The following few lines of code compiles with no errors but generates a run time error since the string in variable p is not numeric and therefore cannot be divided by a number:

```
Dim p As String
Dim i As Int
p="computer"
i=p/5
```

The run time error generated by the above code is shown in Figure 6.4. The error shows the line number where the error occurred in addition to the statement causing the error.

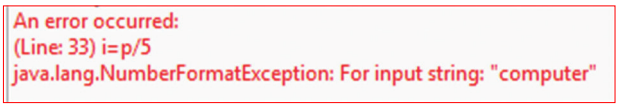


Figure 6.4 Run time error

Run time errors can be handled in B4A programs using the **Try-Catch-End Try** statements. Here, the keyword **Try** is used before the beginning of a block of code. If an error is detected during the run time inside this block, then block of code starting with the keyword **Catch** will be executed. As a result the error condition will be handled orderly inside the program. The **Try-Catch** is terminated with the keyword **End Try**. An example is given below where the run time error has been handled and the message A run time error has been detected is displayed by the MsgBox on the mobile device (see Figure 6.5).

```
Dim p As String
Dim i As Int
Try
   p="computer"
   i=p/5
Catch
   Msgbox("A run time error has been detected","")
End Try
```

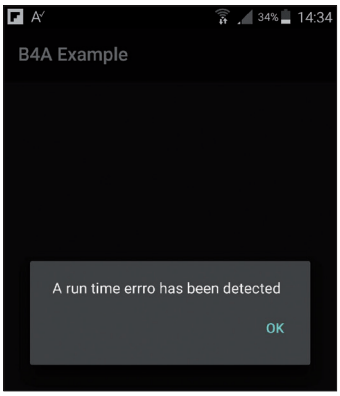


Figure 6.5 Handling the run time error

Details about the error can be obtained by using the statement **LastException.Message** as shown in the following code. The error generated on the mobile device is shown in Figure 6.6:

```
Dim p As String
Dim i As Int
Try
```

```
    p = "computer"
    i = p / 5
Catch
    Msgbox(LastException.Message, "Type of Error")
End Try
```

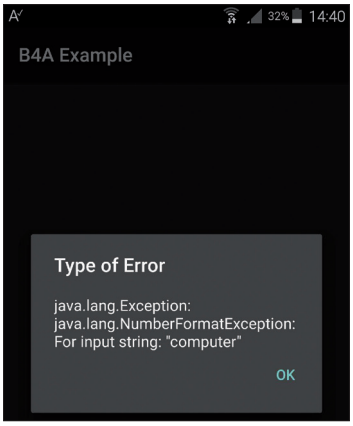


Figure 6.6 Run time error generated on the mobile device

6.16 Timer Events

Timers are used to generate events at specified intervals. A timer must be enabled before it can generate events. Timers must be declared in a **Sub Process_Globals** routine. In the following example, **MyTimer** is declared as a timer:

```
Sub Process_Globals
    Dim MyTimer as Timer
End Sub
```

Timer **MyTimer** can then be initialized and enabled as follows. In the following example the event name is chosen as **Tmr** and the timer is initialized to generate events at every 1000 ms (i.e. every second). The interval has data type **Long**:

```
MyTimer.initialize("Tmr", 1000)
MyTimer.Enabled = True
```

An example project using a timer will be given in the projects section of the book in the next Chapter.

6.17 Delays in Programs

It is sometimes necessary to insert delays in programs. This is achieved using the Sleep function. This function has one argument which specifies the amount of delay in milliseconds. In the following example the program delays for two seconds:

```
Sleep(2000)
```

Sleep(0) is a special case which allows other user interface codes to be handled.

6.18 Dialogs

Dialogs are used to communicate with the users. Basically a dialog displays a message and then expects the user to take some action. In B4A there are two types of dialogs: **Async** dialogs, and **Modal** dialogs.

Async Dialogs

Async dialogs are **InputListAsync**, I**nputMapAsync**, **MsgboxAsync** and **Msgbox2Async**.

MsgboxAsync displays a non-modal message box with a message, title, and OK button. The following code displays the message box shown in Figure 6.7:

```
MsgboxAsync("My name is John", "Name")
```

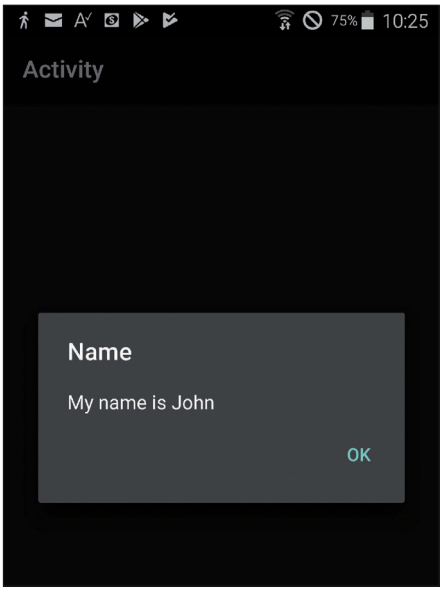


Figure 6.7 Example MsgboxAsync

Msgbox2Async This is similar to the standard MsgboxAsync but here the user is given three buttons with positive response, cancel, and negative response. In addition, an icon bitmap can eb displayed. The last parameter when set True enables the dialog box to be cancelled by clicking outside the box. Normally this parameter should be set False. The following code displays the message box shown in Figure 6.8:

```
Msgbox2Async("Is your age 25?", "Age", "Yes", "Cancel", "No", Null, True)
```

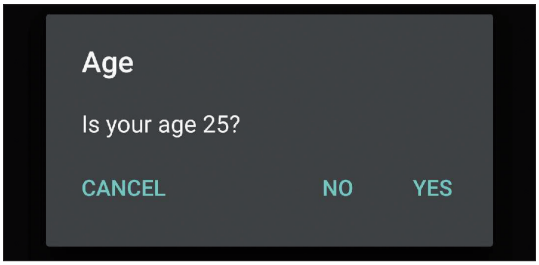


Figure 6.8 Example Msgbox2Async

We can disable a button by setting its value to a blank string, i.e. "". We can wait for the message box result and then check the result as shown in the following code:

```
Msgbox2Async("Is your age 25?", "Age", "Yes", "Cancel", "No", Null, True)
Wait For Msgbox_Result(Res as Int)
If Res = DialogResponse.POSITIVE Then
   MsgboxAsync("Your age is 25", "")
End If
```

Figure 6.9 shows the display on the mobile device when the user clicks button YES (i.e. POSITIVE response).

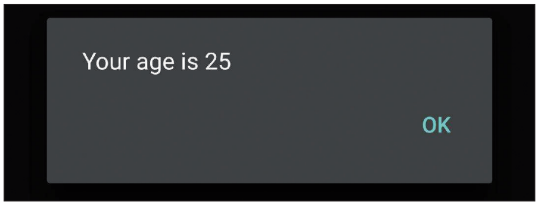


Figure 6.9 Display when YES is clicked

InputListAsync displays a non-modal list of items and radio buttons, where clicking on an item closes the dialog. In the following example code, fruit names Banana, Apple, Pear, and Cherry are displayed. The first parameter to inputListAsync is the list, the second parameter is the title, the third parameter is the index number of the first selected item (index number starts from 0), and the last parameter if set True enables the dialog to be cancelled by clicking outside the dialog:

```
Dim Fruits As List
Fruits = Array("Banana", "Apple", "Pear", "Cherry")
InputListAsync(Fruits,"Pick a fruit", 1, False)
```

Figure 6.10 shows the display on the mobile device.

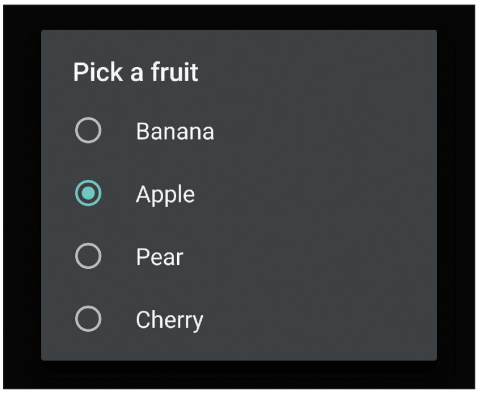


Figure 6.10 Example InputListAsync

We can wait for a choice to be made as shown in the following code. Here, variable Res stores the index number of the selected item. In this example, Pear was selected and Figure 6.11 shows the display on the mobile device:

```
Dim Fruits As List
Dim Selected As String
Fruits = Array("Banana", "Apple", "Pear", "Cherry")
InputListAsync(Fruits,"Pick a fruit", 0, False)
Wait for InputList_Result(Res As Int)
Selected = Fruits.Get(Res)
MsgboxAsync(Selected,"")
```

Figure 6.11 Pear was selected

InputMapAsync is similar to the InputListAsync but here a number of checkboxes are displayed and the user can make more than one selection. In the following code, fruits Banana, Apple, Pear, and Cherry are displayed and Apple and Cherry are ticked (True). Figure 6.12 shows the display on the mobile device:

```
Dim Fruits As Map
Fruits = CreateMap("Banana":False, "Apple":True, "Pear":False, "Cherry":True)
InputMapAsync(Fruits,"Pick a fruit", False)
```

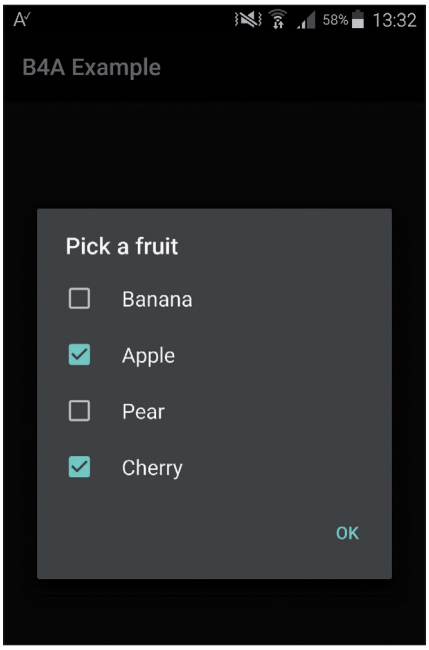


Figure 6.12 Example InputMapAsync

We can wait for user input as shown in the following code. Figure 6.13 shows the display on the mobile device when the user selects Apple and Pear:

```
Dim Fruits As Map
Fruits = CreateMap("Banana":False, "Apple":True, "Pear":False, "Cherry":True)
InputMapAsync(Fruits,"Pick fruits", False)
Wait for inputMap_Result
Msgbox(Fruits,"")
```

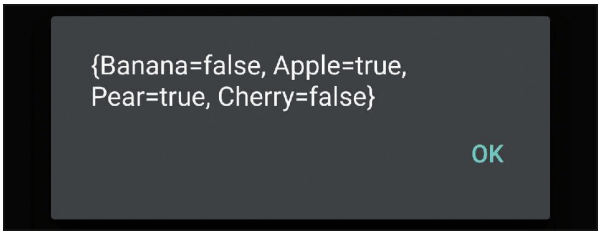


Figure 6.13 User selected Apple and Pear

Modal Dialogs

The use of modal dialogs is not recommended even though they are supported by the B4A. These dialogs have been replaced by the async dialogs, although they are still available. Since the modal dialogs can cause instability and can lead to program crashes, they are not discussed in this book any more. Interested readers can get further information on modal dialogs from the Internet.

6.19 Libraries

One of the strong points of developing an application with the B4A is that a large number of libraries are supported by the B4A. Several different libraries are available.

The Core Library is included both in the Trial and the Full versions of B4A. A list of functions in the Core library and their details can be found at the following link:

https://www.b4x.com/android/help/core.html

In addition to the Core Library, a Standard Library and additional official and user libraries are available. The Standard library is only available in the Full version of B4A. More information on the Standard and the Additional libraries can be obtained from the B4A web site:

https://www.b4x.com

Before using a library it may be necessary to add include the library in your program. This can be done using the B4A **Libraries Manager** which is accessed by clicking the Libraries Manager tab at the bottom right hand side of the screen. Figure 6.14 shows the Libraries Manager:

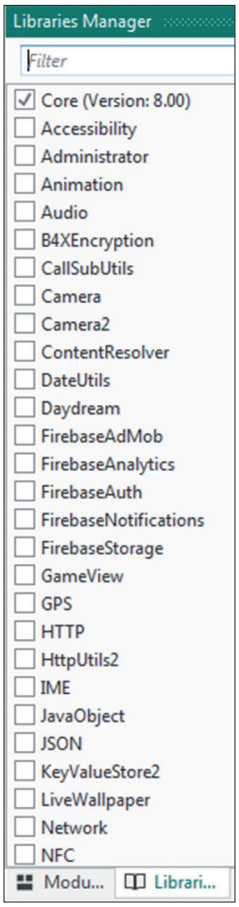


Figure 6.14 Libraries Manager

Notice that the Core Library is always selected. You should include the libraries that you wish to use in your program.

6.20 Summary

This Chapter was about the B4A language reference where we have learned how to write programs using the B4A. Topics such as variables, arrays, conditional statements, error handling, subroutines, libraries, delays and dialogs have all been explained in this Chapter. In the next Chapter we shall be developing projects with the B4A.

Chapter 7 • Mobile device only simple projects

7.1 Overview

In this Chapter we shall be developing several simple projects using the B4A and the mobile device only (e.g. an Android mobile phone). It is hoped that the readers will learn how to use the various features of the B4A during the development of these projects. The projects will be based entirely on using the B4A on a mobile device.

The following headings will be given for each project:

- Project title
- Description of the project
- Aim of the project
- Complete program listing
- Description of the program
- Suggestions for more work (if applicable)

7.2 PROJECT 1 – Digital Chronometer

7.2.1 Description

In this project a digital chronometer is designed. Three push-buttons, named START, STOP, and CLEAR are used. The chronometer starts and stops when the START and STOP buttons are clicked respectively. Clicking the CLEAR button clears the count so that the chronometer is ready for a new count. The count is displayed in seconds.

7.2.2 Aim

The aim of this project is to show how the Timer module of the B4A can be used to design a digital chronometer.

7.2.3 Program Listing

In this program a Label is used to display the time, and three buttons are used to control the chronograph. The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white

- Click **Add View** and add a new Label
- Change the name of the label to **SECONDS**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 20
- Click **Add View** and add 3 Buttons to the bottom of the label. Name these buttons as **START**, **STOP**, and **CLEAR** and set their **Texts** to **START**, **STOP** and **CLEAR** respectively.
- Set the text colours of the buttons **START**, **STOP** and **CLEAR** to green, red, and yellow respectively.
- Figure 7.1 shows the screen layout on the mobile device.

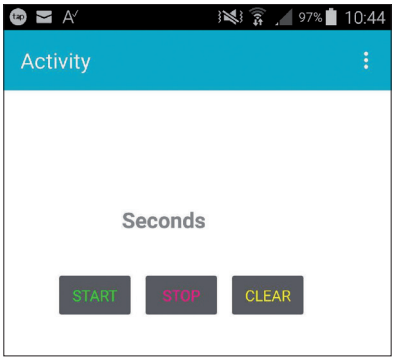


Figure 7.1 Screen layout on the mobile device

- We now have to generate code for the items on the screen.
  Click **Tools** -> **Generate Members** and click **Select All Views**
- Click **START**, **STOP**, and **CLEAR** and set to **Click** so that the buttons respond when clicked (see Figure 7.2)
- Click **Generate Members** and close the screen

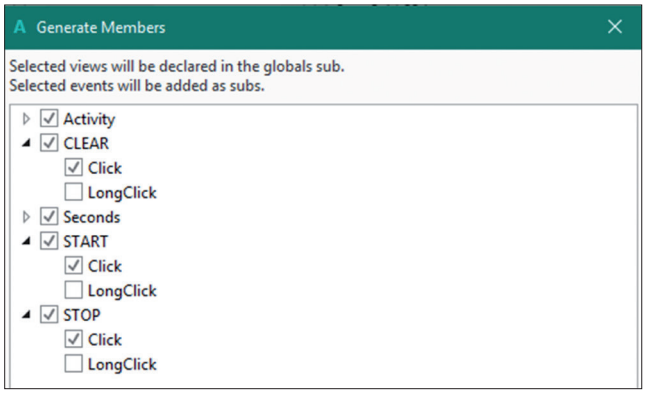


Figure 7.2 Set buttons to Click

- You should see the following code generated in the program section:

```
Sub Globals
    Private CLEAR As Button
    Private SECONDS As Label
    Private START As Button
    Private STOP As Button
End Sub

Sub STOP_Click
End Sub

Sub START_Click
End Sub

Sub CLEAR_Click
End Sub
```

- Now, save the designer screen with the name, e.g. **Chronograph**

- We now have to fill in the code. First of all, we need to set up a timer which is activated every second so that we can start our chronograph. Insert the following code in the **Sub Process_Globals** routine. Notice that we are also initializeing a variable called **Count** that will be used as a seconds counter:

```
Dim Tmr as Timer
Public Count = 0 as Int
```

- Now, initialize the timer to become active every second. Insert the following code into the **Activity_Create** routine. Here, the timer routine is called MyTimer and the timer is set to activate every 1000ms. Notice that the timer has not been enabled at this stage. We are also displaying the time as 0:

```
If FirstTime = True Then
Tmr.Initialize("MyTimer", 1000)
Seconds.Text = Count
    End If
```

- Set the layout name to **Chronograph**:

```
Activity.LoadLayout("Chronograph")
```

- Enter the following statements inside the generated subroutines to disable, enable, and clear the timer and also variable **Count**:

```
Sub STOP_Click
    Tmr.Enabled = False
End Sub

Sub START_Click
    Tmr.Enabled = True
End Sub

Sub CLEAR_Click
    Tmr.Enabled = False
    Count = 0
    SECONDS.Text = 0
End Sub
```

- The timer routine should be in a subroutine called **MyTimer** as declared in the **Activity_Create**. The keyword **Tick** must be added to the end of this subroutine name:

```
Sub MyTimer_Tick
    Count = Count + 1
    SECONDS.Text = Count
End Sub
```

You should now run the program. Clicking the **START** button should start the seconds count. Figure 7.3 shows a typical output on the mobile device. Notice that the heading **Activity** is displayed by default at the top of the screen

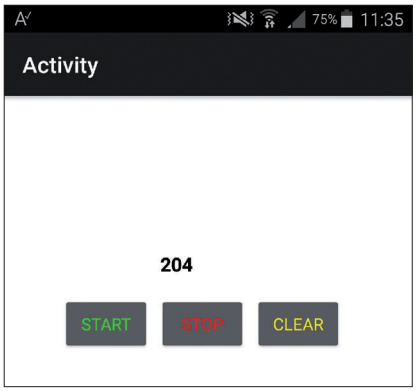


Figure 7.3 Output on the mobile device

The complete program listing is shown in Figure 7.4 (program: CHRONOGRAPH).

```
#Region  Project Attributes
#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
  Dim Tmr As Timer
  Public Count = 0 As Int
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private CLEAR As Button
  Private SECONDS As Label
  Private START As Button
  Private STOP As Button
End Sub

Sub Activity_Create(FirstTime As Boolean)
  'Do not forget to load the layout file created with the visual designer.
For example:
  Activity.LoadLayout("Chronograph")
  If FirstTime = True Then
    Tmr.Initialize("MyTimer", 1000)
    SECONDS.Text = Count
  End If
End Sub
```

```
Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub


Sub STOP_Click
 Tmr.Enabled = False
End Sub

Sub START_Click
 Tmr.Enabled = True
End Sub

Sub CLEAR_Click
 Tmr.Enabled = False
 Count = 0
 SECONDS.Text = 0
End Sub

Sub MyTimer_Tick
 Count = Count + 1
 SECONDS.Text = Count
End Sub
```

Figure 7.4 Program listing

Making the Project Visually Appealing

The project can be made more visually appealing by modifying the program given in Figure 7.4. For example, the **Status Bar** at the top of the screen can be hidden by setting the keyword **#FullScreen** to True in the Activity Attributes at the beginning of the program:

```
#FullScreen: True
```

By default the **Title Bar** is displayed as **Activity**. This can be enabled or disabled by the keyword **#IncludeTitle** in the **Activity Attributes** at the beginning of the program:

```
#IncludeTitle: True
```

The Title Bar can be changed by the statement **Activity.Title**. In the following code the Title Bar is set to **CHRONOGRAPH**:

```
Activity.Title = "CHRONOGRAPH
```

Figure 7.5 shows the mobile device screen when the Status Bar is hidden and the Title is

set to **CHRONOGRAPH**. In this modified program a label is added to show that the counter is in seconds.

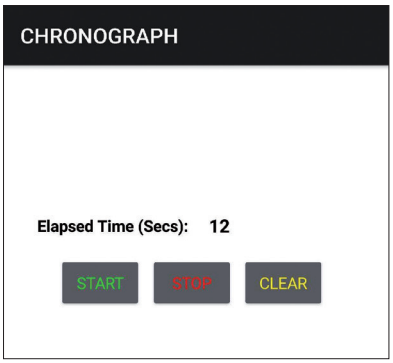


Figure 7.5 Mobile device screen with no Status Bar

The modified program (program: CHRONOGRAPH2) listing is shown in Figure 7.6. Notice that the layout **Chronograph2** is displayed by the program, where this layout includes the additional label.

```
#Region  Project Attributes

#Region  Activity Attributes
  #FullScreen: True
  #IncludeTitle: True
#End Region

Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
  Dim Tmr As Timer
  Public Count = 0 As Int
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private CLEAR As Button
  Private SECONDS As Label
  Private START As Button
  Private STOP As Button
End Sub

Sub Activity_Create(FirstTime As Boolean)
  'Do not forget to load the layout file created with the visual designer.
```

```
For example:
 Activity.LoadLayout("Chronograph2")
 Activity.Title = "CHRONOGRAPH"
 If FirstTime = True Then
   Tmr.Initialize("MyTimer", 1000)
   SECONDS.Text = Count
 End If
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub


Sub STOP_Click
 Tmr.Enabled = False
End Sub

Sub START_Click
 Tmr.Enabled = True
End Sub

Sub CLEAR_Click
 Tmr.Enabled = False
 Count = 0
 SECONDS.Text = 0
End Sub

Sub MyTimer_Tick
 Count = Count + 1
 SECONDS.Text = Count
End Sub
```

Figure 7.6 Modified program

7.3 PROJECT 2 – Dice

7.3.1 Description

This is a dice project. In this project, a push button named **START** and two labels named **Dice1** and **Dice2** are used. When the button is pressed two random numbers are generated between 1 and 6 and are displayed on the two labels as dice numbers. After 3 seconds the labels are cleared and the program is ready to generate new dice numbers.

7.3.2 Aim

The aim of this project is to show how the random number generator can be used in a project.

7.3.3 Program Listing

In this program two labels are used to display the dice numbers, and a button is used to start the game. The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click **START**
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Click **Add View** and add a new Label
- Change the **Name** and **Text** of the label to **Dice1**
- Set thetextl **Color** to black, its **Style** to BOLD, and its **Size** to 20
- Click **Add View** and add another label to the design.
- Change the **Name** and **Text** of this label to **Dice2**
- Set the text **Color** to black, its **Style** to BOLD, and its **Size** to 20
- Click **Add View** and add a button to the bottom of the labels. Name this button as **START** and set its **Text** to **START**
- Set the text **Color** of the button to green, its **Style** to BOLD and **Size** to 20
- Figure 7.7 shows the screen layout on the mobile device.

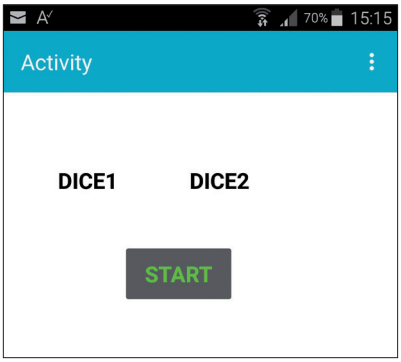


Figure 7.7 Screen layout on the mobile device

We now have to generate code for the items on the screen. Click **Tools** -> **Generate Members** and click **Select All Views**

- Click **START** and set to **Click** so that the button respond when clicked (see Figure 7.8)
- Click **Generate Members** and close the screen
- Save the designer screen with the name **DICE**.

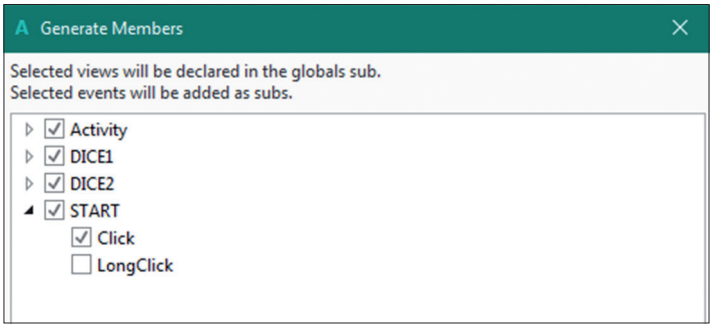


Figure 7.8 Set button to Click

- You should see the following code generated in the program section:

```
Sub Globals
    Private Dice1 As Label
    Private Dice2 As Label
    Private START As Button
End Sub

Sub START_Click
End Sub
```

- We now have to fill in the code. Declare two integer variable Dice1No and Dice2No in **Process_Globals**:

```
Public Dice1No, Dice2No As Int
```

- Insert the following code inside **Activity_Create**. Notice that the Designer screen was saved with the name **DICE**. The message **READY** is displayed waiting for the user to click the **START** button:

```
Activity.LoadLayout("DICE")
    Activity.Title = "DICE"
    Dice1.Text = "READY..."
    Dice2.Text = ""
```

- Insert the following code inside **START_Click**. Two random numbers are generated between 1 and 6 and are stored in variables **Dice1No** and **Dice2No** respectively. Function Rnd (m, n) generates random numbers between m and n-1 inclusive. The dice numbers are then displayed by the two labels **Dice1** and **Dice2** respectively. The program then waits for 3 seconds (3000ms). After this time the program is ready to generate new dice numbers. The message **READY** is displayed at label **Dice1** position and label **Dice2** is cleared. Clicking the **START** button generates two new dice numbers:

```
Dice1No = Rnd(1, 7)
    Dice2No = Rnd(1, 7)
    Dice1.Text = Dice1No
    Dice2.Text = Dice2No
    Sleep(3000)
    Dice1.Text = "READY..."
Dice2.Text = ""
```

- Save the program with the name **DICE**, and click **Run** to start the program. Figure 7.9 and Figure 7.10 show displays on the mobile device. The program listing is shown in Figure 7.11. Notice that the screen is given the title **DICE**.

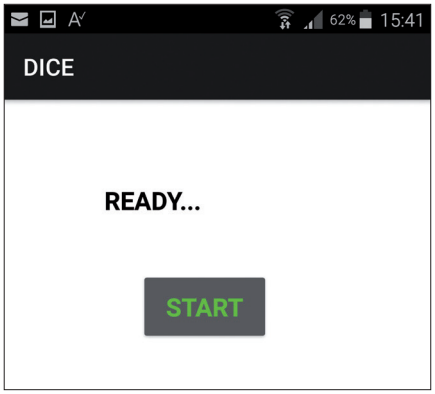


Figure 7.9 READY display on the mobile device

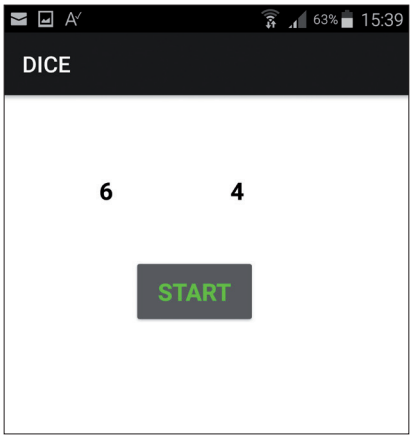


Figure 7.10 Dice numbers on the mobile device

```
#Region  Project Attributes
#Region  Activity Attributes

Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
  Public Dice1No, Dice2No As Int
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.

  Private Dice1 As Label
  Private Dice2 As Label
  Private START As Button
```

```
End Sub

Sub Activity_Create(FirstTime As Boolean)
  'Do not forget to load the layout file created with the visual designer.
For example:
  Activity.LoadLayout("DICE")
  Activity.Title = "DICE"
  Dice1.Text = "READY..."
  Dice2.Text = ""

End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub


Sub START_Click
  Dice1No = Rnd(1, 7)
  Dice2No = Rnd(1, 7)
  Dice1.Text = Dice1No
  Dice2.Text = Dice2No
  Sleep(3000)
  Dice1.Text = "READY..."
  Dice2.Text = ""
End Sub
```

Figure 7.11 Program listing

7.4 PROJECT 3 – Euro Millions Lottery Numbers

7.4.1 Description

This is a Euro Millionaire Lottery Numbers project. In the Euro Millions Lottery the user enters numbers in two sections: The **Lottery Numbers** (or **Euro Millions Numbers**) section and the **Lucky Stars** section. In the **Lottery Numbers** section there are numbers from 1 to 50 and the user is required to tick only 5 numbers. In the **Lucky Stars** section there are numbers from 1 to 12 and the user is required to tick only 2 numbers.

In this project 5 unique random **Lottery Numbers** are generated between 1 and 50 inclusive, and also 2 unique random **Lucky Stars** numbers are generated between 1 and 12 inclusive. Clicking a button named **START** generates a set of numbers for the game. After 15 seconds of delay the numbers are cleared and the program is ready to generate new set of numbers.

7.4.2 Aim

The aim of this project is to show how the random number generator can be used in a project to generate the Euro Millionaire lottery numbers.

7.4.3 Program Listing

In this program the actual lottery numbers are generated on 5 labels. In addition, 2 labels are used to display the Luck Stars numbers. Another two labels are used as comments on the screen. The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click **START**
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Click **Add View** and add a new Label
- Change the **Text** of this label to **Euro Millionaire Numbers:**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14
- Click **Add View** and add another label to the right hand side of the previous label.
- Change the **Name** of this label to **Numbers**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14
- Click **Add View** and add a new label
- Change the **Text** of this label to **Lucky Stars:**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14
- Click **Add View** and add a new label to the right hand side of the previous label
- Change the **Name** of this label to **LuckyStars**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14

- Click Add View and add a button at the bottom of the labels. Name this button as **START** and set its **Text** to **START**

- Set the **Text Color** of the button to green, its **Style** to BOLD and **Size** to 20

- Figure 7.12 shows the screen layout on the designer screen on the PC. The layout on the mobile device is shown in Figure 7.13.

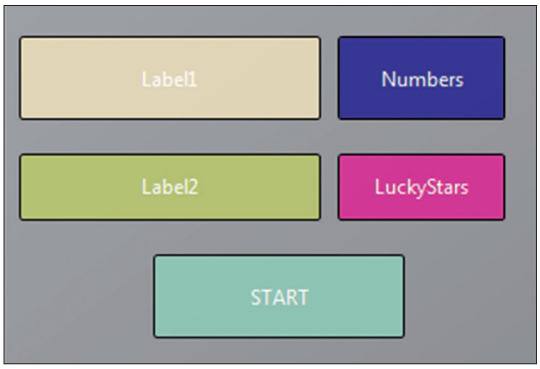


Figure 7.12 The designer screen

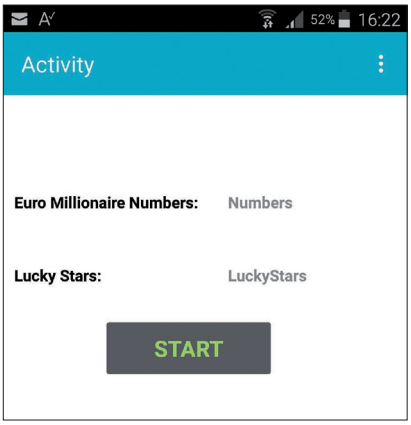


Figure 7.13 Layout on the mobile device

We now have to generate code for the items on the screen. Click **Tools** -> **Generate Members** and click **Select All Views**

- Click **START** and set to **Click** so that the button respond when clicked (see Figure 7.14)

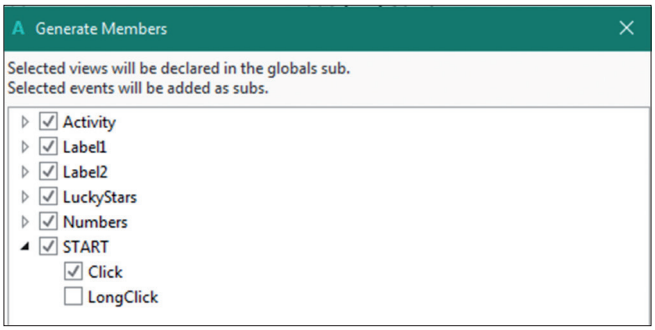


Figure 7.14 Set button to Click

- Click **Generate Members** and close the screen
- Save the designer screen with the name **LOTTERY**.
- You should see the following code generated in the program section:

```
Sub Globals
    Private Label1 As Label
    Private Label2 As Label
    Private LuckyStars As Label
    Private Numbers As Label
    Private START As Button
End Sub
Sub START_Click

End Sub
```

- We now have to fill in the code. Declare an integer array called **LotteryNumbers** with 6 elements (array index is started from 1 and not 0) and another integer array called **LuckyStarNumbers** with 3 elements (array index is started from 1 and not 0) in **Process_Globals**:

```
Public LotteryNumbers(6) As Int
Public LuckyStarNumbers(3) as Int
```

- Insert the following code inside **Activity_Create**. Notice that the Designer screen was saved with the name **LOTTERY**:

```
Activity.LoadLayout("LOTTERY")
    Activity.Title = "Euro Millions Lottery Numbers"
```

- Create a new subroutine called **Generate_Numbers**. This subroutine will have 2 integer arguments named **Cnt**, **Max_No**, and an array named **LNumbers**. **Cnt** is the count of the numbers to be generated. For the lottery numbers, Cnt = 5,

and for the Lucky Stars, Cnt = 2. **Max_No** is the maximum number to be generated. For the lottery numbers, Max_No = 50, and for the Lucky Stars, Max_No = 12. Array **LNumbers** stores the generated random numbers. Subroutine **Generate_Numbers** generates unique random numbers from 1 to **Max_No** inclusive. The generated numbers are checked with the existing numbers to ensure that there is no duplication. Variable **flag** is initially cleared to 0 and is then set to 1 if a duplicate number is found. When this happens a new number is generated and checked again until all the generated numbers are unique.

- Inside **START_Click** the arrays that store the lottery numbers and the Luck Stars numbers are cleared to 0 to start with. Then subroutine **Generate_Numbers** is called to generate lottery numbers and Luck Stars numbers. A variable called **All_Numbers** is used to append the generated numbers with spaces between them. The lottery numbers are displayed by label **Numbers**, and the Lucky Stars numbers are displayed by label **LucyStars**. The generated numbers are cleared after 15 seconds. At this point the program can generate new set of numbers after the **START** button is clicked.

Figure 7.15 shows a typical run on the mobile device. The complete program listing (program: LOTTERY) of the project is shown in Figure 7.16.

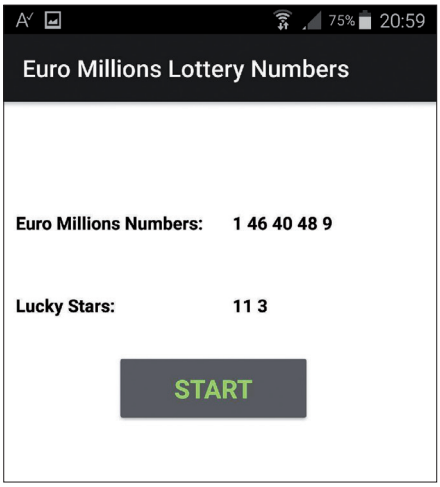


Figure 7.15 Typical run on the mobile device

```
#Region  Project Attributes
#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
  Public LotteryNumbers(6) As Int
```

```
 Public LuckyStarNumbers(3) As Int
End Sub

Sub Globals
 'These global variables will be redeclared each time the activity is
created.
 'These variables can only be accessed from this module.
 Private Label1 As Label
 Private Label2 As Label
 Private LuckyStars As Label
 Private Numbers As Label
 Private START As Button
End Sub

Sub Activity_Create(FirstTime As Boolean)
 'Do not forget to load the layout file created with the visual designer.
For example:
 Activity.LoadLayout("LOTTERY")
 Activity.Title = "Euro Millions Lottery Numbers"
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

'
' This subroutine has 3 arguments: Cnt is the count of the numbers. For
normal
' Lottery Numbers, Cnt = 5 and for the Lucky Stars the Cnt is 2. Max_No is
the
' maximum number to be generated. For Lottery Numbers, Cnt = 50, and for the
' Lucky Stars the Cnt is 12. LNumbers is an array which stores the generated
' numbers
Sub Generate_Numbers(Cnt As Int, Max_No As Int, LNumbers () As Int)
 Dim i, n As Int
 Dim flag, j As Int

 For j = 1 To Cnt
   flag = 1
   Do While flag = 1
     n = Rnd(1, Max_No)
     flag = 0
     For i = 1 To Cnt
       If n = LNumbers(i) Then
```

```
          flag = 1
        End If
      Next
    Loop
    LNumbers(j) = n
  Next
End Sub

Sub START_Click
  Dim i As Int
  Dim LotteryNumberCount, LuckyStarNumberCount As Int
  Dim MaxLotteryNumber, MaxLuckyStarNumber As Int
  Dim All_Numbers As String

  LotteryNumberCount = 5: LuckyStarNumberCount = 2
  MaxLotteryNumber = 50:  MaxLuckyStarNumber = 12
'
' Clear arrays LotteryNumbers and LuckyStarNumbers
'
  For i = 1 To LotteryNumberCount
    LotteryNumbers(i) = 0
  Next
  LuckyStarNumbers(1) = 0: LuckyStarNumbers(2) = 0
'
' Generate Lottery Numbers and Lucky Star Numbers
'
  Generate_Numbers(LotteryNumberCount, MaxLotteryNumber+1, LotteryNumbers)
  Generate_Numbers(LuckyStarNumberCount, MaxLuckyStarNumber+1,
LuckyStarNumbers)
'
' Combine the generated numbers into an array called All_Numbers and then
' display the numebrs with labels Numbers and LuckyStars
'
  All_Numbers = ""
  For i = 1 To LotteryNumberCount
    All_Numbers = All_Numbers & LotteryNumbers(i) & " "
  Next
  Numbers.Text = All_Numbers

  All_Numbers = ""
  For i = 1 To LuckyStarNumberCount
    All_Numbers = All_Numbers & LuckyStarNumbers(i) & " "
  Next
  LuckyStars.Text = All_Numbers

'
```

```
' Sleep for 15 seconds and then clear the numbers, ready to generate new
' numbers when button START is clicked
'
  Sleep(15000)
  Numbers.Text = ""
  LuckyStars.Text = ""

End Sub
```

Figure 7.16 Program listing

7.5 PROJECT 4 – Geography Lesson

7.5.1 Description

This is a simple geography lesson project where the user learns the capital cities of some countries. The user selects a country from a list and the capital city of the selected country is displayed. The display is cleared after 10 seconds so that the user can select another country.

7.5.2 Aim

The aim of this project is to show how to use **ListView** in a project. The country names are listed in a ListView the capital city of the selected country is shown in a label.

7.5.3 Program Listing

The following countries are used in this project: France, Spain, Turkey, Greece, Italy, Cyprus, and Portugal.

The screen layout is shown in Figure 7.17. The activity is given the title **Capital Cities of Countries**. The screen consists of a heading (name: **Label1**) with the title **Select a Country:** , a ListView (name: **Countries**) with the country names, and three labels at the bottom part of the screen with names **SelectedCountry**, **Capital**, and **CapitalCity**. For example, if France is selected, Label **SelectedCountry** displays the text **France's**, label **Capital** displays the fixed text **Capital city is:**, and label **CapitalCity** displays the text **Paris**.

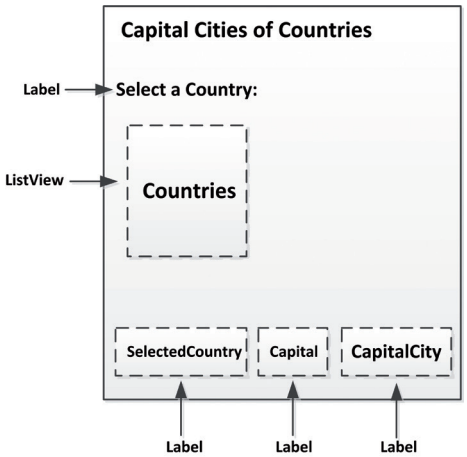


Figure 7.17 Screen layout

The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click **START**
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Click **Add View** and add a new Label
- Change the **Text** of this label to **Select a Country:**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14
- Click **Add View** and add a **ListView** to the bottom of the label
- Change the name of the ListView to Countries
- Click Add View and add 3 labels at the bottom of the screen. Set the names of these labels to **SelectedCountry**, **Capital**, and **CapitalCity**
- Set the **Text Colors** of these labels to black, and their **Styles** to BOLD. Set the **Sizes** of the first two labels to 14 and the **Size** of the **CapitalCity** to 20

- Figure 7.18 shows the screen layout on the designer screen on the PC

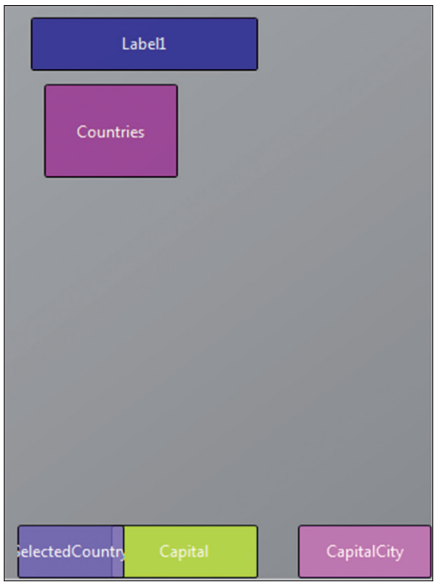


Figure 7.18 The designer screen

We now have to generate code for the items on the screen. Click **Tools** -> **Generate Members** and click **Select All Views**

- Click **Countries** and set to **Item Click** so that the **ListView** respond when clicked (see Figure 7.19)

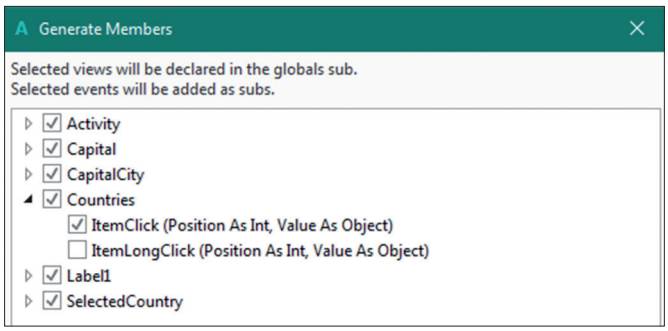


Figure 7.19 Set Countries to Item Click

- Click **Generate Members** and close the screen
- Save the designer screen with the name **COUNTRIES**
- You should see the following code generated in the program section:

```
Sub Globals
    Private CapitalCity As Label
    Private Countries As ListView
    Private Label1 As Label
    Private Capital As Label
    Private SelectedCountry As Label
End Sub

Sub Countries_ItemClick (Position As Int, Value As Object)
End Sub
```

- We now have to fill in the code. Insert the following statement to Sub Globals:

```
Dim CountryNames as ListView
```

- Inside the **Activity_Create,** load screen **COUNTRIES** and insert the following statement to give a title to the activity:

```
Activity.LoadLayout("COUNTRIES")
Activity.Title = "Capital Cities of Countries"
```

- We now have to create a **ListView** and insert our country names once at the beginning of the program (i.e. in section **Activity_Create**). Insert the following code to the **Activity_Create**:

```
CountryNames.Initialize("Countries")
Activity.Addview(CountryNames,10%x,10%y,35%x,63%y)
CountryNames.AddSingleLine2("France","France")
CountryNames.AddSingleLine2("Spain","Spain")
CountryNames.AddSingleLine2("Turkey","Turkey")
CountryNames.AddSingleLine2("Greece","Greece")
CountryNames.AddSingleLine2("Italy","Italy")
CountryNames.AddSingleLine2("Cyprus","Cyprus")
CountryNames.AddSingleLine2("Portugal","Portugal")
CountryNames.Color=Colors.gray

SelectedCountry.Visible = False
Capital.Visible = False
CapitalCity.Visible = False
```

- Notice in the above code **Activity_Addview** defines the co-ordinates and size of our **ListView**. The arguments of this function are the left and top co-ordinates, width, and height. These are set to 10, 10, 35, and 63 after a little trial. Function **AddSingleLine2** is used to add the country names to the **ListView**. There are basically two functions that can be used to add single items to a **ListView**. Function **AddSignleLine** just adds an item, whereas function **AddSingleLine2**

requires two arguments: the first argument is the item to be displayed and the second item is the return value from the **ListView** when function **GetItem** is called, or this is the **Value** returned in subroutine **ItemClick** (see above). The colour of the **ListView** is set to grey in this example. Notice also that in the above code the three labels at the bottom of the screen are hidden so that at the beginning of the program we only see the heading and the country names.

- Subroutine **Countries_ItemClick** is called when the user clicks on an item in the **ListView**. Here, as mentioned above, argument **Value** returns the second argument of the selected item when function **AddSingleLine2** was used to declare the items. For example, if country **France** is clicked in the **ListView**, argument **Value** will be set to **France**. The code inside subroutine **Countries_ItemClick** is as follows:

```
Sub Countries_ItemClick (Position As Int, Value As Object)
    SelectedCountry.Visible = True
    CapitalCity.Visible = True
    Capital.Visible = True
    SelectedCountry.Text = Value & "'s"
    Select Value
      Case "France"
        CapitalCity.Text = "Paris"
      Case "Spain"
        CapitalCity.Text = "Madrid"
      Case "Turkey"
        CapitalCity.Text = "Ankara"
      Case "Greece"
        CapitalCity.Text = "Athens"
      Case "Italy"
        CapitalCity.Text = "Rome"
      Case "Cyprus"
        CapitalCity.Text = "Nicosia"
      Case "Portugal"
        CapitalCity.Text = "Lisbon"
    End Select
    Sleep(10000)
    CapitalCity.Visible = False
    SelectedCountry.Visible = False
    Capital.Visible = False
End Sub
```

- In the above code the three labels at the bottom of the screen are first enabled. Then, a Select-Case statement block is used to check the return Value. Label CapitalCity is then set to display the capital city of the selected item. For example, if France is selected the display will show:

```
France's Capital city is: Paris
```

- After displaying the capital city of the selected item, the program waits for 10 seconds and then disables the labels at the bottom of the screen and the program is ready to accept a new country name.

- Figure 7.20 and Figure 7.21 shows example displays on the mobile device when the program is run.

Figure 7.20 Example display before a country is selected

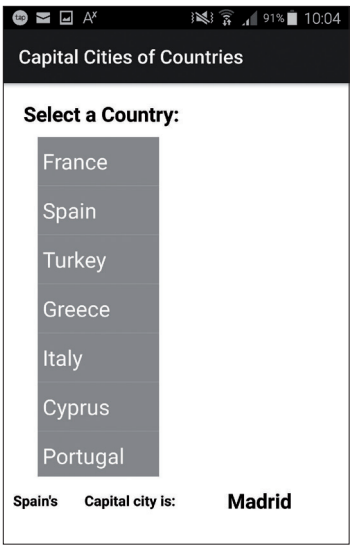


Figure 7.21 Example display after a country is selected

- Figure 7.22 shows the complete program listing (program: COUNTRIES)

```
#Region  Project Attributes
#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private CapitalCity As Label
  Private Countries As ListView
  Private Label1 As Label
  Private Capital As Label
  Private SelectedCountry As Label
  Dim CountryNames As ListView
End Sub

'
' Create a list of countries. The first parameter of AddSingleLine2 is
displayed in
' ListView, while the second parameetr is returned
'
Sub Activity_Create(FirstTime As Boolean)
  'Do not forget to load the layout file created with the visual designer.
For example:
  Activity.LoadLayout("COUNTRIES")
  Activity.Title = "Capital Cities of Countries"
  CountryNames.Initialize("Countries")
  Activity.Addview(CountryNames,10%x,10%y,35%x,63%y)   'left,top,width,height
  CountryNames.AddSingleLine2("France","France")
  CountryNames.AddSingleLine2("Spain","Spain")
  CountryNames.AddSingleLine2("Turkey","Turkey")
  CountryNames.AddSingleLine2("Greece","Greece")
  CountryNames.AddSingleLine2("Italy","Italy")
  CountryNames.AddSingleLine2("Cyprus","Cyprus")
  CountryNames.AddSingleLine2("Portugal","Portugal")
  CountryNames.Color=Colors.gray

  SelectedCountry.Visible = False
  Capital.Visible = False
```

```
  CapitalCity.Visible = False
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

'
' Return the clicked country and display its capital city. The display is
cleared
' after 10 seconds so that another selection can be made
'
Sub Countries_ItemClick (Position As Int, Value As Object)
  SelectedCountry.Visible = True
  CapitalCity.Visible = True
  Capital.Visible = True
  SelectedCountry.Text = Value & "'s"
  Select Value
    Case "France"
      CapitalCity.Text = "Paris"
    Case "Spain"
      CapitalCity.Text = "Madrid"
    Case "Turkey"
      CapitalCity.Text = "Ankara"
    Case "Greece"
      CapitalCity.Text = "Athens"
    Case "Italy"
      CapitalCity.Text = "Rome"
    Case "Cyprus"
      CapitalCity.Text = "Nicosia"
    Case "Portugal"
      CapitalCity.Text = "Lisbon"
  End Select
  Sleep(10000)
  CapitalCity.Visible = False
  SelectedCountry.Visible = False
  Capital.Visible = False
End Sub
```

Figure 7.22 Program listing

7.6 PROJECT 5 – Primary School Mathematics

7.6.1 Description

This is a simple project which can help pupils at the primary schools to improve their

mathematics skills. Two random numbers are generated between 1 and 100. Additionally, a random mathematical operation (e.g. addition, subtraction, multiplication, or division) is generated. The user is given 10 seconds to calculate the result of the mathematical calculation. At the end of 10 seconds the correct result is displayed so that the user can compare with his/her own result. The screen is cleared after 10 seconds and a new question is displayed. A typical display could be as follows:

```
25 X 2 = 50
```

7.6.2 Aim

The aim of this project is to show how to use the random number generator, and also how to use the various string handling operations in B4A programs.

7.6.3 Program Listing

The screen layout is shown in Figure 7.23. A label is used to display a heading called **Mathematical Calculations**. A label called **Operation** is used to display the question. Another label called **Answer** is used to display the result of the required calculation.

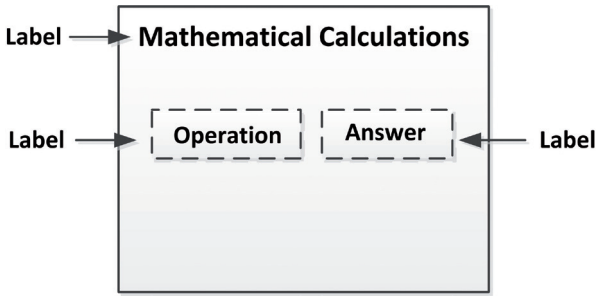


Figure 7.23 Screen layout

The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click **START**
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Click **Add View** and add a new Label

- Change the **Text** of this label to **Mathematical Calculations**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 20
- Click **Add View** and add another label under the first label. Name this label as **Operation**. Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 20
- Click **Add View** and add another label next to the second one. Name this label as **Answer**. Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 20
- Figure 7.24 shows the screen layout on the designer screen on the PC

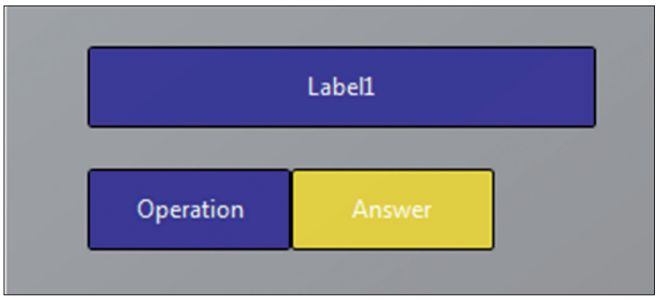


Figure 7.24 The designer screen

- We now have to generate code for the items on the screen. Click **Tools** -> **Generate Members** and click **Select All Views**
- Click **Generate Members** and close the screen
- Save the designer screen with the name **MATHEMATICS**
- You should see the following code generated in the program section:

```
Sub Globals
    Private Answer As Label
    Private Label1 As Label
    Private Operation As Label
End Sub
```

- Now we have to write the main program code inside **Activity_Create**. Name the activity title as **Elementary Mathematics** and load the **MATHEMATICS** designer screen. Create integer variables called **FirstNo**, **SecondNo**, **Result**, and **Oper**. Generate a random number between 1 and 100 and store in variable **FirstNo**. Generate another integer number between 1 and 100 and store in variable **SecondNo**. Generate a random number between 1 and 4 and store in variable **Oper**, which will be used to select a mathematical operation. **Oper** will be assumed to take the values 1,2,3 and 4 for the mathematical operations of ad-

dition, subtraction, multiplication, and division. The program uses a Select-Case block to select the operation. For example, if the operation is addition then label **Operation.Text** is set to display the first number + the second number. The program waits for 10 seconds where the user is required to calculate the answer. After 10 seconds the answer is displayed by label **Answer** as the sum of the first and the second numbers. The same operation is carried out for the subtraction, multiplication, and division. Notice that the **Round2** function is used in division to limit the number of digits to be displayed to 6 after a division.

- After displaying the correct result the program waits for 10 seconds and then clears the screen and generates new question.

Figure 7.25 shows the mobile device screen before the answer is displayed. In Figure 7.26 the answer is displayed by the program.

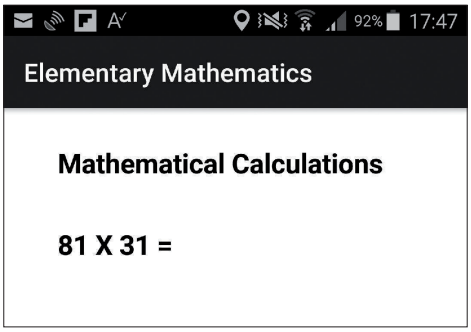


Figure 7.25 Mobile device screen before displaying the answer

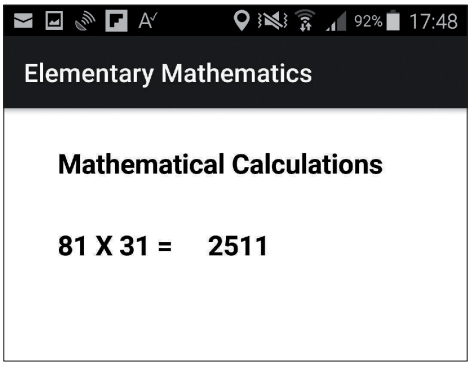


Figure 7.26 Mobile device screen after displaying the result

- Figure 7.27 shows the complete program listing (program: MATHEMATICS).

```
#Region  Project Attributes

#Region  Activity Attributes
```

```
Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private Answer As Label
  Private Label1 As Label
  Private Operation As Label
End Sub

Sub Activity_Create(FirstTime As Boolean)
  'Do not forget to load the layout file created with the visual designer.
For example:
  Dim FirstNo, SecondNo, Result, Oper As Int
  Activity.LoadLayout("MATHEMATICS")
  Activity.Title = "Elementary Mathematics"
  Do While True
    FirstNo = Rnd(1, 101)
    SecondNo = Rnd(1, 101)
    Oper = Rnd(1, 5)
    Select Oper
      Case 1
        Operation.Text = FirstNo & " + " & SecondNo & " = "
        Sleep(10000)
        Answer.Text = FirstNo + SecondNo
      Case 2
        Operation.Text = FirstNo & " - " & SecondNo & " = "
        Sleep(10000)
        Answer.Text =FirstNo - SecondNo
      Case 3
        Operation.Text = FirstNo & " X " & SecondNo & " = "
        Sleep(10000)
        Answer.Text = FirstNo * SecondNo
      Case 4
        Operation.Text = FirstNo & " / " & SecondNo & " = "
        Sleep(10000)
        Answer.Text = Round2(FirstNo / SecondNo, 6)
    End Select
    Sleep(10000)
    Operation.Text=""
    Answer.Text=""
```

```
  Loop
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)

End Sub
```

Figure 7.27 Program listing

Chapter 8 • Projects using the mobile device features

8.1 Overview

In this Chapter we shall be developing several simple projects using the B4A and the **Phone Library** which includes many features for the mobile device (e.g. an Android mobile phone). Some of the functions included in the **Phone Library** are:

- Call log (access to phone call log)
- Contacts list (access to stored contacts)
- Email
- Phone information
- Phone sensors (see next section)
- Phone events
- SMS handling (sending and receiving)
- Phone vibration control
- Ringtone management
- Voice recognition (speech to text)

It is hoped that the readers will learn how to use the various features of the mobile device by developing B4A based projects. The projects will be described as in the previous Chapter.

8.2 Phone Sensors

Phone Sensors is inside the **Phone Library**. Depending upon the model chosen, Android mobile phones include various sensors. Almost all up to date Android mobile phones support the following sensors:

- Ambient pressure sensor
- Accelerometer
- Gyroscope
- Light sensor
- Gravity sensor
- Magnetic field sensor
- Proximity sensor

Some models include additional sensors such as ambient temperature and relative humidity sensors. In B4A programs the sensor data is obtained using the **PhoneSensors** library. Before a sensor is used it must be initialized by specifying the type of sensor we wish to use. At the time of writing this book the following sensor types could be initialized:

- TYPE_ACCELEROMETER
- TYPE_GYROSCOPE
- TYPE_LIGHT
- TYPE_MAGNETIC_FIELD
- TYPE_ORIENTATION
- TYPE_PRESSURE
- TYPE_PROXIMITY

- TYPE_TEMPERATURE

After initializing the sensor type we can check to see whether the specified sensor is supported by our mobile phone. This is done using the statement **StartListening** which returns **True** if the specified sensor is supported. If the specified sensor is supported by our mobile phone then we can get the sensor data in the **SensorChanged** subroutine. This subroutine has a floating point array argument called **Values** that stores the sensor data. Depending upon the type of sensor used, **Values** stores one or more data. For example when using an accelerometer sensor, **Values** stores three values which are the acceleration in the x, y and z directions. If for example a pressure sensor is used then **Values** stores only one data item.

Some examples of using the mobile phone sensors are given in the following sections.

8.3 PROJECT 6 - Displaying the Ambient Pressure

8.3.1 Description

In this project the ambient pressure is read from the Android mobile phone pressure sensor, and is then displayed using a label.

8.3.2 Aim

The aim of this project is to show how the ambient pressure can be read from the Android mobile phone pressure sensor module.

8.3.3 Program Listing

In this program, a label is used to display the ambient pressure. The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking Tools-> B4A Bridge -> Connect
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Click **Add View** and add a new Label
- Change the name of the label to **AmbientPressure**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14

- Click **Tools** -> **Generate Members**, click Select **All Views**, and then click **Generate Members**, and then close the screen

- You should see the following code generated in the program section:

```
Sub Globals
    Private AmbientPressure As Label
End Sub
```

- Now, save the designer screen with the name, e.g. **PRESSURE**

- We have to include the library **Phone** in our program. Click the **Libraries Manager** tab at the bottom right hand side of the main program screen and select **Phone** by clicking on the box at its left.

- We now have to fill in the code. Insert the following statement inside **Sub Process_Globals** to define a variable called **apressure** of type **PhoneSensors**:

```
Dim apressure As PhoneSensors
```

- Load the designed layout **PRESSURE**, set the title to **Ambient Pressure** inside the **Activity_Create** and also insert the following code. In this code. In this code, **apressure** is initialized with type **TYPE_PRESSURE**. The program then calls function **StartListening** to checks whether the mobile phone supports pressure sensor module and if not an error message is displayed in a Msgbox. Notice here that **PressureTrig** is the name that will be used in the subroutine that will be triggered when there is change in sensor output:

```
Sub Activity_Create(FirstTime As Boolean)
    Activity.LoadLayout("PRESSURE")
    Activity.Title = "Ambient Pressure"
    apressure.Initialize(apressure.TYPE_PRESSURE)
    If apressure.StartListening("PressureTrig") = False Then
      Msgbox("Presure sensor is not supported", "Error")
    End If
End Sub
```

- Now, create a subroutine with the name **PressureTrig_SensorChanged** and display the pressure value inside this subroutine. Variable **Values(0)** returns the value of the pressure reading. Function **NumberFormat** converts a floating point number into a string. The first parameter of this function is the floating point number to be converted, the second parameter is the minimum integers, and the third parameter is the maximum number of digits after the decimal point:

```
Sub PressureTrig_SensorChanged(Values() As Float)
    Dim MyPressure As String
    MyPressure = NumberFormat(Values(0), 0, 3)
    AmbientPressure.Text = "Pressure = " & MyPressure & " millibars"
End Sub
```

- Figure 8.1 shows a typical display on an Android mobile phone. Notice that only 3 digits are displayed after the decimal point.

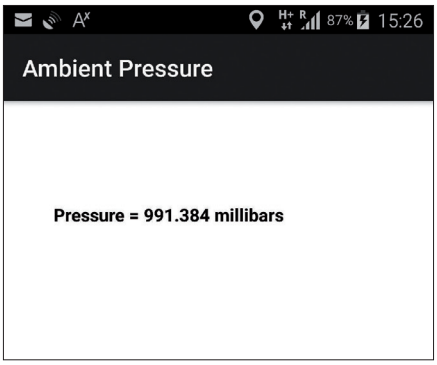


Figure 8.1 Typical display on an Android mobile phone

- The B4A program listing (program: PRESSURE) of the project is shown in Figure 8.2

```
#Region  Project Attributes
#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application
starts.
  'These variables can be accessed from all modules.
  Dim apressure As PhoneSensors
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private AmbientPressure As Label
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("PRESSURE")
  Activity.Title = "Ambient Pressure"
```

```
  apressure.Initialize(apressure.TYPE_PRESSURE)
  If apressure.StartListening("PressureTrig") = False Then
    Msgbox("Presure sensor is not supported", "Error")
  End If
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub PressureTrig_SensorChanged(Values() As Float)
  Dim MyPressure As String
  MyPressure = NumberFormat(Values(0), 0, 3)
  AmbientPressure.Text = "Pressure = " & MyPressure & " millibars"
End Sub
```

Figure 8.2 Program listing

8.3.4 Modified Program

The program given in Figure 8.2 can be modified by displaying the current date and time on the screen in addition to the pressure data. The steps are as follows:

- Add another label to the top of the first label. Name this label as DateAndTime and set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14
- Click **Tools** -> **Generate Members**, click Select **All Views**, and then click **Generate Members**, and then close the screen. Save the new layout as **PRESSURE2**
- Insert the date and time function inside the **PressureTrig_SensorChanged** subroutine as shown below:

```
Sub PressureTrig_SensorChanged(Values() As Float)
    Dim MyPressure As String
CurrentDateTime.Text = "Current Date & Time = " & _  DateTime.
Date(DateTime.Now) & " " & _
    DateTime.Time(DateTime.Now)
    MyPressure = NumberFormat(Values(0), 0, 3)
    AmbientPressure.Text = "Pressure = " & MyPressure & " millibars"
End Sub
```

Functions **DateTime.Date(DateTime.Now)** and **DateTime.Time(DateTime.Now)** are used to return the current date and the current time. Figure 8.3 shows a typical display on the Android mobile phone.

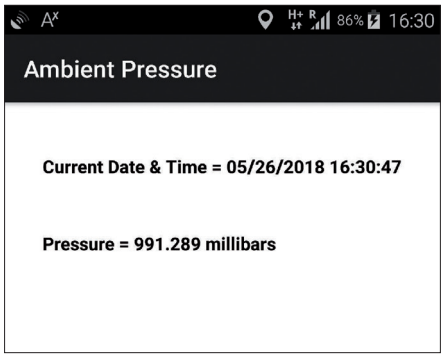


Figure 8.3 Typical display on an Android mobile phone

- The B4A program listing (program: PRESSURE2) of the project is shown in Figure 8.4

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application
starts.
  'These variables can be accessed from all modules.
  Dim apressure As PhoneSensors
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private AmbientPressure As Label
  Private CurrentDateTime As Label
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("PRESSURE2")
  Activity.Title = "Ambient Pressure"
  apressure.Initialize(apressure.TYPE_PRESSURE)
  If apressure.StartListening("PressureTrig") = False Then
    Msgbox("Presure sensor is not supported", "Error")
  End If
End Sub

Sub Activity_Resume
```

```
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub PressureTrig_SensorChanged(Values() As Float)
  Dim MyPressure As String
  CurrentDateTime.Text = "Current Date & Time = " & DateTime.
Date(DateTime.Now) & " " & _
  DateTime.Time(DateTime.Now)
  MyPressure = NumberFormat(Values(0), 0, 3)
  AmbientPressure.Text = "Pressure = " & MyPressure & " millibars"
End Sub
```

Figure 8.4 Program listing

8.4 PROJECT 7 - Displaying the Ambient Light Level

8.4.1 Description

In this project the ambient light level is read from the Android mobile phone light level sensor, and is then displayed using a label as in the previous project.

8.4.2 Aim

The aim of this project is to show how the ambient light level can be read from the Android mobile phone light level sensor module.

8.4.3 Program Listing

This program is basically same as in the previous project, but here the label is named **AmbientLight** instead of **AmbientPressure**. Additionally, the designer screen is saved as **LIGHT** instead of **PRESSURE**. Do not forget to include the **Phone** library in your program. The steps to write the code is as follows:

- Insert the following statement inside **Sub Process_Globals** to define a variable called **alight** of type **PhoneSensors**:

```
Dim alight As PhoneSensors
```

- Load the designed layout **LIGHT**, set the title to **Ambient Light Level** inside the **Activity_Create** and also insert the following code. In this code, alight is initialized with type **TYPE_LIGHT**. The program then calls function **StartListening** to checks whether the mobile phone supports light sensor module and if not an error message is displayed in a Msgbox. Notice here that **LightTrig** is the name that will be used in the subroutine that will be triggered when there is change in sensor output:

```
Sub Activity_Create(FirstTime As Boolean)
```

```
    Activity.LoadLayout("LIGHT")
    Activity.Title = "Ambient Light Level"
    alight.Initialize(alight.TYPE_LIGHT)
    If alight.StartListening("LightTrig") = False Then
      Msgbox("Light sensor is not supported", "Error")
    End If
End Sub
```

- Now, create a subroutine with the name **LightTrig_SensorChanged** and display the light level value inside this subroutine. Variable **Values(0)** returns the value of the light reading in lux. Function **NumberFormat** converts a floating point number into a string. The first parameter of this function is the floating point number to be converted, the second parameter is the minimum integers, and the third parameter is the maximum number of digits after the decimal point:

```
Sub LightTrig_SensorChanged(Values() As Float)
    Dim MyLight As String
    MyLight = NumberFormat(Values(0), 0, 3)
    AmbientLight.Text = "Light = " & MyLight & " lux"
  End Sub
```

- Figure 8.5 shows a typical display on the Android mobile phone.

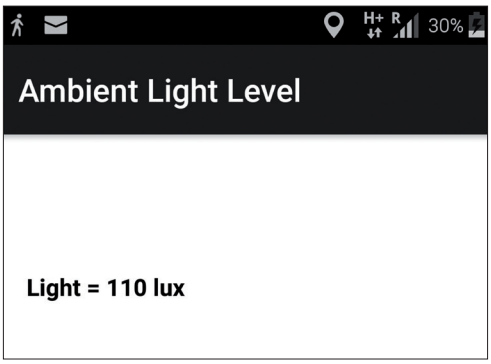


Figure 8.5 Typical display on mobile phone

- The program listing (program: LIGHT) is shown in Figure 8.6.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application
starts.
```

```
  'These variables can be accessed from all modules.
  Dim alight As PhoneSensors
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private AmbientLight As Label
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("LIGHT")
  Activity.Title = "Ambient Light Level"
  alight.Initialize(alight.TYPE_LIGHT)
  If alight.StartListening("LightTrig") = False Then
    Msgbox("Light sensor is not supported", "Error")
  End If
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub LightTrig_SensorChanged(Values() As Float)
  Dim MyLight As String
  MyLight = NumberFormat(Values(0), 0, 3)
  AmbientLight.Text = "Light = " & MyLight & " lux"
End Sub
```

Figure 8.6 Program listing

8.5 PROJECT 8 – Vibrating Phone at Low Light Level

8.5.1 Description

In this project the ambient light level is read from the Android mobile phone light level sensor, and it is compared with a pre-defined value (e.g. 10 lux). If the light level is below this pre-defined value then the phone is set to vibrate for 2 seconds to indicate that the light level is low.

8.5.2 Aim

The aim of this project is to show how the ambient light level can be read from the Android mobile phone light level sensor module and also how the phone can be set to vibrate.

8.5.3 Program Listing

Make sure that you include the **Phone** library in your program. This program is basically same as in the previous project, but here the code inside subroutine **LightTrig_Sensor-Changed** is modified to vibrate the phone is the light level is below the value specified by a variable called **MinLightLevel**. Save the layout as **VIBRATE**. Define a variable called **Vib** of type **PhoneVibrate** in **Process_Globals**. The contents of the subroutine is as follows. The light level is compared with the specified minimum light level and the phone is set to vibrate for 2 seconds (2000ms) if the measured light level is below the minimum value (make sure that your mobile phone **Notifications** is enabled):

```
Sub LightTrig_SensorChanged(Values() As Float)
    Dim MyLight As String
    Dim MinLightLevel = 10 As Int
    MyLight = NumberFormat(Values(0), 0, 3)
    AmbientLight.Text = "Light = " & MyLight & " lux"
    If MyLight < MinLightLevel Then
      Vib.Vibrate(2000)
    End If
End Sub
```

Figure 8.7 shows the program listing (program:VIBRATE) of the project).

```
#Region  Project Attribute

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
  Dim alight As PhoneSensors
  Dim Vib As PhoneVibrate
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private AmbientLight As Label
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("VIBRATE")
  Activity.Title = "Ambient Light Level"
  alight.Initialize(alight.TYPE_LIGHT)
  If alight.StartListening("LightTrig") = False Then
    Msgbox("Light sensor is not supported", "Error")
```

```
  End If
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub LightTrig_SensorChanged(Values() As Float)
  Dim MyLight As String
  Dim MinLightLevel = 10 As Int
  MyLight = NumberFormat(Values(0), 0, 3)
  AmbientLight.Text = "Light = " & MyLight & " lux"
  If MyLight < MinLightLevel Then
    Vib.Vibrate(2000)
  End If
End Sub
```

Figure 8.7 Program listing

8.6 PROJECT 9 - Displaying the Proximity With Start/Stop Buttons

8.6.1 Description

The proximity sensor gives the distance from the front of the mobile phone screen. The value is given in centimetres, although most devices return only two values: **near** and **far**, or 0 (for near) and a fixed integer value (e.g. 8 for far). In this project two buttons called **START** and **STOP** are used to start and stop the proximity displays respectively. The proximity is displayed using a label called **AmbientProximity**.

8.6.2 Aim

The aim of this project is to show how to use the proximity sensor and also how to start and stop the sensor display.

8.6.3 Program Listing

The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you

can see the design on your mobile phone

- Set the **Color** in the **Activity Properties** to white
- Click **Add View** and add a new Label
- Change the name of the label to **AmbientProximity**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14
- Click **Add View** and add two buttons to the design. Name the first button as **START** and the second button as **STOP**. Set the **Text Color** of these buttons to green and red respectively, their **Styles** to Bold, and their **Sizes** to 14. Set the **Text** of the buttons to **START** and **STOP** respectively. Figure 8.8 shows the design layout on the PC screen.

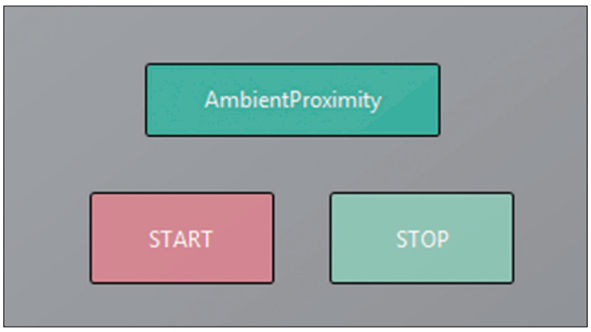


Figure 8.8 Design layout on the PC screen

- Click **Tools** -> **Generate Members**, click Select **All Views**, and then set **START** and **STOP** buttons to **Click** so that these buttons respond when clicked. Click **Generate Members**, and then close the screen (see Figure 8.9)

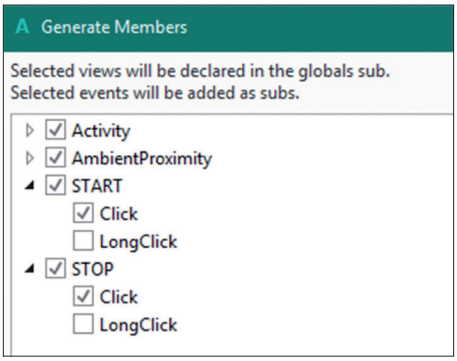


Figure 8.9 Generate members

- You should see the following code generated in the program section:

```
Sub Globals
    Private AmbientProximity As Label
    Private START As Button
    Private STOP As Button
End Sub

Sub STOP_Click
End Sub

Sub START_Click
End Sub
```

- Now, save the designer screen with the name, e.g. **PROXIMITY**

- We have to include the library **Phone** in our program. Click the **Libraries Manager** tab at the bottom right hand side of the main program screen and select **Phone** by clicking on the box at its left.

- We now have to fill in the code. Insert the following statement inside **Sub Process_Globals** to define a variable called **aproximity** of type **PhoneSensors**:

```
Dim aproximity As PhoneSensors
```

- Load the designed layout **PROXIMITY**, set the title to **Proximity** inside the **Activity_Create** and also insert the following code. In this code. In this code, **aproximity** is initialized with type **TYPE_PROXIMITY**.

- Insert the following code inside the **START_Click** subroutine so that the sensor display starts when the button is clicked:

```
aproximity.StartListening("ProximityTrig")
```

- Insert the following code inside the STOP_Click subroutine so that the display stops when the button is clicked:

```
aproximity.StopListening
```

- The program then calls subroutine **ProximityTrig_SensorChanged** when the **START** button is clicked. Inside this subroutine the proximity value is displayed as either 0 or 8 cm (for the Android phone used by the author)

- Figure 8.10 shows the screen of the Android mobile phone when the display was started.

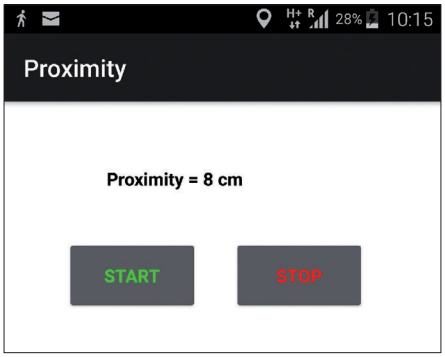


Figure 8.10 Display on the Android mobile phone

- The program listing (program: PROXIMITY) is shown in Figure 8.11.#

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application
starts.
  'These variables can be accessed from all modules.
  Dim aproximity As PhoneSensors
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private AmbientProximity As Label
  Private START As Button
  Private STOP As Button
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("PROXIMITY")
  Activity.Title = "Proximity"
  aproximity.Initialize(aproximity.TYPE_PROXIMITY)
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
```

```
End Sub

Sub ProximityTrig_SensorChanged(Values() As Float)
  Dim MyProximity As Int
  MyProximity = Values(0)
  AmbientProximity.Text = "Proximity = " & MyProximity & " cm"
End Sub


Sub STOP_Click
  aproximity.StopListening
End Sub

Sub START_Click
  aproximity.StartListening("ProximityTrig")
End Sub
```

Figure 8.11 Program listing

8.7 PROJECT 10 - Displaying the Acceleration and Sending via SMS

8.7.1 Description

In this project the acceleration sensor output is read and the acceleration of the mobile phone is displayed in three coordinates X, Y, and Z in meters/s^2. At the same time the X value of the acceleration is send to a specified phone number via SMS when a button called **SEND** is clicked.

8.7.2 Aim

The aim of this project is to show how to read the acceleration and also how to send data to a mobile phone via the SMS.

8.7.3 Program Listing

In this project a label and a button are used. The label displays the acceleration. When the button is clicked the current value of the acceleration is sent to a pre-specified phone number as an SMS text message.

The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect Click Designer and then Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone

- Set the **Color** in the **Activity Properties** to white
- Click **Add View** and add a new Label
- Change the name of the label to **PhoneAcceleration**
- Set the **Text Color** to black, its **Style** to BOLD, and its **Size** to 14
- Click **Add View** and add a button to the design. Name the button as **SEND** and set its **Text** to **SEND**. Set the **Text Color** of the button to green, its **Style** to Bold, and **Size** to 14.
- Figure 8.12 shows the design layout on the PC screen.

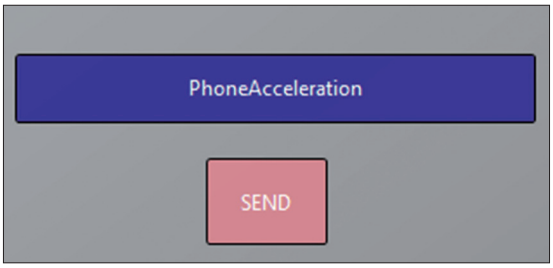


Figure 8.12 Design layout on the PC screen

- Click **Tools** -> **Generate Members**, click Select **All Views**, and then set **SEND** button to **Click** so that the button responds when clicked. Click **Generate Members**, and then close the screen (see Figure 8.13)

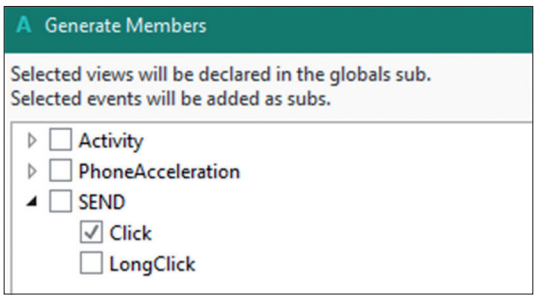


Figure 8.13 Generate members

You should see the following code generated in the program section:

```
Sub Globals
    Private PhoneAcceleration As Label
    Private SEND As Button
End Sub
```

```
Sub SEND_Click
End Sub
```

- Now, save the designer screen with the name, e.g. **ACCELERATION**

- We have to include the library **Phone** in our program. Click the **Libraries Manager** tab at the bottom right hand side of the main program screen and select **Phone** by clicking on the box at its left.

- We now have to fill in the code. Insert the following statements inside **Sub Process_Globals**. Notice that sending an SMS requires runtime permission and this can be done by first declaring a variable (e.g. **rp**) of type **RuntimePermissions**. Variables **X**,**Y** and **Z** will store the 3 values of the accelerometer:

```
Dim acc As PhoneSensors
  Dim MySms As PhoneSms
  Dim X, Y, Z As String
Dim rp As RuntimePermissions
```

- Load the designed layout **ACCELERATION**, set the title to **Acceleration** inside the **Activity_Create** and also insert the following code. In this code. In this code, **acc** is initialized with type **TYPE_ACCELEROMETER.** Insert the **StartListening** statement and specify the subroutine name (**AccTrig**) to be triggered when there is a change in the sensor data (notice that all Android mobile phones have built-in accelerometers and therefore it is not checked here whether the phone is equipped with an accelerometer). Subroutine **Activity_Create** should have the following contents. Notice that the program checks during the runtime if there is permission to send SMS, and if there is no permission then permission is requested. Subroutine **Activity_PermissionResult** is called automatically where the result of the permission can be checked if required:

```
Sub Activity_Create(FirstTime As Boolean)
    Activity.LoadLayout("ACCELERATION")
    Activity.Title = "Accelaration"
    acc.Initialize(acc.TYPE_ACCELEROMETER)
    acc.StartListening("AccTrig")
    If rp.Check(rp.PERMISSION_SEND_SMS) = False Then
      rp.CheckAndRequest(rp.PERMISSION_SEND_SMS)
    End If
End Sub
```

- The program calls subroutine **AccTrig_SensorChanged** when there is a change in the sensor data. Insert the following code inside this subroutine. This code extracts the **X**,**Y** and **Z** values of the acceleration and displays on the label:
-

```
Sub AccTrig_SensorChanged(Values() As Float)
    X = NumberFormat(Values(0), 0, 3)
    Y = NumberFormat(Values(1), 0, 3)
    Z = NumberFormat(Values(2), 0, 3)
    PhoneAcceleration.Text="X=" & X & "  Y=" & Y & "  Z=" & Z & "  m/s2"
End Sub
```

- When the **SEND** button is clicked, an SMS is sent to the specified mobile phone number using function **Send2**. The first argument of this function is the mobile phone number, the second argument is the message to be sent, the third argument is the **ReceiveSentNotification** and if it is True an SMS sent event will be called. Finally, the last argument is the **ReceiveDeliveredNotification** which if True then an SMS delivered event will be called.

```
Sub SEND_Click
    MySms.Send2("00447415987153", "X=" & X, True, True)
End Sub
```

- Figure 8.14 shows the screen of the Android mobile phone, displaying the accelerometer values.

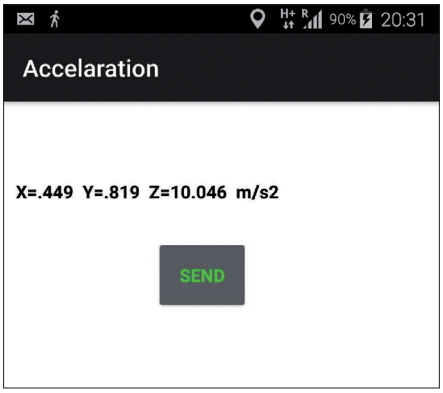


Figure 8.14 Display on the Android mobile phone

- Figure 8.15 shows the received SMS where the X value of the acceleration is received and displayed.

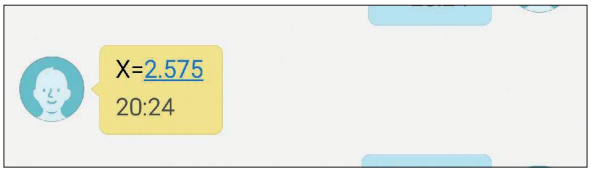


Figure 8.15 Received SMS

- The program listing (program: ACCELERATION) is shown in Figure 8.16.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
 'These global variables will be declared once when the application starts.
 'These variables can be accessed from all modules.
 Dim acc As PhoneSensors
 Dim MySms As PhoneSms
 Dim X, Y, Z As String
 Dim rp As RuntimePermissions
End Sub

Sub Globals
 'These global variables will be redeclared each time the activity is
created.
 'These variables can only be accessed from this module.
 Private PhoneAcceleration As Label
 Private SEND As Button
End Sub

Sub Activity_Create(FirstTime As Boolean)
 Activity.LoadLayout("ACCELERATION")
 Activity.Title = "Accelaration"
 acc.Initialize(acc.TYPE_ACCELEROMETER)
 acc.StartListening("AccTrig")
 If rp.Check(rp.PERMISSION_SEND_SMS) = False Then
   rp.CheckAndRequest(rp.PERMISSION_SEND_SMS)
 End If
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub Activity_PermissionResult (Permission As String, Result As Boolean)
End Sub

Sub AccTrig_SensorChanged(Values() As Float)
 X = NumberFormat(Values(0), 0, 3)
 Y = NumberFormat(Values(1), 0, 3)
 Z = NumberFormat(Values(2), 0, 3)
 PhoneAcceleration.Text="X=" & X & "  Y=" & Y & "  Z=" & Z & "  m/s2"
```

```
End Sub

Sub SEND_Click
  MySms.Send2("00447415987053", "X=" & X, False, False)
End Sub
```

Figure 8.16 Program listing

8.7.4 Modified Program

The program given in Figure 8.16 can be modified and made more user friendly if we add sent notification and delivery notification. These can easily be added to the program as Phone Events. The steps are as follows:

- Add the following statement to the Sub Globals:

```
Dim PEvt as PhoneEvents
```

- Initialize the Phone Events in Activity_Create:

```
PEvt.Initialize("PEvt")
```

- Set the notification arguments of function Send2 to True:

```
MySms.Send2("00447415987053", "X=" & X, True, True)
```

- Create a subroutine called **PEvt_SmsDelivered** to notify about the delivered message:

```
Sub PEvt_SmsDelivered(PhoneNumber As String, Intent As Intent)
    Msgbox("SMS delivered to number: " & PhoneNumber, "")
End Sub
```

- Create a subroutine called **PEvt_SmsSentStatus** to notify about the sent message using a Msgbox:

```
Sub PEvt_SmsSentStatus(SentOK As Boolean, Msg As String, _
    PhoneNumber As String, Intent As Intent)
If SentOK Then
    Msgbox("SMS sent to number: " & PhoneNumber, "")
  Else
    Msgbox("Error in sending SMS to: " & PhoneNumber, "")
    End If
End Sub
```

- Figure 8.17 shows the message displayed on the mobile phone in a Msgbox when the SMS message is sent.

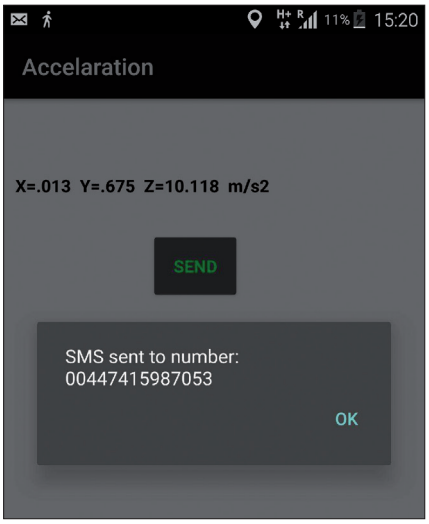


Figure 8.17 Displaying the sent notification message

- The complete program listing (program: ACCELERATION2) is shown in Figure 8.18.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application
starts.
  'These variables can be accessed from all modules.
  Dim acc As PhoneSensors
  Dim MySms As PhoneSms
  Dim X, Y, Z As String
  Dim rp As RuntimePermissions
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
   created.
  'These variables can only be accessed from this module.
  Private PhoneAcceleration As Label
  Private SEND As Button
  Dim PEvt As PhoneEvents
End Sub

Sub Activity_Create(FirstTime As Boolean)
```

```
  Activity.LoadLayout("ACCELERATION")
  Activity.Title = "Accelaration"
  acc.Initialize(acc.TYPE_ACCELEROMETER)
  PEvt.Initialize("PEvt")
  acc.StartListening("AccTrig")
  If rp.Check(rp.PERMISSION_SEND_SMS) = False Then
    rp.CheckAndRequest(rp.PERMISSION_SEND_SMS)
  End If
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub Activity_PermissionResult (Permission As String, Result As Boolean)
End Sub

Sub AccTrig_SensorChanged(Values() As Float)
  X = NumberFormat(Values(0), 0, 3)
  Y = NumberFormat(Values(1), 0, 3)
  Z = NumberFormat(Values(2), 0, 3)
  PhoneAcceleration.Text="X=" & X & "  Y=" & Y & "  Z=" & Z & "  m/s2"
End Sub

Sub PEvt_SmsDelivered(PhoneNumber As String, Intent As Intent)
  Msgbox("SMS delivered to number: " & PhoneNumber, "")
End Sub


Sub PEvt_SmsSentStatus(SentOK As Boolean, Msg As String, _
  PhoneNumber As String, Intent As Intent)
  If SentOK Then
    Msgbox("SMS sent to number: " & PhoneNumber, "")
  Else
    Msgbox("Error in sending SMS to: " & PhoneNumber, "")
  End If
End Sub


Sub SEND_Click
  MySms.Send2("00447415987053", "X=" & X, True, True)
End Sub
```

Figure 8.18 Program listing

8.8 PROJECT 11 – Using Multiple Sensors

8.8.1 Description

In this project the pressure sensor and the light sensor are used and their data are displayed by two labels.

8.8.2 Aim

The aim of this project is to show how more than one sensor can be used in a project.

8.8.3 Program Listing

In this project two labels are used to display the ambient pressure and the light level. The first one is called **AmbientPressure** and the second one is called **AmbientLight**.
The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Click **Add View** and add two new Labels
- Change the name of the first label to **AmbientPressure** and the second label to AmbientLight
- Set the **Text Colors** to black, their **Styles** to BOLD, and **Sizes** to 14
- Click **Tools** -> **Generate Members**, click Select **All Views**, and then Click **Generate Members**, and then close the screen (see Figure 8.19)

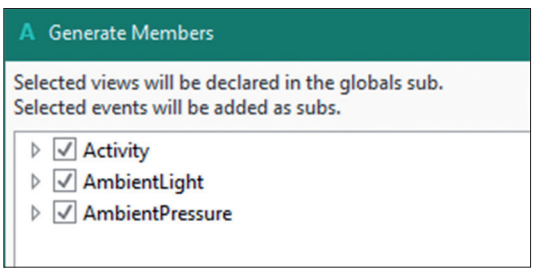


Figure 8.19 Generate members

- Now, save the designer screen with the name, e.g. **MULTIPLE**

- We have to include the library **Phone** in our program. Click the **Libraries Manager** tab at the bottom right hand side of the main program screen and select **Phone** by clicking on the box at its left.

- We now have to fill in the code. Insert the following statements inside **Sub Process_Globals**:

```
Dim apressure As PhoneSensors
Dim alight As PhoneSensors
```

- Load the designed layout **MULTIPLE**, set the title to **Ambient Pressure and Light Level** inside the **Activity_Create** and also insert the following code. In this code, **alight** is initialized with type **TYPE_LIGHT**, and **apressure** is initialized with type **TYPE_PRESSURE**. Insert the **StartListening** statements for the light and pressure sensors and specify the subroutine name as **SensorTrig**, to be activated when there is change in the light or the pressure sensor data. Subroutine **Activity_Create** should have the following contents:

```
Sub Activity_Create(FirstTime As Boolean)
    Activity.LoadLayout("MULTIPLE")
    Activity.Title = "Ambient Pressure and Light Level"
    alight.Initialize(alight.TYPE_LIGHT)
    apressure.Initialize(apressure.TYPE_PRESSURE)
    If alight.StartListening("SensorTrig") = False Then
      Msgbox("Light sensor is not supported", "Error")
    End If

    If apressure.StartListening("SensorTrig") = False Then
      Msgbox("Pressure sensor is not supported", "Error")
    End If
End Sub
```

- The program calls subroutine **SensorTrig_SensorChanged** when there is a change in the sensor data. Insert the following code inside this subroutine. Variable **ps** is defined of type **PhoneSensors**. Sender returns the object that triggered the event. In the code below, if the event is triggered by the light sensor than the light sensor output is displayed. Similarly, if the event is triggered by the pressure sensor then the pressure sensor output is displayed.

```
Sub SensorTrig_SensorChanged(Values() As Float)
    Dim MyLight As String
    Dim MyPressure As String
    Dim ps As PhoneSensors
    ps = Sender
```

```
    If Sender = alight Then
      MyLight = NumberFormat(Values(0), 0, 3)
      AmbientLight.Text = "Light = " & MyLight & " lux"
    End If

    If Sender = apressure Then
      MyPressure = NumberFormat(Values(0), 0, 3)
      AmbientPressure.Text = "Pressure = " & MyPressure & " mb"
    End If
End Sub
```

- Figure 8.20 shows a typical display on the Android mobile phone.

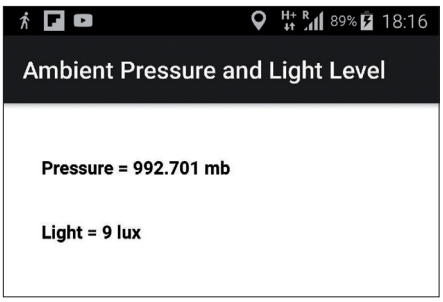


Figure 8.20 Display on the Android mobile phone

- Figure 8.21 shows the program listing (program: MULTIPLE) of the project.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application
starts.
  'These variables can be accessed from all modules.
  Dim alight As PhoneSensors
  Dim apressure As PhoneSensors
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity
  is created.
  'These variables can only be accessed from this module.
  Private AmbientLight As Label
  Private AmbientPressure As Label
End Sub
```

```
Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("MULTIPLE")
  Activity.Title = "Ambient Pressure and Light Level"
  alight.Initialize(alight.TYPE_LIGHT)
  apressure.Initialize(apressure.TYPE_PRESSURE)
  If alight.StartListening("SensorTrig") = False Then
    Msgbox("Light sensor is not supported", "Error")
  End If
  If apressure.StartListening("SensorTrig") = False Then
    Msgbox("Pressure sensor is not supported", "Error")
  End If
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub SensorTrig_SensorChanged(Values() As Float)
  Dim MyLight As String
  Dim MyPressure As String
  Dim ps As PhoneSensors
  ps = Sender
  If Sender = alight Then
    MyLight = NumberFormat(Values(0), 0, 3)
    AmbientLight.Text = "Light = " & MyLight & " lux"
  End If
  If Sender = apressure Then
    MyPressure = NumberFormat(Values(0), 0, 3)
    AmbientPressure.Text = "Pressure = " & MyPressure & " mb"
  End If
End Sub
```

Figure 8.21 Program listing

8.9 PROJECT 12 – Making Phone Calls

8.9.1 Description

In this project phone call is made to a pre-defined number when a button is pressed.

8.9.2 Aim

The aim of this project is to show how phone calls can be made from a B4A program.

8.9.3 Program Listing

In this project a label and a button are used. The label displays the message **CLICK TO CALL HOME**. Call is made when the button named **CALL** is clicked.

The steps to design the project are given below:

- Insert a label into the design layout and set the Text of this label to **CLICK TO CALL HOME**
- Insert a button under the label. Name and also set the Text of this button to **CALL**
- Generate Members as before. Name the layout as **CALL**
- Insert the following statements into **Process_Globals**:

```
Dim rp As RuntimePermissions
  Dim PC As PhoneCalls
```

- Insert the following statements into subroutine **Activity_Create**. Notice that as in the previous project, making a phone call requires the **CALL_PHONE** run-time permission to be enabled:

```
Sub Activity_Create(FirstTime As Boolean)
    Activity.LoadLayout("CALL")
    Activity.Title = "Call Home"
    If rp.Check(rp.PERMISSION_CALL_PHONE) = False Then
      rp.CheckAndRequest(rp.PERMISSION_CALL_PHONE)
    End If
End Sub
```

- Clicking the **CALL** button makes phone call to the specified number as shown in subroutine **CALL_Click** below:

```
Sub CALL_Click
  StartActivity(PC.Call("123456789"))
End Sub
```

- Figure 8.22 shows the display on the Android mobile phone.

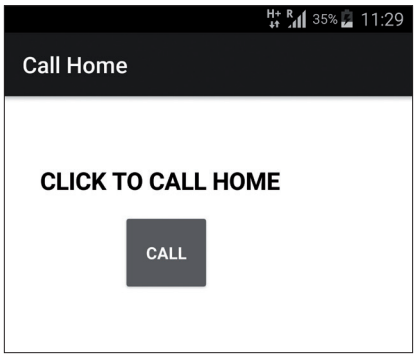


Figure 8.22 Display on the Android mobile phone

- Figure 8.23 shows the program (program: CALL) listing of the project.

```
#Region  Project Attribute

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application
starts.
  'These variables can be accessed from all modules.
  Dim rp As RuntimePermissions
  Dim PC As PhoneCalls
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
  Private CALL As Button
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("CALL")
  Activity.Title = "Call Home"
  If rp.Check(rp.PERMISSION_CALL_PHONE) = False Then
    rp.CheckAndRequest(rp.PERMISSION_CALL_PHONE)
  End If
End Sub

Sub Activity_Resume
End Sub
```

```
Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub Activity_PermissionResult (Permission As String, Result As Boolean)
End Sub


Sub CALL_Click
  StartActivity(PC.Call("00903922236464"))
End Sub
```

Figure 8.23 Program listing

8.10 PROJECT 13 – Saving the Sensor Data

8.10.1 Description

In this project the pressure sensor data is read and is then saved in a text file on the Android mobile phone device storage. The data is saved with the date and time attached to each record. In this project only 10 records are saved for simplicity. The file is named **Log.txt** and is saved in a folder called **Pressure**. The actual path to the saved file is:

Device storage/Android/data/b4a.example/files/pressure/Log.txt

8.10.2 Aim

The aim of this project is to show how text data can be created and then how data can be saved in a text file on the Android mobile phone.

8.10.3 Program Listing

In this project a label is used and this label displays the heading **Pressure Log...** Name the layout as **PRESSURE3**.

Writing to a file required the **WRITE_EXTERNAL_STORAGE** run-time permission. Insert the following statement inside **Sub_Globals.** Here, variable **Count** is initialized to 0 and is incremented each time a record is written to the file. The file is closed when 10 records are written to the file:

```
Sub Process_Globals
   Dim apressure As PhoneSensors
   Dim MyFile As TextWriter
   Dim rp As RuntimePermissions
   Dim Count = 0 As Int
End Sub
```

Insert the following statements inside Activity_Create:

```
Sub Activity_Create(FirstTime As Boolean)
```

```
Activity.LoadLayout("PRESSURE3")
Activity.Title = "Ambient Pressure"
apressure.Initialize(apressure.TYPE_PRESSURE)
If rp.Check(rp.PERMISSION_WRITE_EXTERNAL_STORAGE)=False Then
  rp.CheckAndRequest(rp.PERMISSION_WRITE_EXTERNAL_STORAGE)
End If

If apressure.StartListening("PressureTrig") = False Then
  Msgbox("Presure sensor is not supported", "Error")
End If
```

Also, insert the following lines inside **Activity_Create** to create a folder called **pressure** and a file under this folder called **Log.txt.** Write heading to the file:

```
File.MakeDir(File.DirDefaultExternal, "pressure")
MyFile.Initialize(File.OpenOutput(File.DirdefaultExternal & "/pressure",
"Log.txt",False))
MyFile.WriteLine("PRESSURE DATA LOG")
MyFile.WriteLine("=================")
MyFile.Writeline("")
End Sub
```

Function **MakeDir** creates a folder**.** Here, **File.DirDefaultExternal** refers to the default drive on the device which is also called the **Device storage** on most Android mobile phones. Files stored using **DirDefaultExternal** can be viewed using a browser. It is also possible to use other file storage areas such as the **File.DirInternal** to save private data that cannot be viewed using a browser.

Subroutine **PressureTrig_SensorChanged** is called whenever the sensor data changes. Inside this subroutine we read the current date and time and also the pressure output data. The pressure data is combined with the current data and time in the following format where a comma separates the two (assuming the date is 05/29/2-18, the time is 12:27:00 and the pressure value is 992.176mb):

```
992.176,  05/29/2018 12:27:00
```

The contents of this subroutine is as follows. Notice that variable **MyPressure** stores the combined date and time and the pressure data. The data is written to the file using function **WriteLine**. The data collection stops after 10 entries are made to the log file. The function **ToastMessageShow** displays a quick message on the Android mobile phone. By setting the argument to True we can extend the duration of this message:

```
Sub PressureTrig_SensorChanged(Values() As Float)
   Dim MyPressure As String
   Dim CDateTime As String
   CDateTime = DateTime.Date(DateTime.Now) & " " & DateTime.Time(DateTime.
```

```
Now)
   MyPressure = Round2(Values(0), 3)
   MyPressure = MyPressure & ",  " & CDateTime
   MyFile.WriteLine(MyPressure)
   Count = Count + 1
   If Count = 10 Then
     apressure.StopListening
     MyFile.close
     ToastMessageShow("End of data collection...",False)
   End If
End Sub
```

Figure 8.24 shows the Andoid display when the data collection is completed. Notice the message **End of data collection**.. at the bottom of the screen.

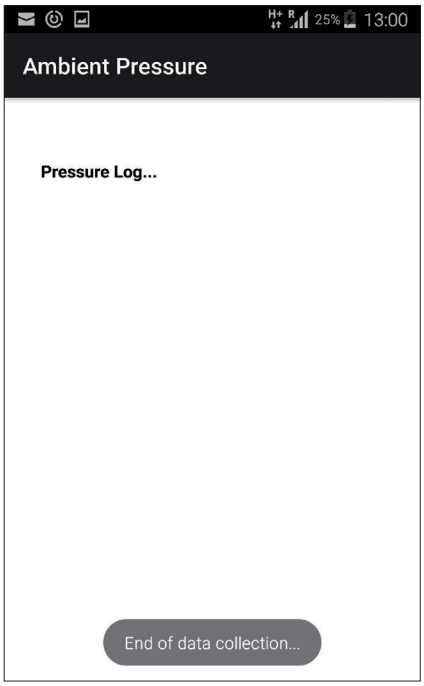
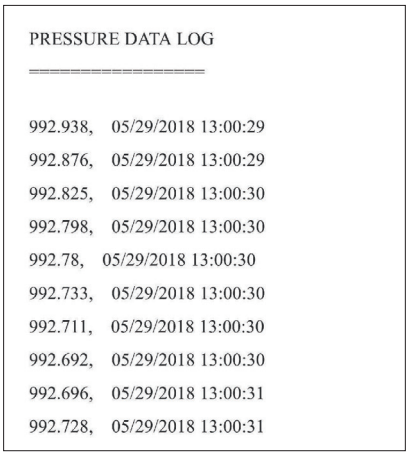


Figure 8.24 End of data collection

Figure 8.25 shows the contents of file Log.txt.


PRESSURE DATA LOG

992.938, 05/29/2018 13:00:29
992.876, 05/29/2018 13:00:29
992.825, 05/29/2018 13:00:30
992.798, 05/29/2018 13:00:30
992.78, 05/29/2018 13:00:30
992.733, 05/29/2018 13:00:30
992.711, 05/29/2018 13:00:30
992.692, 05/29/2018 13:00:30
992.696, 05/29/2018 13:00:31
992.728, 05/29/2018 13:00:31

Figure 8.25 Contents of file Log.txt

The program listing (program: PRESSURE3) is shown in Figure 8.26.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
  Dim apressure As PhoneSensors
  Dim MyFile As TextWriter
  Dim rp As RuntimePermissions
  Dim Count = 0 As Int
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
created.
  'These variables can only be accessed from this module.
End Sub

Sub Activity_Create(FirstTime As Boolean)
  'Do not forget to load the layout file created with the visual designer.
For example:
  Activity.LoadLayout("PRESSURE3")
  Activity.Title = "Ambient Pressure"
  apressure.Initialize(apressure.TYPE_PRESSURE)
  If rp.Check(rp.PERMISSION_WRITE_EXTERNAL_STORAGE) = False Then
    rp.CheckAndRequest(rp.PERMISSION_WRITE_EXTERNAL_STORAGE)
```

```
  End If
  If apressure.StartListening("PressureTrig") = False Then
    Msgbox("Presure sensor is not supported", "Error")
  End If
'
' Create folder called pressure and create (open) file Log.txt under
pressure
' Write heading PRESSURE DATA LOG to the first line of the file (the written
' lines are terminated with carriage-return and line-feed)
'
  File.MakeDir(File.DirDefaultExternal, "pressure")
  MyFile.Initialize(File.OpenOutput(File.DirdefaultExternal & "/
pressure","Log.txt", False))
  MyFile.WriteLine("PRESSURE DATA LOG")
  MyFile.WriteLine("=================")
  MyFile.Writeline("")
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub PressureTrig_SensorChanged(Values() As Float)
  Dim MyPressure As String
  Dim CDateTime As String
  CDateTime = DateTime.Date(DateTime.Now) & " " & DateTime.Time(DateTime.Now)
  MyPressure = Round2(Values(0), 3)
  MyPressure = MyPressure & ",  " & CDateTime
  MyFile.WriteLine(MyPressure)
  Count = Count + 1
  If Count = 10 Then
    apressure.StopListening
    MyFile.close
    ToastMessageShow("End of data collection...",False)
  End If
End Sub
```

Figure 8.26 Program listing

8.11 PROJECT 14 – Talking Light Level

8.11.1 Description

In this project a button named **AmbientLight** is placed on the mobile phone screen. The ambient light level sensor data is read and is then displayed on the button. When the button is clicked the current light level data is sent to the Text-to-Speech engine of the Android

mobile phone. Thus we can hear the light level spoken on the mobile phone when the button is clicked.

8.11.2 Aim

The aim of this project is to show how the text-to-Speech engine of the Android mobile phone can be used in an B4A project.

8.11.3 Program Listing

The steps to write the program are:

- Create a button called **AmbientLight** and set its Text **Color** to white. Click **Tools -> Generate Members.** Click **Select All Views** and click the button as shown in Figure 8.27. Click **Generate Members** and exit the screen. Save the layout as **TTS**.

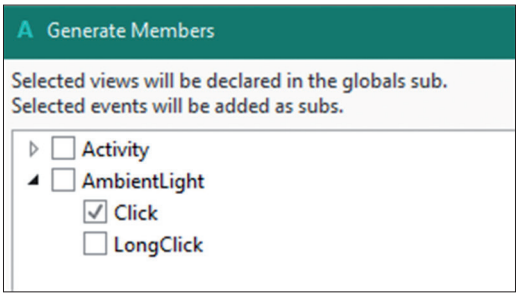


Figure 8.27 Generate members

- Include libraries **Phone** and **TTS** by clicking the **Libraries Manager** at the bottom right hand side of the screen. Insert the following statements inside subroutine **Process_Globals**. Notice that variable **MyTTS** is of type **TTS**.

```
Dim alight As PhoneSensors
  Dim MyTTS As TTS
Dim MyLight As String
  Dim RDY as Int
```

- Insert the following statement inside **Activity_Create**, which loads layout **TTS**, sets the activity title to **Ambient Light Level**, initializes the Text-to-Speech library, and also initializes the light sensor library:

```
Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("TTS")
  Activity.Title = "Ambient Light Level"
  MyTTS.Initialize("MyTTS")
  alight.Initialize(alight.TYPE_LIGHT)
  If alight.StartListening("LightTrig") = False Then
    Msgbox("Light sensor is not supported", "Error")
```

```
  End If
End Sub
```

- Notice that the argument of the TTS initialization routine specifies the event that will be raised when the TTS engine is ready. In this program the event will be handled by subroutine **MyTTS_Ready**. Inside this subroutine we will set a flag (called **RDY**) to indicate that the TST engine is ready:

```
Sub MyTTS_Ready(Success As Boolean)
    If Success Then
      RDY = 1
    Else
      RDY = 0
    End If
End Sub
```

- The value of the current light level is continuously displayed on the button by subroutine **LightTrig_SensorChanged**:

```
Sub LightTrig_SensorChanged(Values() As Float)
  MyLight = NumberFormat(Values(0), 0, 3)
  AmbientLight.Text = MyLight
End Sub
```

- When the button is clicked the current value of the light sensor is spoken through the TTS engine. The statements inside subroutine **AmbientLight_Click** are as follows:

```
Sub AmbientLight_Click
    If RDY = 1 Then
        MyTTS.Speak("The ambient light level is" & MyLight & "
Lux",True)
    End If
End Sub
```

If the second argument of function **Speak** is True then any waiting texts are discarded and the newly specified text is spoken. If for example the light level is 60, the Text-to-Speech engine speaks the words:

The ambient light level is sixty lux

- Figure 8.28 shows a typical display on the Android mobile phone.

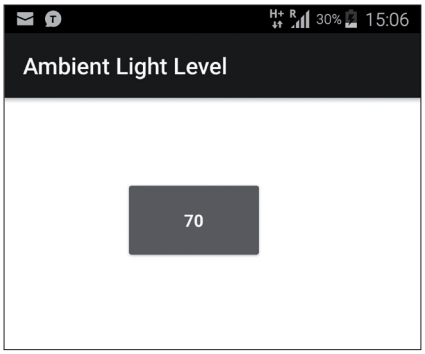


Figure 8.28 Display on the Android mobile phone

- Figure 8.29 shows the program listing (program: TTS) of the project.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application
  starts.
  'These variables can be accessed from all modules.
  Dim alight As PhoneSensors
  Dim MyTTS As TTS
  Dim MyLight As String
  Dim RDY As Int
End Sub

Sub Globals
  'These global variables will be redeclared each time the activity is
  created.
  'These variables can only be accessed from this module.
  Private AmbientLight As Button
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("TTS")
  Activity.Title = "Ambient Light Level"
  MyTTS.Initialize("MyTTS")
  alight.Initialize(alight.TYPE_LIGHT)
  If alight.StartListening("LightTrig") = False Then
    Msgbox("Light sensor is not supported", "Error")
  End If
End Sub
```

```
Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub


'
' See if the TTS engine is ready. Variable RDY is set to 1 when the
' TTS engine is ready, otherwise it is cleared to 0
'
Sub MyTTS_Ready(Success As Boolean)
  If Success Then
    RDY = 1
  Else
    RDY = 0
  End If
End Sub

'
' Read and display the ambient light level
'
Sub LightTrig_SensorChanged(Values() As Float)
  MyLight = NumberFormat(Values(0), 0, 3)
  AmbientLight.Text = MyLight
End Sub


'
' This subroutine "speaks" the ambient light level if the TTS engine is
ready
'
Sub AmbientLight_Click
  If RDY = 1 Then
    MyTTS.Speak("The ambient light level is" & MyLight & " Lux",True)
  End If
End Sub
```

Figure 8.29 Program listing

The TTS library supports some other function such as specifying a different language, changing the speech rate, changing the speech pitch and so on.

8.12 Other Phone Sensors

Some other useful phone sensors are described below briefly. Although these sensors are

not used in the projects in this book, their use is similar to the use of the accelerometer sensor:

Gyroscope: This sensor gives the mobile phone angular velocity in Radians/second for three axes.

Magnetic field: This sensor gives the ambient magnetic field around the mobile phone in micro-tesla for the three axes.

Orientation: This sensor gives the mobile phone orientation in degrees for the azimuth, pitch, and roll.

Chapter 9 • Using the Global Positioning System (GPS)

9.1 Overview

In this Chapter we shall be looking at how to use the built-in GPS of the Android mobile phone. A project is given in this Chapter which extracts and displays the local GPS location data.

Android GPS functions are handled by the B4A GPS library. Three types of objects are available in the GPS library: **GPS**, **GPSSatellite**, and **Location**.

GPS handles the connections and events, initializes, starts, and stops the GPS. **GPSSatellite** returns various information about the GPS satellites, such as the azimuth, elevation, pseudo-random number (Prn) for the satellites, and the signal to noise ratio (Snr). **Location** returns the local location data such as the latitude, longitude, altitude, bearing, speed, time, and conversion functions.

9.2 PROJECT 15 – Displaying the Location Data

9.2.1 Description

In this project we will extract and display the local location data such as the latitude, longitude, and the altitude on our Android mobile phone.

9.2.2 Aim

The aim of this project is to show how the built-in GPS of the Android mobile phone can be used in a project.

9.2.3 Program Listing

In this program, three labels are used to display the headings latitude, longitude, and altitude. Additionally, three EditText boxes are used to display the values of the latitude, longitude, and the altitude. The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Add three labels with **Text Color** in black, **Size** 14, and set the **Text** of these labels to **Latitude:**, **Longitude:**, and **Altitude:** respectively.

- Add three Edit Text boxes with **Text Color** black, **Size** 12, and set their names to **Latitude**, **Longitude**, and **Altitude** respectively. Position these Edit Text boxes next to the labels with the same names.

- Click **Tools** -> **Generate Members**, click Select **All Views**, and then click **Generate Members** (see Figure 9.1). Save the layout with the name **GPS** and then close the designed window.

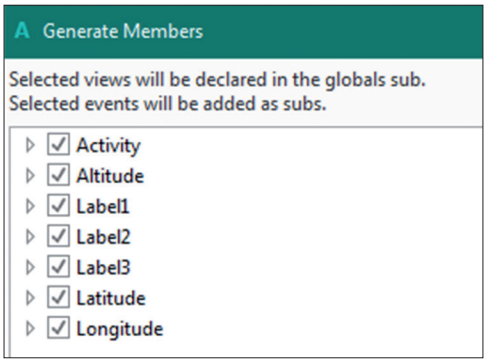


Figure 9.1 Generate members

We are now ready to fill in the code for our GPS project. Click the **Libraries Manager** at the bottom right hand side of the screen and click to include the **GPS** library and the **RuntimePermissions** library in our code.

Create a variable called **MyGPS** of type GPS, and a variable called **rp** of type RuntimePermissions inside **Sub Process_Globals**:

```
Sub Process_Globals
    Dim MyGPS As GPS
    Dim rp As RuntimePermissions
End Sub
```

Inside subroutine **Activity_Create** we define the layout to be loaded as **GPS**, set the activity title to **MY LOCATION** and enable the runtime permission to access the GPS. The code then initializes the GPS library with the name **MyGPS** so that subroutine **MyGPS_LocationChanged** will be called automatically whenever there is change in the received GPS data. If the GPS is not enabled on the Android device then the message **Enable GPS on your device** will be displayed temporarily and you will be set to the relevant part of the device Settings screen to enable the GPS. If on the other hand the GPS is already enabled then it is started. Notice that function **MyGPS.start** has two arguments: The first argument specifies the shortest time between events (measured in milliseconds) and the second argument specifies the shortest change in distance (measured in metres). Setting both parameters to 0 assumes the highest frequency. If for example the first parameter is set to 5000 and the second parameter to 10, then the GPS data will be updated at every 5 seconds, or whenever the distance changes by 10 metres or more:

```
Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("GPS")
  Activity.Title = "MY LOCATION"
  rp.CheckAndRequest(rp.PERMISSION_ACCESS_FINE_LOCATION)
  Wait For Activity_PermissionResult (Permission As String, Result As
Boolean)

  MyGPS.Initialize("MyGPS")
  If MyGPS.GPSEnabled = False Then
    ToastMessageShow("Enable GPS on your device", True)
    StartActivity(MyGPS.LocationSettingsIntent)
  Else
    ToastMessageShow("GPS already enabled",True)
    MyGPS.Start(0, 0)
  End If
End Sub
```

Subroutine **MyGPS_LocationChanged** is called automatically whenever there is change in the GPS data. This subroutine has one argument that can be used to extract the received GPS location data. Inside this subroutine the current values of the latitude, longitude, and the altitude are read and displayed by the Edit Text boxes. The latitude and the longitude are converted so that they are displayed in degrees and minutes formats. The altitude is displayed in metres:

```
Sub MyGPS_LocationChanged(MyLocation As Location)
    Latitude.Text = MyLocation.ConvertToMinutes(MyLocation.Latitude)
    Longitude.Text = MyLocation.ConvertToMinutes(MyLocation.Longitude)
    Altitude.Text = NumberFormat(MyLocation.Altitude, 2, 6)
End Sub
```

Figure 9.2 shows a sample display of the local location data on the Android mobile phone. In this example, the latitude was 51º 27.35748 minutes, the longitude was 0º 3.13066 minutes, and the altitude was 92 metres. Make sure that you are outdoors when using the GPS since there may not be satellite signals indoors.

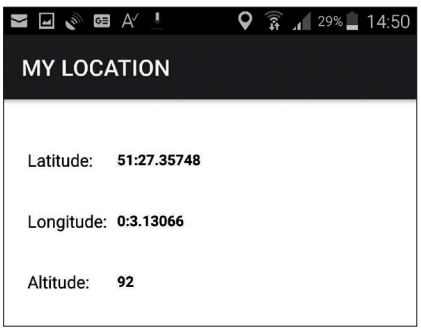


Figure 9.2 Sample display of the location data

The program listing (program: GPS) is shown in Figure 9.3.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
 'These global variables will be declared once when the application starts.
 'These variables can be accessed from all modules.
   Dim MyGPS As GPS
 Dim rp As RuntimePermissions
End Sub

Sub Globals
 'These global variables will be redeclared each time the activity is
created.
 'These variables can only be accessed from this module.
 Private Altitude As EditText
 Private Label1 As Label
 Private Label2 As Label
 Private Label3 As Label
 Private Latitude As EditText
 Private Longitude As EditText
End Sub

Sub Activity_Create(FirstTime As Boolean)
 Activity.LoadLayout("GPS")
 Activity.Title = "MY LOCATION"
 rp.CheckAndRequest(rp.PERMISSION_ACCESS_FINE_LOCATION)
 Wait For Activity_PermissionResult (Permission As String, Result As
Boolean)

 MyGPS.Initialize("MyGPS")
 If MyGPS.GPSEnabled = False Then
   ToastMessageShow("Enable GPS on your device", True)
   StartActivity(MyGPS.LocationSettingsIntent)
 Else
   ToastMessageShow("GPS already enabled",True)
   MyGPS.Start(0, 0)
 End If
End Sub

Sub Activity_Resume
End Sub
```

```
Sub Activity_Pause (UserClosed As Boolean)
End Sub

Sub MyGPS_LocationChanged(MyLocation As Location)
  Latitude.Text = MyLocation.ConvertToMinutes(MyLocation.Latitude)
  Longitude.Text = MyLocation.ConvertToMinutes(MyLocation.Longitude)
  Altitude.Text = NumberFormat(MyLocation.Altitude, 2, 6)
End Sub
```

Figure 9.3 Program listing

Chapter 10 • Android to PC WI-FI interface

10.1 Overview

In this Chapter we shall be looking at how an Android device (mobile phone or tablet) can communicate with a PC over a Wi-Fi link.

Communication over a Wi-Fi link uses the B4A network library. This library contains objects for working with **TCP** (ServerSocket and Socket) and **UDP** (UDPSoocket and UDPPacket) type data communication protocols. Both TCP and UDP are protocols used for sending and receiving data packets over the Internet, where the data is sent to or received from a specified IP address and port number. TCP stands for Transmission Control Protocol and it is a reliable and connection based protocol for sending and receiving data packets over the Internet. UDP stands for User Datagram Protocol and it is a connectionless protocol for sending and receiving data packets over the Internet. TCP is a highly reliable communication protocol as it provides error checking and acknowledges the received and sent data packets. Any missing or corrupted data packets are re-transmitted. UDP on the other hand is less reliable and there is no acknowledgement to confirm the sending or receipt of a data packet. As a result, TCP is not as fast as the UDP protocol as it has bigger overheads. TCP protocol requires the client and the server sides to be connected to each other before data exchange can take place. UDP on the other hand does not require the client or the server to be connected before exchanging data.

In this Chapter we shall be developing several projects to show how data can be exchanged between a PC and an Android mobile phone using UDP data packets.

10.2 PROJECT 16 – Sending and Receiving Data From a PC

10.2.1 Description

In this project we will send messages from the Android mobile phone to the PC over the Internet using the UDP protocol where the messages will be displayed on the PC. At the same time, the data sent by the PC will be displayed on the mobile phone.

10.2.2 Aim

The aim of this project is to show how the UDP protocol can be used to send and receive data over the internet between a PC and an Android mobile phone.

10.2.3 Block Diagram

Figure 10.1 shows the block diagram of the project. The communication between the PC and the Android mobile phone is via the local Wi-Fi router, using the UDP protocol. The IP addresses of the mobile phone and the PC are "192.168.1.78" and "192.168.1.71" respectively.

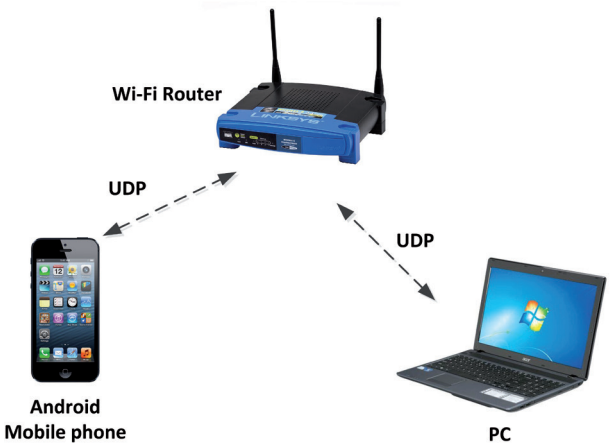


Figure 10.1 Block diagram of the project

10.2.4 Program Listing

In this program, two labels, two Edit Text boxes, and a button are used. The labels display the texts **SENT** and **RECEIVED**. Next to these labels are two Edit Text boxes named **Sent** and **Received**. Next to the **Sent** Edit Text box is a button labelled and named as **SEND**. Clicking this button sends the text in the Sent box to the PC. Data received from the PC is displayed in the Received Edit text box (see Figure 10.2 for the Android screen layout).

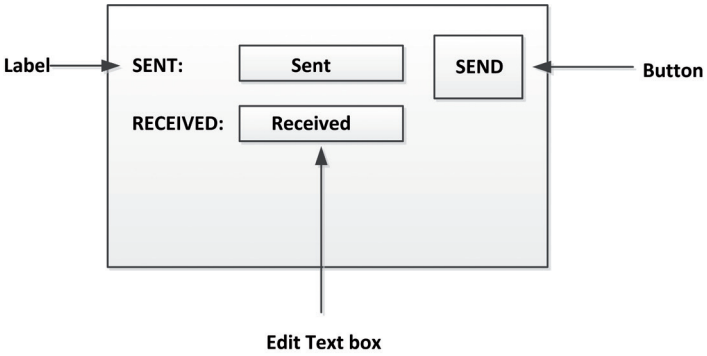


Figure 10.2 The Android screen layout

The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**

- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Add two labels with **Text Color** in black, **Size** 14, and set the **Text** of these labels to **SENT:** and **RECEIVED**:.
- Add two Edit Text boxes next to the labels and name them as **Sent** and **Received**, set their **Styles** to Bold, **Sizes** to 12, and **Text Colors** to black.
- Add a button to the right hand side of the upper Edit Text box and name it as **Send**.
- Click **Tools -> Generate Members** and **Select All Views**. Then click button **Send** and enable **Click** so that the button responds to clicks. Exit after clicking **Generate Members**.
- Save the layout with the name **UDP1** and then close the window.

We can now write the code for our project. Enable the **Network** library by clicking the Libraries Manager. Create a variable called **UdpSocket** of type UDPSocket and a variable called **UdpPAcket** of type UDPPacket in **Process_Globals**:

```
Sub Process_Globals
   Dim UdpSocket As UDPSocket
   Dim UdpPacket As UDPPacket
End Sub
```

In subroutine **Activate_Create**, load layout **UDP1**, set the title to **UDP TEST** and initialize the socket with local port number 2000, packet size 1024 bytes, and specify the first name of the subroutine to receive packets as **UDP** (the actual subroutine name will be **UDP_PacketArrived**):

```
Sub Activity_Create(FirstTime As Boolean)
   Activity.LoadLayout("UDP1")
   Activity.Title = "UDP TEST"
   If FirstTime Then
     UdpSocket.Initialize("UDP", 2000, 1024)
   End If
End Sub
```

Now, we should create a subroutine with the name **UDP_PacketArrived** which will be activated automatically whenever a packet is received. This subroutine should have one argument of type **UDPPacket**. Here, we convert the received packet into a string and then display it in Edit Text box called **Received**:

```
Sub UDP_PacketArrived(Packets As UDPPacket)
Dim msg As String
 msg = BytesToString(Packets.Data, Packets.Offset, Packets.Length, "UTF8")
 Received.Text = msg
End Sub
```

The remainder of the code implements the button code. Here, the text entered by the user to the Edit Text box **Sent** is sent after the packet is initialized. Variable **MyData** stores the text entered by the user. Notice that in this example, the IP address of the PC was "192.168.1.71". The target port address is set to 2000. Packet is sent by calling function **Send** with the **UdpPacket** as its argument:

```
Sub Send_Click
 Dim MyData() As Byte
 MyData = Sent.Text.GetBytes("UTF8")
 UdpPacket.Initialize(MyData, "192.168.1.71", 2000)
 UdpSocket.Send(UdpPacket)
End Sub
```

Figure 10.3 shows the complete program listing (program: UDP1).

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
 'These global variables will be declared once when the application starts.
 'These variables can be accessed from all modules.
 Dim UdpSocket As UDPSocket
 Dim UdpPacket As UDPPacket
End Sub

Sub Globals
 'These global variables will be redeclared each time the activity is
created.
 'These variables can only be accessed from this module.
 Private Send As Button
 Private Label1 As Label
 Private Label2 As Label
 Private Received As EditText
 Private Sent As EditText
End Sub

Sub Activity_Create(FirstTime As Boolean)
 Activity.LoadLayout("UDP1")
```

```
  Activity.Title = "UDP TEST"
  If FirstTime Then
    UdpSocket.Initialize("UDP", 2000, 1024)
  End If
End Sub

Sub UDP_PacketArrived(Packets As UDPPacket)
  Dim msg As String
  msg = BytesToString(Packets.Data, Packets.Offset, Packets.Length, "UTF8")
  Received.Text = msg
End Sub


Sub Send_Click
  Dim MyData() As Byte
  MyData = Sent.Text.GetBytes("UTF8")
  UdpPacket.Initialize(MyData, "192.168.1.71", 2000)
  UdpSocket.Send(UdpPacket)
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub
```

Figure 10.3 Program listing

10.2.5 Testing

The project can be tested by running a UDP test program on the PC. Packet Sender (https://packetsender.com) is one such program that can be used to send or receive UDP or TCP packets over the Internet. Download the program on your PC and then run it. Select the UDP protocol, set the Port number to 2000, the destination IP address (mobile phone IP address) to "192.168.1.78". **Click File -> Settings** and then click the **Network** tab and set the UDP Port Server number to 2000.

Install the UDP program (Figure 10.3) on your mobile phone and activate it. Write a text message (e.g. **This is the PC**) in the box labelled **ASCII** and click the **Send** button. You should see this message displayed on the mobile phone. Write a text message on your mobile phone (e.g. **This is the mobile**) and click the **SEND** button. You should see this message received and displayed by the Packet Sender program as shown in Figure 10.4.

Figure 10.5 shows the message displayed on the mobile phone.

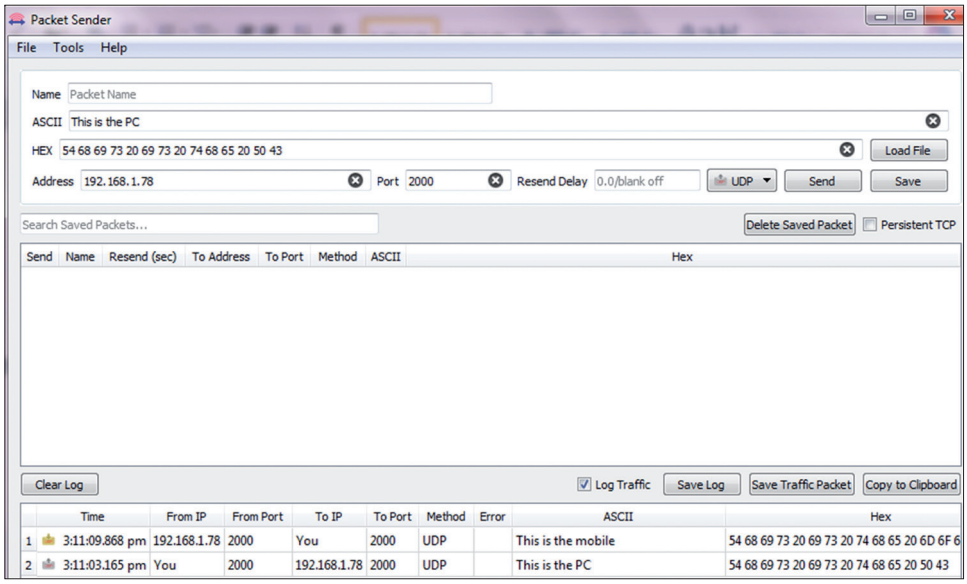


Figure 10.4 Message displayed by the packet Sender program

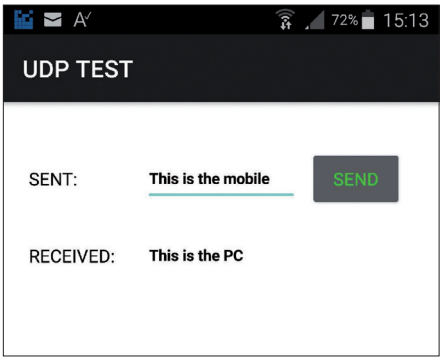


Figure 10.5 Message displayed on the mobile phone screen

10.3 PROJECT 17 – Word Reversing By the PC

10.3.1 Description

In this project we will send words from the Android mobile phone to the PC over the Internet using the UDP protocol. The letter positions in the words will be reversed by the PC and will then be sent back to the mobile phone where it will be displayed on the screen.

10.3.2 Aim

The aim of this project is to show how a UDP based program can be developed on the PC using the Visual Studio software package.

10.3.3 Block Diagram

The block diagram of the project is as in Figure 10.1.

10.3.4 Program Listing

The mobile phone program is as in Figure 10.3 where the user enters text to Edit Text box **Sent**, and then clicks the **SEND** button to send it to the PC. The PC will receive this text, reverse the letter positions and will then send back the text which will be displayed in the **Received** Edit Text box on the mobile phone.

The program on the PC (program: UDPTEST) has been developed using the **Visual Studio** package where the **Visual Basic** language was used. The program was **Windows Form Application** type with a single form, and its full listing shown in Figure 10.6. At the beginning of the program the Network library and various other libraries are imported to the program. The program is executed as soon as the form is loaded. The UDP port number is stored in integer variable called **port**. A new UDP client has been created with port number 2000 and the **IPEndPoint** has been declared such that the program can receive packets on port 2000 and from any IP address. The program connects to the mobile phone using the **Connect** function where the mobile phone IP address and the port number are specified. The remainder of the program is executed in an endless loop where text messages are received and stored in variable **data**. This data is then converted into string, the letter positions are reversed (by calling function **StrReverse**), and the string is converted to bytes and is sent back to the mobile phone as an UDP packet. This text is displayed on the mobile phone screen.

```
Imports System.Net
Imports System.Net.Sockets
Imports System.Text


Public Class Form1

    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.
Load
        Dim port As Integer = 2000
        Dim UDPClient As New UdpClient(2000)
        Dim endPoint As IPEndPoint = New IPEndPoint(IPAddress.Any, port)
        '
        ' Connect to the mobile phone
        '
        UDPClient.Connect("192.168.1.78", 2000)

        '
        ' Endless loop. Receive a packet, convert into a string in message,
        ' reverse the letter positions, convert to bytes, and send back to
        ' the mobile phone
        '
        While (True)
            Dim data() As Byte
            data = UDPClient.Receive(endPoint)
```

```
                Dim message As String = Encoding.ASCII.GetString(data)
                message = StrReverse(message)
                Dim sdata() As Byte = Encoding.ASCII.GetBytes(message)
                UDPClient.Send(sdata, sdata.Length)
            End While
        End Sub

    End Class
```

Figure 10.6 Visual Studio program listing

Figure 10.7 shows the screen of the mobile phone where a text message is sent and the letter positions are reversed in the received message.

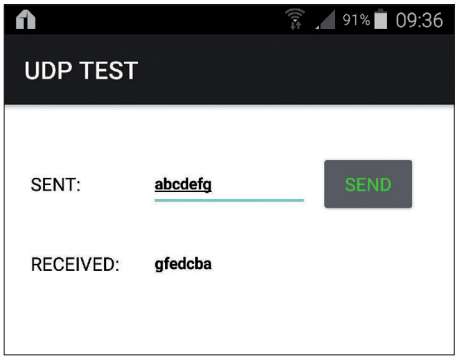


Figure 10.7 Mobile phone screen

Chapter 11 • Android to Raspberry PI WI-FI interface

11.1 Overview

In this Chapter we shall be looking at how an Android device (mobile phone or tablet) can communicate with a Raspberry Pi computer over a Wi-Fi link using the UDP protocol.
Before looking at the details of the Android to Raspberry Pi interface, it is worthwhile to review some basic features of the Raspberry Pi for those readers who may be new to the Raspberry Pi.

11.2 The Raspberry Pi Computer

The Raspberry Pi is a low-cost, single-board, powerful computer, capable of running a full operating system and also capable of doing everything that a laptop or a desktop computer can do, such as creating and editing documents, getting on the Internet, receiving and sending mails, playing games, developing programs to monitor and control its environment via electronic sensors and actuators, and many more.

There are several different models of the Raspberry Pi available, each having slightly different features. The fundamental features of all the raspberry Pi computers are similar, all using ARM processors, all having the operating system installed on an SD card, all having on-board memory, and input-output interface connectors. Some models, such as the *Raspberry Pi 3* and *Raspberry Pi Zero W* have built-in Wi-Fi and Bluetooth capabilities, making them easy to get online and to communicate with similar devices having Wi-Fi or Bluetooth capabilities. In this Chapter we shall be using the Raspberry Pi 3, which is currently one of the most popular Raspberry Pi models available.

11.2.1 The Raspberry Pi 3 Board

Figure 11.1 shows the Raspberry Pi 3 board with the major components marked. Some details on each component are given in this section.

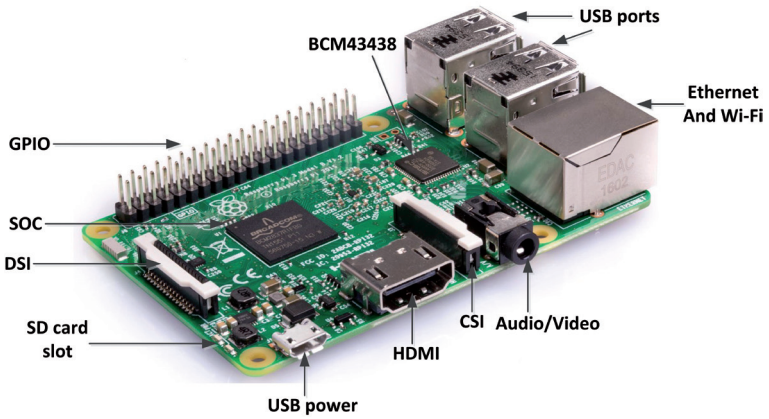


Figure 11.1 Raspberry Pi 3 board

USB ports: The Raspberry Pi 3 has 4 USB ports to connect mouse, keyboard, webcam etc.

Ethernet and Wi-Fi: Although the Raspberry Pi 3 has built-in Wi-Fi, it can also directly be connected to a router through an Ethernet cable connected to this socket.

Audio/Video Jack: A headphone or a speaker can be connected to this 3.5mm socket. This socket also carries composite video interface.

CSI: This is the Camera Serial Interface where an official Raspberry Pi camera can be attached here.

HDMI: A suitable monitor can be connected to this port. The port carries both audio and video.

USB power: A +5V 2A power supply should be connected to this USB socket to provide power to the raspberry Pi 3.

SD card slot: A micro SD card carrying the operating system must be attached to this slot.

DSI: A suitable display can be connected to this Display Interface.

SOC: This is the Broadcom BCM2837 System On Chip which contains the 1.2GHz 64-bit quad core ARM Cortex-A53 processor.

GPIO: The General Purpose Input-Output port is 40 pin wide.

BCM43438: This chip provides the Wi-Fi and Bluetooth to the Raspberry Pi 3.

11.2.2 Setting Up the Wi-Fi and Remote Access on Raspberry Pi

It is very likely that you will want to access your Raspberry Pi 3 remotely from your desktop or laptop computer. The easiest option here is to enable Wi-Fi on your Pi computer and then access it from your computer using the SSH client protocol. This protocol requires a server and a client. The server is your Pi computer and the client is your desktop or laptop computer. In this section we will see how to enable the Wi-Fi on your Pi computer and how to access it remotely.

Setting Up Wi-Fi

To enable the Wi-Fi on your Raspberry Pi, the steps are as follows:

- Click on the Wi-Fi icon which is a pair of red crosses at the top right hand side of the screen
- Select your Wi-Fi router from the displayed list (see Figure 11.2)

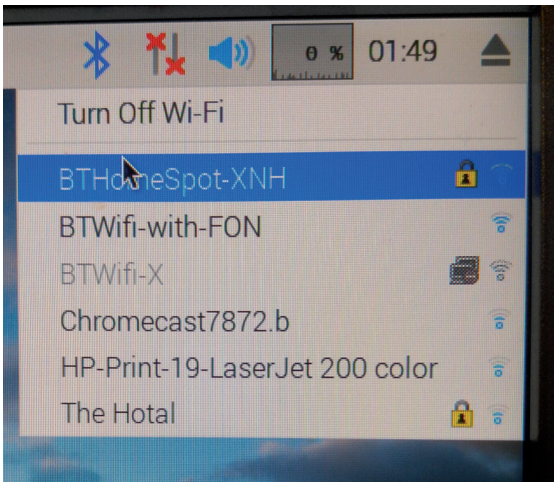


Figure 11.2 Select your Wi-Fi from the list

- Enter the password for your Wi-Fi router
- The Wi-Fi icon should become a typical Wi-Fi image. If you click on the icon now you should see a green tick next to the selected router as shown in Figure 11.3.

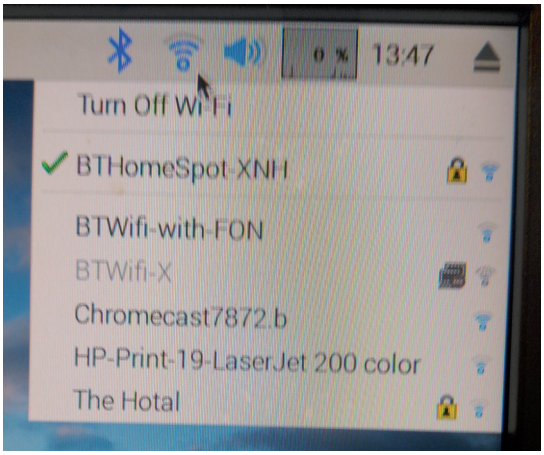


Figure 11.3 Connected to the Wi-Fi successfully

- To see the IP address of your Wi-Fi connection, place the mouse over the Wi-Fi icon as shown in Figure 11.4. In this example the IP address was 192.168.1.84

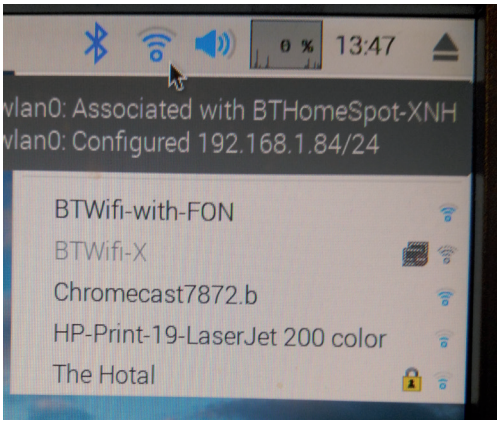


Figure 11.4 IP address of our connection

Remote Access

The program we will be using to access our Raspberry Pi 3 is called **Putty** with the SSH protocol. The steps to download and use Putty are as follows:

- Download Putty from the following link (or search Google for "Download Putty")

  http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

- For security reasons the SSH protocol is disabled by default on a new operating system. To enable it, click on the **Applications** menu at the top left of the screen, click **Accessories**, and then click **Terminal** (see Figure 11.5)

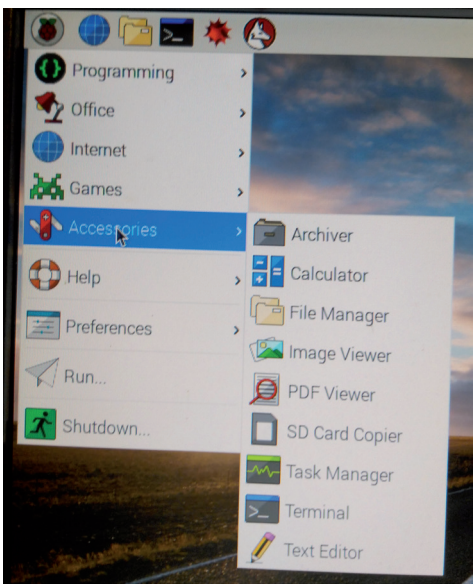


Figure 11.5 Access the Terminal menu

- You should now be in the Raspberry Pi 3 command prompt. Type:

```
sudo raspi-config
```

  to go into the configuration menu and select **Interface Options**. Go down to **P2 SSH** and enable SSH as shown in Figure 11.6

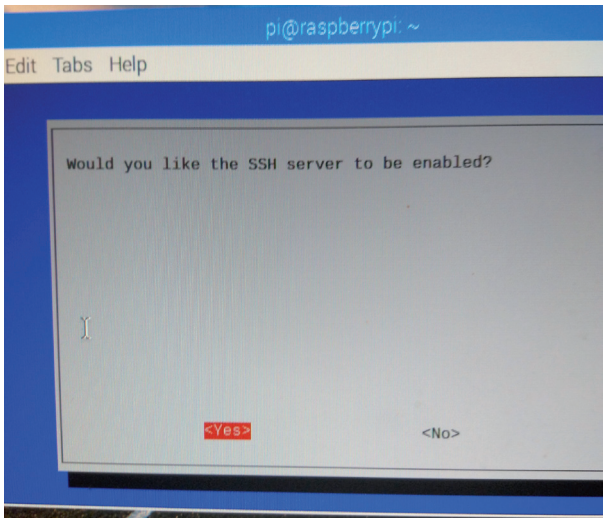


Figure 11.6 Enable the SSH server

- Click **<Finish>** to exit the configuration menu. You should now be back in the command mode, identified by the prompt:

```
pi@raspberrypi:~ $
```

- Putty is a standalone program and there is no need to install it. Simply double click to run it. You should see the Putty startup screen as in Figure 11.7.

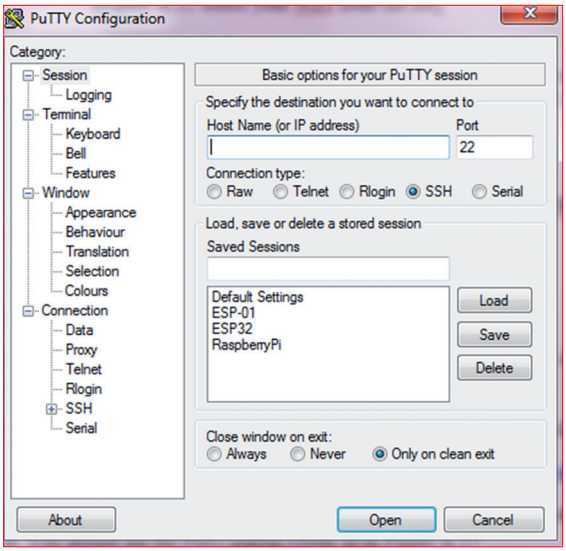


Figure 11.7 Putty startup screen

- Make sure that the Connection type is SSH and enter the IP address of your Raspberry Pi 3. Click Open as shown in Figure 11.8.

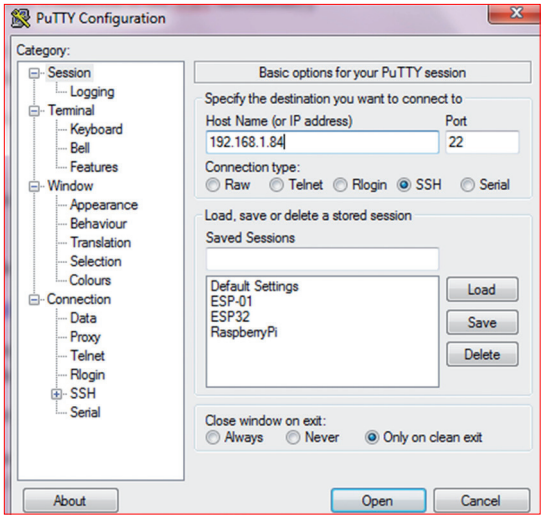


Figure 11.8 Enter the IP address

The message shown in Figure 11.9 will be displayed on the PC screen the first time you access the Raspberry Pi 3. Click **Yes** to accept this security alert.

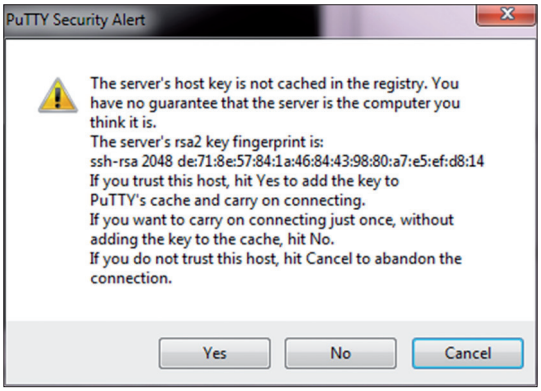


Figure 11.9 Click Yes to accept

- You will then be prompted for the username and password. The default values are:

  Username: **pi**
  Password: **raspberry**

- After a successful login you should see the Raspberry Pi command prompt as in Figure 11.10.

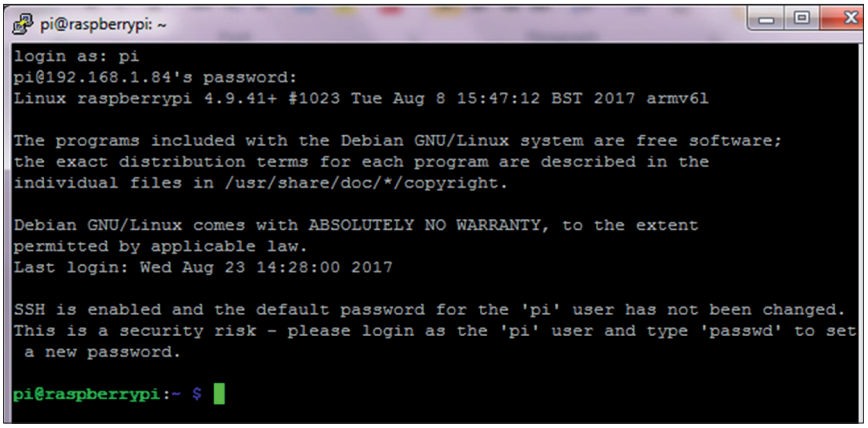


Figure 11.10 Successful login

11.2.3 Raspberry Pi 3 GPIO Pin Definitions

The Raspberry Pi 3 is connected to external electronic circuits and devices using its GPIO (General Purpose Input Output) port connector. This is a 2.54mm, 40-pin expansion header, arranged in a 2x20 strip as shown in Figure 11.11.

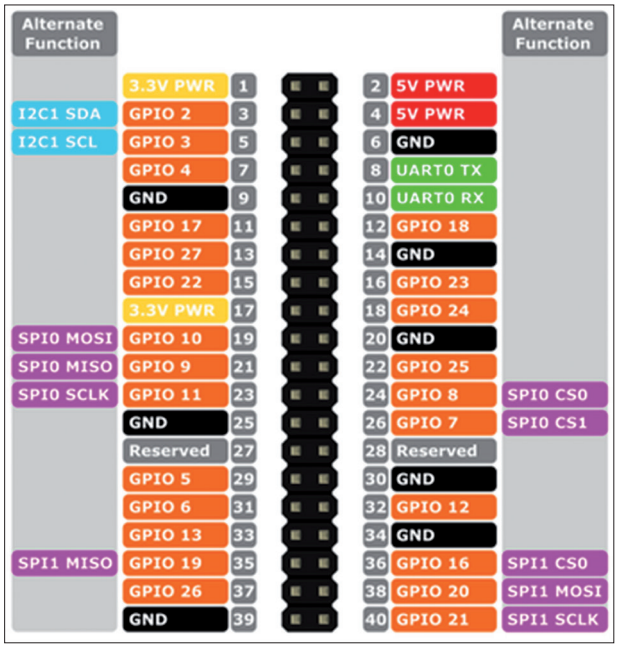


Figure 11.11 Raspberry Pi 3 GPIO pins

When the GPIO connector is at the far side of the board, the pins at the bottom, starting from the left of the connector are numbered as 1, 3, 5, 7, and so on, while the ones at the top are numbered as 2, 4, 6, 8 and so on.

The GPIO provides 26 general purpose bi-directional I/O pins. Some of the pins have multiple functions. For example, pins 3 and 5 are the GPIO2 and GPIO3 input-output pins respectively. These pins can also be used as the I2C bus I2C SDA and I2C SCL pins respectively. Similarly, pins 9,10,11,19 can either be used as general purpose input-output, or as the SPI bus pins. Pins 8 and 10 are reserved for UART serial communication.

Two power outputs are provided: +3.3V and +5.0V. The GPIO pins operate at +3.3V logic levels (not like many other computer circuits that operate with +5V). A pin can either be an input or an output. When configured as an output, the pin voltage is either 0V (logic 0) or +3.3V (logic 1). Raspberry Pi 3 is normally operated using an external power supply (e.g. a mains adapter) with +5V output and minimum 2A current capacity. A 3.3V output pin can supply up to 16mA of current. The total current drawn from all output pins should not exceed the 51mA limit. Care should be taken when connecting external devices to the GPIO pins as drawing excessive currents or short-circuiting a pin can easily damage your Pi. The amount of current that can be supplied by the 5V pin depends on many factors such as the current required by the Pi itself, current taken by the USB peripherals, camera current, HDMI port current, and so on.

When configured as an input, a voltage above +1.7V will be taken as logic 1, and a voltage below +1.7V will be taken as logic 0. Care should be taken not to supply voltages greater

than +3.3V to any I/O pin as large voltages can easily damage your Pi. The Raspberry Pi 3, like others in the family has no over-voltage protection circuitry.

11.2.4 The GPIO Library

The GPIO library is called RPi.GPIO and it should already be installed on your Raspberry Pi 3. This library must be included at the beginning of your Python programs if you will be using the GPIO functions. The statement to include this library is:

```
import RPi.GPIO as GPIO
```

If you get an error while trying to import the GPIO library then it is possible that the library is not installed. Enter the following commands while in the command mode (identified by the prompt **pi@raspberrypi:~ $**) to install the GPIO library (characters that should be entered by you are in bold):

```
pi@raspberrypi: ~$ sudo apt-get update
pi@raspberrypi: ~$ sudo apt-get install python-dev
pi@raspberrypi: ~$ sudo apt-get install python-rpi.gpio
```

The GPIO provides a number of useful functions. The available functions are given in the next sections

11.2.5 Pin Numbering

There are two ways that we can refer to the GPIO pins. The first is using the BOARD numbering, where the pin numbers on the GPIO connector of the Raspberry Pi 3 are used. Enter the following statement to use the BOARD method:

```
GPIO.setmode(GPIO.BOARD)
```

The second numbering system, also known as the BCM method is the preferred method and it uses the channel numbers allocated to the pins. This method requires that you know which channel number refers to which pin on the board. In this book we shall be using this second method. Enter the following statement to use the BCM method:

```
GPIO.setmode(GPIO.BCM)
```

The GPIO is a 40 pin header, mounted at one side of the board. Appendix A shows the Raspberry Pi 3 GPIO pin configuration.

11.2.6 Channel (I/O port pin) Configuration

Input Configuration

You need to configure the channels (or port pins) you are using whether they are input or output channels. The following statement is used to configure a channel as an input. Here, channel refers to the channel number based on the **setmode** statement above:

```
GPIO.setup(channel, GPIO.IN)
```

When there is nothing connected to an input pin, the data at this input is not defined. We can specify additional parameters with the input configuration statement to connect pull-up or pull-down resistors by software to an input pin. The required statements are:
For pull-down:

```
GPIO.setup(channel, GPIO.IN,  pull_up_down=GPIO.PUD_DOWN)
```

For pull-up:

```
GPIO.setup(channel, GPIO.IN,  pull_up_down=GPIO.PUD_UP)
```

We can detect an edge change of an input signal at an input pin. Edge change is when the signal changes from LOW to HIGH (rising edge), or from HIGH to LOW (falling edge). For example, pressing a push-button switch can cause an edge change at the input of a pin. The following statements can be used to wait for an edge of the input signal. These are blocking functions. i.e. the program will wait until the specified edge is detected at the input signal. For example, if this is a push-button, the program will wait until the button is pressed:

To wait for a rising edge:

```
GPIO.wait_for_edge(channel, GPIO.RISING)
```

To wait for a falling edge:

```
GPIO.wait_for_edge(channel, GPIO.FALLING)
```

We can also wait until either a rising or a falling edge is detected by using the following statement:

```
GPIO.wait_for_edge(channel, GPIO.BOTH)
```

We can use event detection function with an input pin. This way, we can execute the event detection code whenever an event is detected. Events can be rising edge, falling edge, or change in either edge of the signal. Event detection is usually used in loops where we can check for the event while executing other code.

For example, to add rising event detection to an input pin:

```
GPIO.add_event_detect(channel, GPIO.RISING)
```

We can check whether or not the event occurred by the following statement:

```
If GPIO.event_detected(channel):
  ……
  ……
```

Event detection can be removed by the following statement:

```
GPIO.remove_event_detect(channel)
```

We can also use interrupt facilities (or callbacks) to detect events. Here, the event is handled inside a user function. The main program carries on its usual duties and as soon as the event occurs the program stops whatever it is doing and jumps to the event handling function. For example, the following statement can be used to add interrupt based event handling to our programs on rising edge of an input signal. In this example, the event handling code is the function named **MyHandler**:

```
GPIO.add_event_detect(channel, GPIO.RISING, callback=MyHandler)
……
……

def MyHandler(channel):
  ……
  ……
```

We can add more than one interrupt by using the add_event_callback function. Here the callback functions are executed sequentially:

```
GPIO.add_event_detect(channel, GPIO.RISING)
GPIO.add_event_callback(channel, MyHandler1)
GPIO.add_event_callback(channel, MyHandler2)
……
……

def MyHandler1(channel):
  ……
  ……

def MyHandler2(channel):
  ……
  ……
```

When we use mechanical switches in our projects we get what is known as the switch bouncing problem. This occurs as the contacts of the switch bounce many times until they settle to their final state. Switch bouncing could generate several pulses before it settles down. We can avoid switch bouncing problems in hardware or software. GPIO library provides a parameter called bounce-time that can be used to eliminate the switch bouncing problem. An example use of this parameter is shown below where the switch bounce time

is assumed to be 10ms:

```
GPIO.add_event_detect(channel,GPIO=RISING,callback=MyHandler, bouncetime=10)
```

We can also use the callback statement to specify the switch bouncing time as:

```
GPIO.add_event_callback(channel, MyHandler, bouncetime=10)
```

To read the state of an input pin we can use the following statement:

```
GPIO.input(channel)
```

Output Configuration
The following statement is used to configure a channel as an output. Here, channel refers to the port number based on the **setmode** statement described earlier:

```
GPIO.setup(channel, GPIO.OUT)
```

We can specify a value for an output pin during its setup. For example, we can configure a channel as output and at the same time set its value to logic HIGH (+3.3V):

```
GPIO.setup(channel, GPIO.OUT, initial=GPIO.HIGH)
```

To send data to an output port pin we can use the following statement:

```
GPIO.output(channel, value)
```

Where value can be 0 (or GPIO.LOW, or False), or 1 (or GPIO.HIGH, or True)

At the end of the program we should return all the used resources to the operating system. This is done by including the following statement at the end of our program:

```
GPIO.cleanup()
```

11.3 PROJECT 18 – Controlling an LED From Android Mobile Phone

11.3.1 Description

In this project an LED is connected to one of the GPIO ports of the Raspberry Pi 3. This LED is controlled by sending **ON** and **OFF** messages from the Android mobile phone using the UDP protocol over the Wi-Fi link. Additionally, sending command **#** closes the network and exits the program orderly.

11.3.2 Aim

The aim of this project is to show how a B4A program can be developed to send UDP packets from an Android mobile phone to a Raspberry Pi 3 over a Wi-Fi link to control an LED connected to the Raspberry Pi 3.

11.3.3 Block Diagram

Figure 11.12 shows the block diagram of the project. The communication between the Android mobile phone and the Raspberry Pi 3 computer is via the local Wi-Fi router.

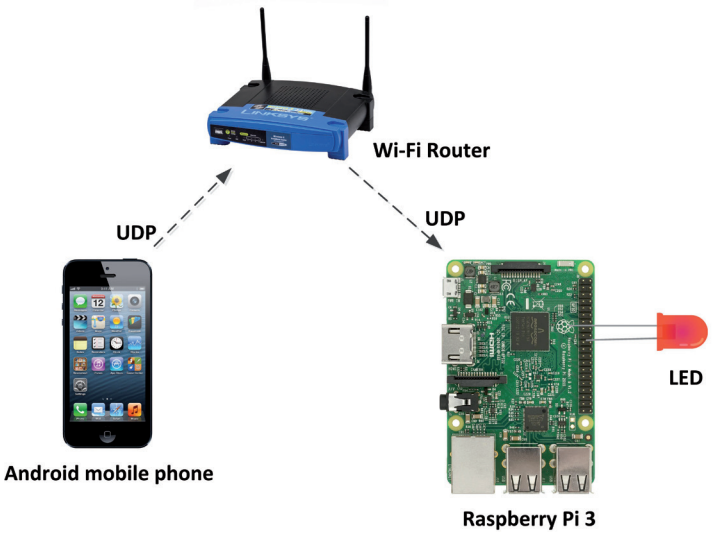


Figure 11.12 Block diagram of the project

11.3.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 11.13. A small LED is connected to port pin GPIO 2 (pin 3) of the Raspberry Pi 3 through a current limiting resistor. The value of the current limiting resistor is calculated as follows:

The output high voltage of a GPIO pin is 3.3V. The voltage across an LED is approximately 1.8V. The current through the LED depends upon the type of LED used and the amount of required brightness. Assuming that we are using a small LED, we can assume a forward LED current of about 3mA. Then, the value of the current limiting resistor is:

R = (3.3V – 1.8V) / 3mA = 500 ohm. We can choose a 470 ohm resistor

In Figure 11.13 the LED is operated in current sourcing mode where a high output from the GPIO pin drives the LED. The LED can also be operated in current sinking mode where the other end of the LED is connected to +3.3V supply and not to the ground. In current sinking mode the LED is turned ON when the GPIO pin is at logic low.

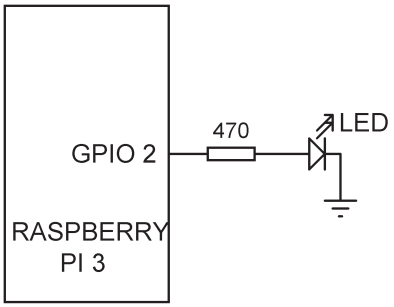


Figure 11.13 Circuit diagram of the project

11.3.5 Construction

The Raspberry Pi side of the project is constructed on a breadboard as shown in Figure 11.14. Female-male jumper cables are used to connect the LED to the GPIO port. Notice that the short side of the LED must be connected to ground.

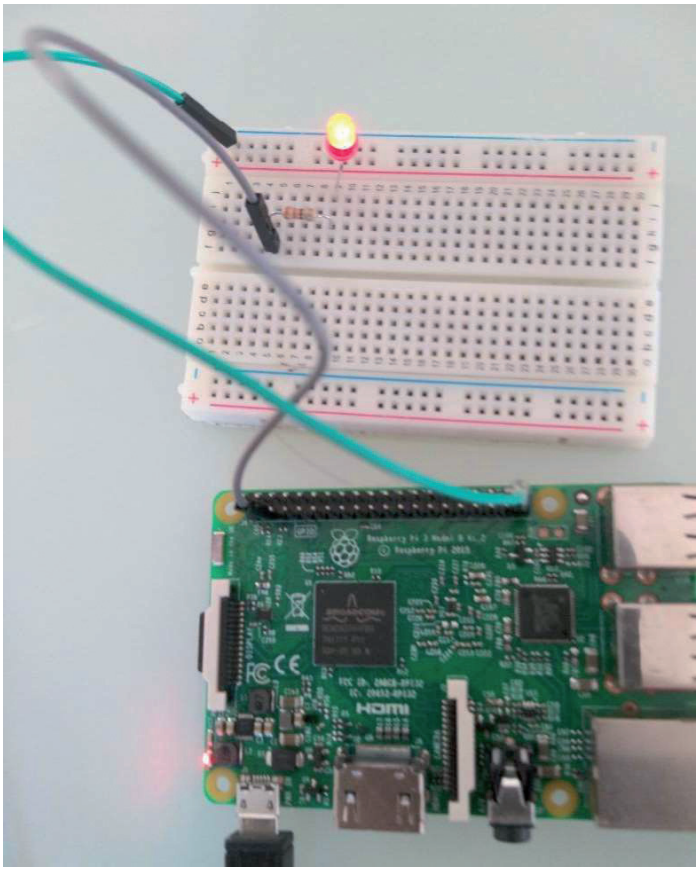


Figure 11.14 Raspberry Pi connected to an LED

11.3.6 Android Program

The B4A program consists of a label, an Edit Text box, and a button, where the label displays the text **COMMAND:**. The command is entered to the Edit Text box and is sent to the Raspberry Pi when the button is clicked. The LED is turned ON and OFF with the commands ON and OFF respectively. The Edit Text box is named as **Command**, and the button is named as **Send**. The text of the button is set to **SEND**.

The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Add a label with **Text Color** in black, **Size** 14, and set the **Text** of this label to **COMMAND:**
- Add an Edit Text boxes next to the label and name it as **Send**, set its **Style** to Bold, **Size** to 12, and **Text Color** to black.
- Add a button to the right hand side of the Edit Text box and name it as **Sendto**, set its Text to **SEND**
- Click **Tools -> Generate Members** and **Select All Views**. Then click button **Sendto** and enable **Click** so that the button responds to clicks. Exit after clicking **Generate Members**.
- Save the layout with the name **UDP2** and then close the window.

We can now develop the B4A code for the Android mobile phone. You should add the **Network** library to your project by clicking it at the **Libraries Manager** at the bottom right hand side of the screen.

Create variables **UdpSocket** and **UdpPacket** of types UDPSocket and UDPPacket respectively.

In subroutine **Activity_Create**, load layout **UDP2**, set the activity title to **LED CONTROL**, and initialize the UDP (UDP packets are not received in this program):

```
Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("UDP2")
  Activity.Title = "LED CONTROL"
  If FirstTime Then
    UdpSocket.Initialize("UDP", 2000, 1024)
  End If
End Sub
```

Enter the following code inside subroutine **Sento_Click** so that this code is executed whenever a command is entered into the Edit Text box and the **SEND** button is clicked. The entered command is sent to its destination (Raspberry Pi 3 computer) when the **SEND** button is clicked. Notice that the IP address of the Raspberry Pi computer is "192.168.1.164":

```
Sub Sendto_Click
  Dim MyData() As Byte
  MyData = Send.Text.GetBytes("UTF8")
  UdpPacket.Initialize(MyData, "192.168.1.164", 2000)
  UdpSocket.Send(UdpPacket)
End Sub
```

Figure 11.15 shows the Android screen when command **ON** is entered. The program listing (program: UDP2) is shown in Figure 11.16.

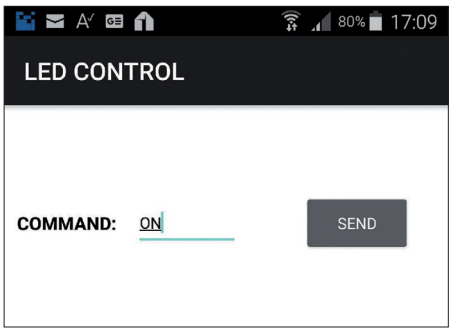


Figure 11.15 Android screen

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  Dim UdpSocket As UDPSocket
  Dim UdpPacket As UDPPacket
End Sub

Sub Globals
```

```
  Private Label1 As Label
  Private Sendto As Button
  Private Send As EditText
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("UDP2")
  Activity.Title = "LED CONTROL"
  If FirstTime Then
    UdpSocket.Initialize("UDP", 2000, 1024)
  End If
End Sub

Sub Sendto_Click
  Dim MyData() As Byte
  MyData = Send.Text.GetBytes("UTF8")
  UdpPacket.Initialize(MyData, "192.168.1.164", 2000)
  UdpSocket.Send(UdpPacket)
End Sub

Sub UDP_PacketArrived(Packets As UDPPacket)
End Sub


Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub
```

Figure 11.16 Program listing

11.3.7 Raspberry Pi Program

The Raspberry Pi program (program: UDPLED.py) is written in Python and its listing is shown in Figure 11.17. At the beginning of the program, libraries RPi.GPIO, socket, and sys are imported to the program. GPIO2 pin is then configured as an output, UDP port number is set to 2000. A UDP socket is created and bound to the specified IP address and port. The LED is then turned OFF at the beginning of the program. The remainder of the program is executed in an endless loop formed with a **while** statement. Inside this loop the program waits until it receives a packet (i.e. a command) from the mobile phone. When a packet is received, the program checks the contents of the packet. If the packet contains the word **ON** then the LED is turned ON. Similarly, if the packet contains the word **OFF** then the LED is turned OFF. If on the other hand the packet contains the **#** character then the network is closed and the program terminates.

```
#-----------------------------------------------------------------------
#--
#                LED CONTROL
#                ===========
#
# In this project a small LED is connected to GPIO 2 of
# the Raspberry Pi 3. The program receives UDP packets
# from an Android mobile phone to control the LED. Valid
# commands are ON and OFF to turn the LED ON and OFF
# respectively. Command # closes the network and exits
# from the program
#
#
# Program: UDPLED.py
# Date   : June 2018
# Author : Dogan Ibrahim
#-----------------------------------------------------------------------
import RPi.GPIO as GPIO     # import GPIO library
import socket
import sys
GPIO.setwarnings(False)     # disable warning messages

GPIO.setmode(GPIO.BCM)      # set BCM pin numbering
GPIO.setup(2, GPIO.OUT)     # configure GPIO 2 as output
UDP_PORT = 2000             # port number
UDP_IP = ''

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
sock.bind((UDP_IP, UDP_PORT))
GPIO.output(2, 0)           # LED OFF to start with

#
# Receive a packet with the command in it and then decode the command.
# valid commands are: ON, OFF, #
#
while True:
  data,addr = sock.recvfrom(1024)
  if data[0] == 'O' and data[1] == 'N':
      GPIO.output(2, 1)
  elif data[0] == 'O' and data[1] == 'F' and data[2] == 'F':
      GPIO.output(2,0)
  elif data[0] == '#':
      sock.close()
      sys.exit()
```

Figure 11.17 The Raspberry Pi program

11.4 PROJECT 19 – Displaying the Temperature on the Mobile Phone

11.4.1 Description

In this project the ambient temperature is read by the Raspberry Pi 3 and is sent as an UDP packet to the Android mobile phone which then displays the temperature on the screen in an Edit Text box every second. The **Sense HAT** sensor board is used with the Raspberry Pi 3 to read the ambient temperature.

11.4.2 Aim

The aim of this project is to show how the temperature can be read by the **Sense HAT** board connected to the Raspberry Pi 3, and also how this data can be sent and then displayed on the Android mobile phone.

11.4.3 Block Diagram

Figure 11.18 shows the block diagram of the project. The Sense HAT board is plugged on top of the Raspberry Pi 3 computer. The communication between the Raspberry Pi 3 and the Android mobile phone is via the local Wi-Fi router, using the UDP protocol.

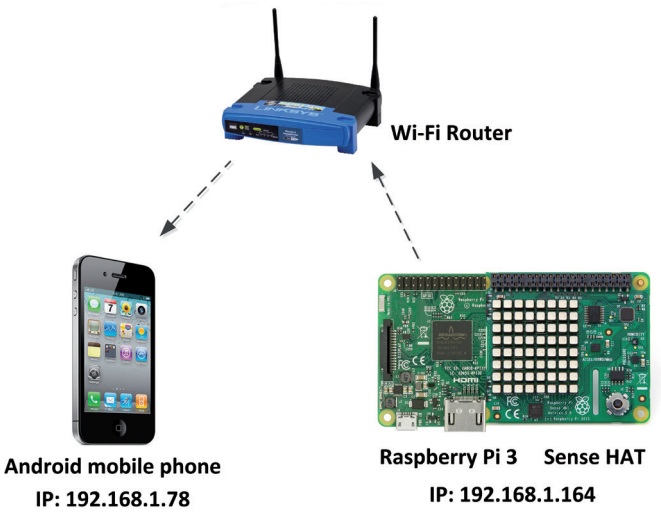


Figure 11.18 Block diagram of the project

11.4.4 The Sense HAT Board

It is worthwhile to look at the details of the Sense HAT board before using it in a project with the Raspberry Pi 3 computer.

The Sense HAT is an add-on board for the Raspberry Pi containing a number of useful sensors and an LED array. HAT is an acronym for **H**ardware **A**ttached on **T**op. Sense HAT was an important component of the Astro Pi project, which was an educational Raspberry Pi sent to the International Space Station with the British astronaut Tim Peake to run code developed by children. The actual Astro Pi had some modifications and had metal casing to make it suitable for use in space.

Sense HAT includes sensors to measure temperature, humidity, pressure, accelerometer, gyroscope, and a magnetometer. In addition, an 8 x 8 independently programmable LED array is included on the board that can be programmed to display text and small images. Figure 11.19 shows the Sense HAT board. We can identify the following components on the board:

- 8 x 8 LED array, having 15-bit colour resolution
- One chip containing accelerometer, gyroscope, and magnetometer to measure speed, orientation and the strength and direction of a magnetic field
- One chip containing temperature and humidity sensor
- Barometric pressure sensor chip capable of measuring the pressure exerted by small air molecules
- Graphics controller chip
- Five-button joystick with left, right, up, down, and enter movements

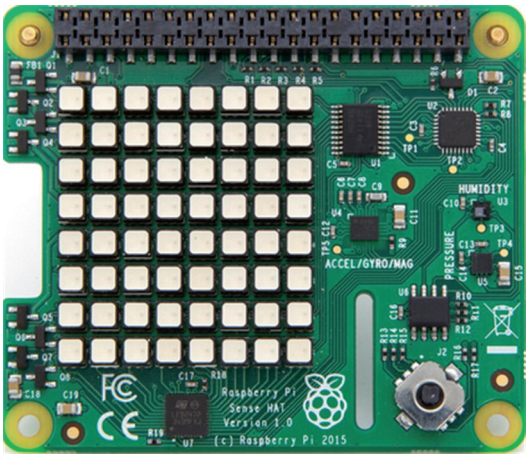


Figure 11.19 Sense HAT board

The Sense HAT library must be imported into your Python program and also the **sense** object must be created at the beginning of the program. i.e. the following two statements must be included at the beginning of your programs:

```
from sense_hat import SenseHAT
sense = SenseHat()
```

Reading the Temperature, Pressure, and Humidity

The following statements can be used to read the ambient temperature, pressure, and the humidity from the Sense HAT board:

```
Temp = sense.get_temperature()          - temperature in degrees C
Pressure = sense.get_pressure()         - pressure in millibars
Humidity = sense.get_humidity()         - relative humidity as %
```

The temperature, humidity, and pressure readings can contain several digits after the decimal point. In many applications we may want to round the readings to integers or to one or two decimal places.

By default, the statement **get_temperature** reads the temperature from the humidity sensor. Notice that it is also possible to read the temperature from the pressure sensor:

or, to read the temperature from the pressure sensor:

```
Temp = sense.get_temperature_from_pressure()
```

In the following program code the ambient temperature, humidity, and pressure are read and displayed on the monitor every second:

```
from sense_hat import SenseHat
import time
sense = SenseHat()
while True:
     T = sense.get_temperature()
     H = sense.get_humidity()
     P = sense.get_pressure()
     print("Temperature: %s, Humidity: %s, Pressure:%s" %(T,H,P))
     time.sleep(1)
```

Figure 11.20 shows the output of the program.

```
Temperature: 25.0324363708,Humidity: 42.6796798706, Pressure: 997.669677734
Temperature: 24.9414024353,Humidity: 43.2469673157, Pressure: 997.675537109
Temperature: 24.9778156281,Humidity: 43.1847038269, Pressure: 997.656005859
Temperature: 25.0142288208,Humidity: 43.5582809448, Pressure: 997.68359375
Temperature: 25.0324363708,Humidity: 43.001373291, Pressure: 997.670166016
Temperature: 24.9049873352,Humidity: 43.319606781, Pressure: 997.701171875
Temperature: 24.959608078,Humidity: 43.4026260376, Pressure: 997.677978516
Temperature: 25.0142288208,Humidity: 43.3784103394, Pressure: 997.663574219
```

Figure 11.20 Displaying the temperature, humidity, and pressure

We can round the displayed data for example using the Python **round** function as shown in the following program code. The output of this program is shown in Figure 11.21:

```
from sense_hat import SenseHat
import time
sense = SenseHat()
while True:
    T = sense.get_temperature()
```

```
H = sense.get_humidity()
P = sense.get_pressure()
TT = round(T, 1)
HH = round(H, 1)
PP = round(P, 1)
print("Temperature: %s, Humidity: %s, Pressure:%s" %(TT,HH,PP))
time.sleep(1)
```

```
Temperature: 24.9, Humidity: 43.3, Pressure: 997.8
Temperature: 24.9, Humidity: 43.4, Pressure: 997.8
Temperature: 25.0, Humidity: 43.2, Pressure: 997.8
Temperature: 25.0, Humidity: 43.4, Pressure: 997.8
Temperature: 25.0, Humidity: 43.4, Pressure: 997.8
Temperature: 25.1, Humidity: 43.3, Pressure: 997.9
Temperature: 25.0, Humidity: 43.6, Pressure: 997.8
Temperature: 25.0, Humidity: 43.3, Pressure: 997.9
```

Figure 11.21 Rounding the displayed data

11.4.5 Android Program

The B4A program consists of a label and an Edit Text box. The label displays the text **TEMPERATURE:**. The Edit Text box is named **Temperature** and it displays the temperature received from the Raspberry Pi.
The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Add a label with **Text Color** in black, **Size** 14, and set the **Text** of this label to **TEMPERATURE:**
- Add an Edit Text boxes next to the label and name it as **Temperature**, set its **Style** to Bold, **Size** to 14, and **Text Color** to black.
- Click **Tools -> Generate Members** and **Select All Views**. Exit after clicking **Generate Members**.
- Save the layout with the name **UDP3** and then close the window.

We can now develop the B4A code for the Android mobile phone. You should add the **Network** library to your project by clicking it at the **Libraries Manager** at the bottom right hand side of the screen.

Create variables **UdpSocket** and **UdpPacket** of types UDPSocket and UDPPacket respectively.

In subroutine **Activity_Create**, load layout **UDP3**, set the activity title to **TEMPERATURE**, and initialize the UDP so that packets are received automatically by subroutine called **UDP_PacketArrived**:

```
Sub Activity_Create(FirstTime As Boolean)
    Activity.LoadLayout("UDP3")
    Activity.Title = "TEMPERATURE"
    If FirstTime Then
        UdpSocket.Initialize("UDP", 2000, 1024)
    End If
End Sub
```

Enter the following code to the **PacketArrived** subroutine to display the temperature in the Edit Text box:

```
Sub UDP_PacketArrived(Packets As UDPPacket)
    Dim msg As String
    msg = BytesToString(Packets.Data, Packets.Offset, Packets.Length, "UTF8")
    Temperature.Text = msg
End Sub
```

Figure 11.22 shows the Android screen displaying the ambient temperature. The program listing (program: UDP3) is shown in Figure 11.23.

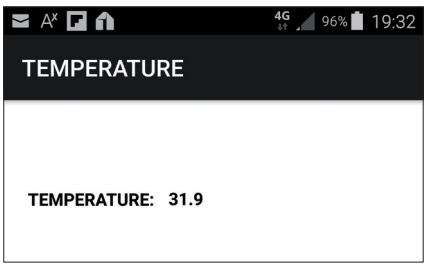


Figure 11.22 Android screen displaying the temperature

```
#Region  Project Attributes

#Region  Activity Attributes
```

```
Sub Process_Globals
    Dim UdpSocket As UDPSocket
    Dim UdpPacket As UDPPacket
End Sub

Sub Globals
    Private Label1 As Label
    Private Temperature As EditText
End Sub

Sub Activity_Create(FirstTime As Boolean)
    Activity.LoadLayout("UDP3")
    Activity.Title = "TEMPERATURE"
    If FirstTime Then
      UdpSocket.Initialize("UDP", 2000, 1024)
    End If
End Sub

Sub UDP_PacketArrived(Packets As UDPPacket)
    Dim msg As String
    msg = BytesToString(Packets.Data, Packets.Offset, Packets.Length, "UTF8")
    Temperature.Text = msg
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub
```

Figure 11.23 Program listing

11.4.6 Raspberry Pi Program

The Raspberry Pi program (program: UDPTEMP.py) is written in Python and its listing is shown in Figure 11.24. At the beginning of the program, libraries socket, time, and sense_hat are imported to the program. UDP IP and the UDP port number are set to "192.168.1.78" (IP address of the mobile phone) and to 2000 respectively. A UDP socket is then created. The remainder of the program is executed in an endless loop formed with a **while** statement. Inside this loop the program reads the ambient temperature from the Sense HAT board, rounds it, and then sends the reading to the mobile phone as an UDP packet. This process is repeated every second where the mobile phone displays the temperature.

```
#------------------------------------------------------------
#                   AMBIENT TEMPERATURE
#                   -------------------
#
# In this project the Sense HAT is connected to the
# Raspberry Pi 3. The ambient temperature is read
# and is sent to the Android mobile phone over a
# Wi-Fi link using the UDP protocol. The Android
# mobile phone displays the temperature every second
#
#
# Author: Dogan Ibrahim
# File  : UDPTEMP.py
# Date  : June, 2018
#------------------------------------------------------------
import socket
import time
from sense_hat import SenseHat
sense = SenseHat()

#
# Network code. Create a UDP socket on port 2000
#
UDP_PORT = 2000             # UDP port no
UDP_IP = "192.168.1.78"     # Mobile phone IP
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


#
# Start of main program loop. Send the temperature to
# the Android mobile phone every second
#
while True:
  T = sense.get_temperature()   # Read temperature
  T = round(T, 1)               # Round temperature
  sock.sendto(str(T), (UDP_IP, UDP_PORT))
  time.sleep(1)                 # Wait one second
```

Figure 11.24 The Raspberry Pi program

Chapter 12 • Android to Raspberry PI 3 SMS interface

12.1 Overview

In this Chapter we shall be looking at how an Android device (mobile phone or tablet) can communicate with a Raspberry Pi 3 computer through SMS text messages. In this Chapter the SIM800 GSM/GPRS shield will be used to give the SMS capability to a Raspberry Pi computer.

Before going into the details of the project it is worthwhile to look at briefly the specifications of the SIM800C shield.

12.2 The SIM800C Shield

SIM800C GSM/GPRS is a quad-band modem is customized for Raspberry Pi interface. The modem is controlled with the standard AT commands over a serial link.
SIM800C has the following basic features:

- Raspberry Pi compatible
- Quad-band in the range 850/900/1800/1900 MHz
- Speaker and microphone sockets
- Micro USB connector
- Grove interface
- 5V 2A (external) power, range: 5-20V
- 9600 baud rate (default), range: 1200-115200
- SIM card slot

There are several models of the SIM800C shield and Figure 12.1 shows one such model. The shield is supplied with an antenna. A SIM card slot is provided on the board. The shield can be directly mounted on top of a Raspberry Pi. Power to the shield must be supplied externally through the micro USB socket using a 5V (2.5A) power supply, or by connecting the power supply to the two left hand pins on the yellow header (J1) at the top left corner of the board (Figure 12.2). The pin at the left is the external +5V, and the one next to it is the GND pin. The power supply can power both the Raspberry Pi and the shield. A microphone and a speaker socket are provided on the board. Additionally, one Grove interface is also available on the board.

Before using the shield make sure that you insert a suitable SIM card to the slot, and also connect the antenna to the provided socket.

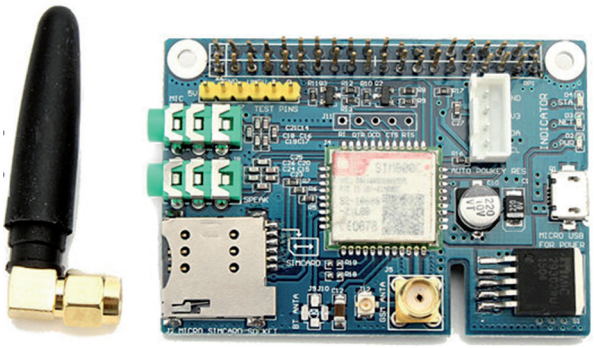


Figure 12.1 SIM800C GSM/GPRS shield

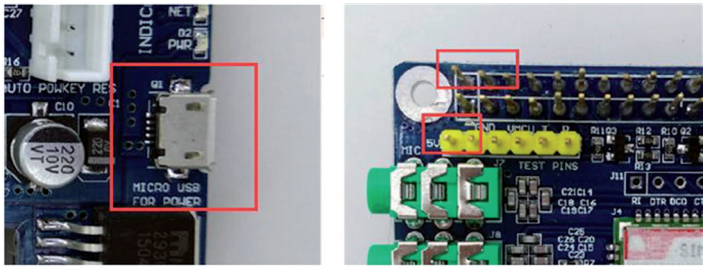


Figure 12.2 Power connectors on the board

The yellow header (J1) has the following pins:

Pin	Function
1	+5V
2	GND
3	VMCU
4	Not used
5	TXD
6	RXD

VMCU is used to provide the correct voltage to the UART pins. This pin should be connected to +3.3V for interfacing to a Raspberry Pi or STM32 processors where the UART is 3.3V compatible. Connect the pin to +5V for interfacing to an Arduino where the UART is 5V compatible.

SIM800 has three LED indicators on-board. The Power LED (PWR) is ON when power is applied to the board. The Status LED (STA) is also ON when SIM800C works normally. The Network LED (NET) should be ON for 64ms and OFF for 300ms when the modem communicates. It is 64ms ON and 800ms OFF when the modem does not find a network. Figure 12.3 shows the board layout of the shield used in this project.

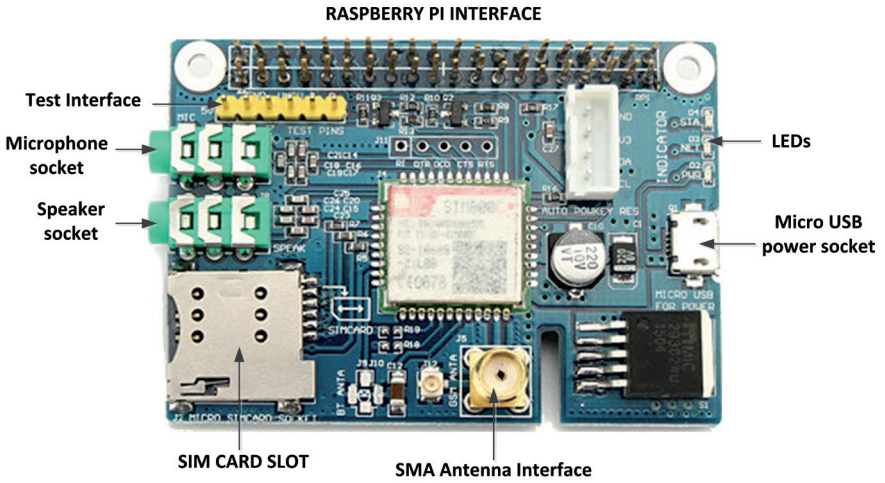


Figure 12.3 Board layout of the shield

The shield has a set of UART TTL ports for communication over the serial link. There are two sets of connectors for the UART on the board (see Figure 12.4) where there is no difference between them (see the **SIM800C shield-V1.2 for Raspberry Pi 2 user manual V1.0** for more details).

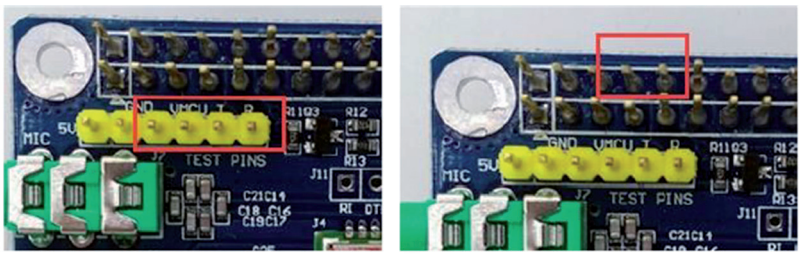


Figure 12.4 Serial ports on the board

The hardware connections between the Raspberry Pi and the SIM800C shield are through the UART port as follows:

SIM800C	**Raspberry Pi**
T	RXD
R	TXD
GND	GND

The SIM800C board can be controlled by AT commands where the commands must be terminated by the carriage-return character (i.e. \r in Python language). A test program (SSCOM.32) is available from the manufacturers to test the board.

It is important to note that you should have a valid SIM card installed in your SIM800C board before the board can be used in GSM/GPRS projects.

12.3 PROJECT 20 – Controlling a Relay on Raspberry Pi 3 by SMS Messages

12.3.1 Description

In this project a relay is connected to one of the digital input-output ports of the Raspberry Pi 3 computer. The relay is controlled remotely from an Android mobile phone by sending SMS text messages to activate and de-activate the relay.

12.3.2 Aim

The aim of this project is to show how the Android mobile phone can communicate with a Raspberry Pi using SMS messages.

12.3.3 Block Diagram

Figure 12.5 shows the block diagram of the project. The SIM800 GSM/GPRS shield is plugged on top of the Raspberry Pi 3. A relay is connected to one of the digital ports of the Raspberry Pi 3.

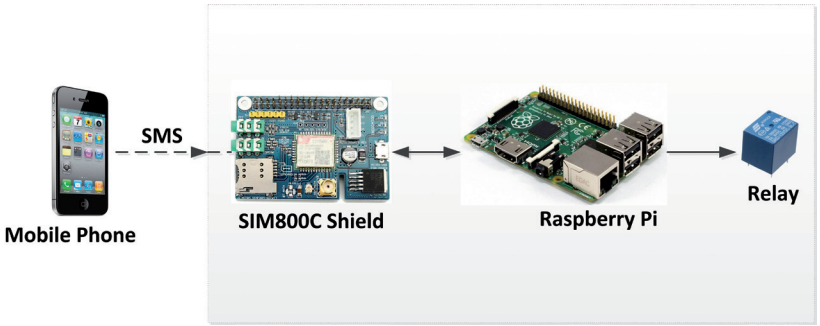


Figure 12.5 Block diagram of the project

12.3.4 Circuit Diagram

Figure 12.6 shows the circuit diagram of the project. In this project a relay module is used with built-in diode and transistor. The relay **S** pin is connected to digital port pin 3 (GPIO 2) of the Raspberry Pi 3. Notice that power to the SIM800C module is supplied externally through the USB port using a +5V, 2.5A power supply. This is necessary since the +5V output pin of the Raspberry Pi cannot provide enough current to drive the SIM800C module. SIM800C can either plugged on top of the Raspberry Pi 3, or the SIM800C **T** (transmit) and **R** (receive) pins can be connected to the Raspberry Pi 3 **RXD** (pin 10) and **TXD** (pin 8) pins respectively. Additionally, the **GND** pins of both boards should also be connected together and the VMCU pin of SIM800C should be connected to +3.3V (pin 1) of the Raspberry Pi 3.

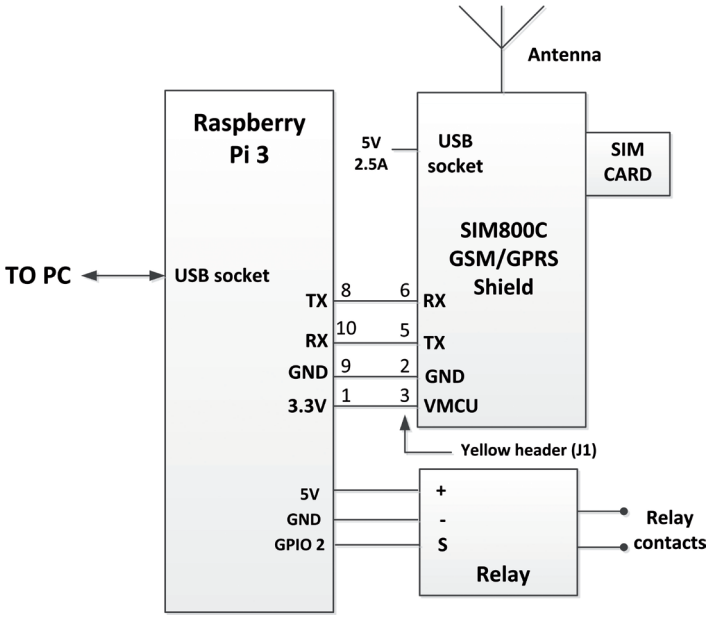


Figure 12.6 Circuit diagram of the project

Note that when the SIM800C board is plugged on top of the Raspberry Pi 3 Model B, there is no space to make connection to the USB power socket of the SIM800C. Therefore, it is recommended not to plug the SIM800C on top of the Raspberry Pi 3 Model B, but to power the SIM800C using a USB power supply, and then to make connections to the Raspberry Pi 3 using jumper wires. The relay can then be connected to the Raspberry Pi 3 via a bread-board.

12.3.5 Android Program

The Android program is exactly same as the one given in Figure 14.9 (see program SMS-RELAY in Chapter 14). The program consists of a label, a Text Box, and a button. The label displays the text **COMMAND:**. The user command is entered into the Text Box named **Command**, and is sent to the Arduino Uno when the **SEND** button is clicked. Valid commands are **ON** and **OFF** to activate and de-activate the relay respectively. You should include the **Phone** and the **RuntimePermissions** libraries in the program by selecting from the **Libraries Manager**. Inside the **Activity_Create**, load the layout and set the activity text to **SMS RELAY CONTROL**, and enable the runtime permissions for sending SMS.

12.3.6 Raspberry Pi 3 Program

The Raspberry Pi 3 program has been written using the Python programming language. First of all, the UART interface must be enabled on the Raspberry Pi 3. The steps for this are:

- Use the nano editor and edit file /boot/config.txt

  @raspberrypi:~$ **sudo nano /boot/config.txt**

- Add the following line to the bottom of the file:

```
#Enable UART
enable_uart=1
```

The serial port on the Raspberry Pi 3 is called serial0 (or ttyS0) when the Bluetooth interface is disabled. This serial port is available through pins 8 and 10 of the Raspberry Pi.

The Raspberry Pi 3 program listing is shown in Figure 12.7 (program: sms.py). At the beginning of the program the RPi.GPIO, serial, and time libraries are imported to the program. The program then sets up the serial port with baud rate 9600 and with the name sim800. Function **StartUp** is then called to configure port GPIO2 as output and also to de-activate the relay at the beginning of the program. The modem is configured in SMS text mode by sending the AT command **AT+CMGF=1**, and all the previous SMSs are deleted by the command **AT+CMGDA=DEL ALL**. The program then clears the receive buffer with the statement **response = sim800.read(sim800.inWaiting()).** Here, **sim800.inWaiting()** waits to receive bytes from the serial port and it returns the number of bytes read from the serial port. **sim800.read** then reads these number of bytes and stores them in variable called **response**. The remainder of the program is executed in an endless loop. Here, the program waits to receive an SMS text message. When a message is detected, command **AT+CMGR=1** is sent to the modem to read the received message. The message is stored in variable called **response**. If the received message contains word **ON** (i.e. **response.find("ON")** is greater than zero) then the relay is activated. Similarly, if the message contains word **OFF** then the relay is de-activated. The program then deletes all the messages and the above loop is repeated with the program waiting to receive a new message.

It is interesting to note that a received message has the following format (when the ON command is sent by the Android mobile phone). Here, the mobile phone number of the SIM800C modem, the current date, the time, and the command (**ON)** are returned, followed by the word **OK**.

```
+CMGR: "REC UNREAD" , "+447415987053","","18/07/03",11:14:18:04"
ON

OK
```

```
#-------------------------------------------------------------
#
#             RELAY CONTROL WITH SMS MESSAGE
#             ==============================
#
# In this program a relay is connected to port GPIO2 (pin 3)
# of the Raspberry Pi 3 computer. Additionally, an SMS800C
# GSM/GPRS modem shield is connected to the Raspberry Pi,
# where the TX and RX pins of the SIM800C are connected to
# the RX and TX pins of the Raspberry Pi respectively. In
```

```
# addition the GND pins of both boards are connected together.
# Power to the SIM800C board is supplied externally using a
# 5V, 2.5A power supply. Valid command are ON and OFF to
# activate and de-activate the relay
#
#
# Program: sms.py
# Date   : July 2018
# Author : Dogan Ibrahim
#-----------------------------------------------------------------
import RPi.GPIO as GPIO
import serial
import time

#
# This function configures the relay port as an output
# The relay is de-activated at the beginning of the program
#
def StartUp():
   GPIO.setmode(GPIO.BCM)
   GPIO.setwarnings(False)
   GPIO.setup(2, GPIO.OUT)
   GPIO.output(2, GPIO.LOW)

# Define the serial port to be used. On Raspberry Pi 3, ttyS0
# is the serial port, also called serial0. This port is at GPIO
# pin 8 (TX) and pin 10 (RX). Set the baud rate to 9600 bps
#
SerialPort = "/dev/ttyS0"                  # UART port
sim800 = serial.Serial(SerialPort,baudrate=9600,timeout=5)
StartUp()                                  # Configure ports

time.sleep(1)
sim800.write("AT+CMGF=1\r")                # Set text mode
time.sleep(1)
sim800.write('AT+CMGDA="DEL ALL"\r')       # Delete all SMS
time.sleep(5)
response = sim800.read(sim800.inWaiting()) # Clear the buffer

#
# Program loop. Loop here to look for SMS messages
#
while True:
  response = sim800.read(sim800.inWaiting()) # Look for SMS
  if response != "":                         # SMS received
    sim800.write("AT+CMGR=1\r")              # Issue to read SMS
```

```
time.sleep(1)
response=sim800.read(sim800.inWaiting()) # Read the SMS

if response.find("ON") > 0:     # If ON command
  GPIO.output(2, GPIO.HIGH)     # Activate relay
if response.find("OFF") > 0:    # If OFF command
  GPIO.output(2, GPIO.LOW)      # De-activate relay

time.sleep(0.5)
sim800.write('AT+CMGDA="DEL ALL"\r')     # Delete ALL SMSs
time.sleep(1)
response=sim800.read(sim800.inWaiting()) # Clear the buffer
time.sleep(1)
```

Figure 12.7 Raspberry Pi 3 program

Chapter 13 • Android to Arduino WI-FI interface

13.1 Overview

In this Chapter we shall be looking at how an Android device (mobile phone or tablet) can communicate with an Arduino Uno computer over a Wi-Fi link using the UDP protocol.
Before looking at the details of the Android to Arduino Uno interface, it is worthwhile to review some basic features of the Arduino Uno for those readers who may be new to the world of Arduino.

13.2 The Arduino Uno

The Arduino Uno is perhaps currently the most popular single board computer around. Arduino is an open-source platform supported by large number of libraries, thus making it very easy to develop projects. It is low-cost and has been used in thousands of simple as well as complex projects over the years by many people. There is vast amount of information on the architecture and programming of the Arduino family of computers on the Internet. Many books have also been published to teach the use of the Arduino boards in various projects. Some of the reasons why the Arduino family has been popular over the years are: low-cost, simple programming environment, simple up-loading the compiled code to the target board, open-source, large numbers of freely available libraries, extensible libraries, and vast amount of tutorials, data sheets, projects, and code examples on the Internet.
In this Chapter we shall be using the Arduino Uno board in Android based projects. Arduino Uno is the entry level board. Figure 13.1 shows the Arduino Uno board. Some details on the Arduino Uno board are given in this section.

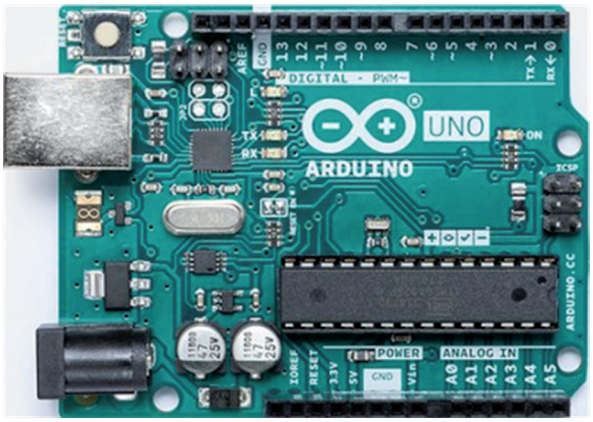


Figure 13.1 The Arduino Uno board

Arduino Uno Features

Microcontroller:	ATmega328P
Clock Speed:	16MHz
Operating Voltage:	5V
Input Voltage:	7 – 12V
Digital I/O pins:	14 (6 PWM outputs)

Analog inputs:	6
I/O pin DC current:	20mA
Flash Memory:	32KB
SRAM:	2KB
EEPROM:	1KB
Built-in LED:	At I/O pin 13
Length:	68.6mm
Width:	53.4mm

Arduino Uno is based on the ATmega328P microcontroller operating at 16MHz. It has 14 digital input-output pins and 6 analog inputs (ADC). 6 digital input-output pins can be used as PWM outputs. Connection to the host computer is via standard USB cable. A power jack socket, an ICSP header, and a reset button are provided on the board. Arduino Uno is the entry level board of the Arduino family. Interested readers can search the Internet for other more complex Arduino boards.

Arduino Uno Pin Configuration

Figure 13.2 shows the Arduino Uno pin configuration. The digital input-output pins are located at the right hand side of the board, labelled as 0 to 13 and 18, 19. Some pins have multiple functions. For example, input-output pin 1 is also the UART TX pin and so on. The analog inputs are located at the bottom left hand side of the board and are labelled as A0 to A5. The other pins at this side are: Vin, GND, +5V, +3.3V, RESET, and IOREF. An LED is connected to digital output pin 13 of the board that can be used for test purposes. It is important to note that the Arduino Uno input-output pins are +5V compatible and care should be taken when using the Arduino Uno in +3.3V interface applications. For example, an output pin of the Arduino Uno should not be connected directly to a circuit whose maximum allowable input voltage is +3.3V. Either resistive voltage divider circuits or +5V to +3.3V converter circuits should be used in such applications.

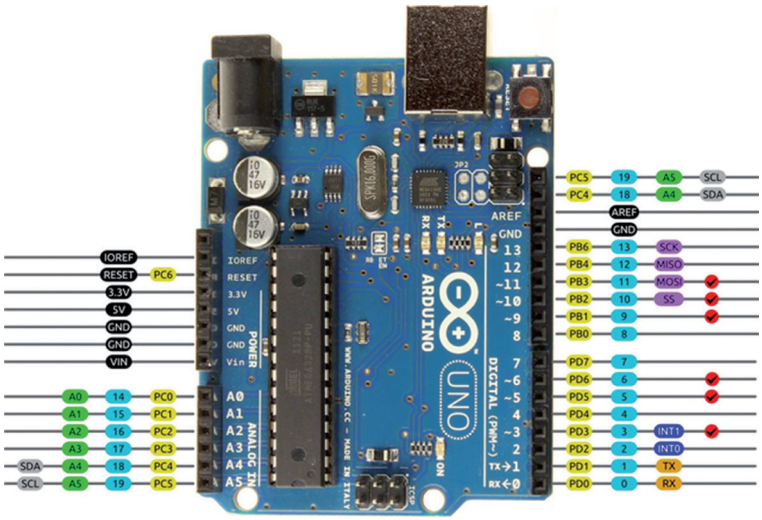


Figure 13.2 Arduino Uno pin configuration

13.3 PROJECT 21 – Controlling an LED on the Arduino Uno

13.3.1 Description

In this project the LED on the Arduino Uno board is controlled remotely from an Android mobile phone over a Wi-Fi link using the UDP protocol. As in Project 18 in Chapter 11, Android commands **ON** and **OFF** turn the LED ON and OFF respectively. Command **#** terminates the program on the Arduino.

13.3.2 Aim

The aim of this project is to show how the Android mobile phone can communicate with an Arduino Uno computer over a Wi-Fi link using the UDP protocol.

13.3.3 Block Diagram

Figure 13.3 shows the block diagram of the project. The communication between the Arduino Uno and the Android mobile phone is via the local Wi-Fi router, using the UDP protocol. The Arduino Uno has no built-in Wi-Fi capability. In this project the ESP-01 low-cost tiny Wi-Fi board is used to provide the Wi-Fi capability to the Arduino Uno board.

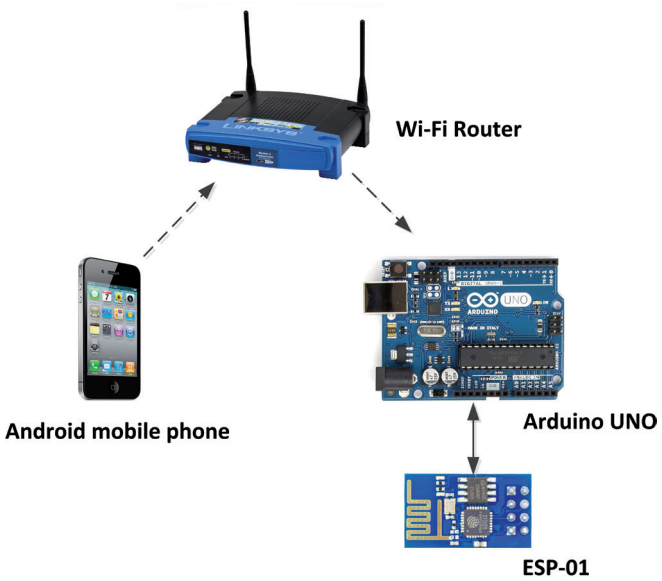


Figure 13.3 Block diagram of the project

13.3.4 Circuit Diagram

In this project an ESP-01 type Wi-Fi module with the ESP8266 processor on-board is used. The ESP-01 is connected to the Arduino Uno so that the Arduino Uno can communicate over the Wi-Fi link. Figure 13.4 shows the ESP-01 module. This is a very low-cost small Wi-Fi enabled module that has an on-board processor and can either be used as a standalone processor with limited I/O capability, or it can be connected to a host processor to give Wi-Fi capability to the host processor. In this project ESP-01 is used to give Wi-Fi capability to the Arduino Uno. This is accomplished by sending AT commands to the ESP-01 to configure

it as a Wi-Fi device and then to communicate with external world via the Wi-Fi link.

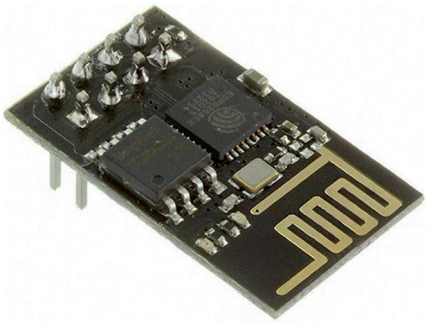


Figure 13.4 ESP-01 module

ESP-01 is a 3.3V compatible device and its I/O pins must not be raised to greater than +3.3V. The device has 8 pins:

VCC: Power supply pin. Must be connected to a +3.3V external power supply
GND: Power supply ground
GPIO0: I/O pin. This pin must be connected to +3.3V for normal operation, and to 0V for uploading firmware to the chip
GPIO2: I/O pin
RST: Reset pin. Must be connected to +3.3V for normal operation
CH_PD: Enable pin. Must be connected to +3.3V for normal operation
TX: Serial output pin
RX: Serial input pin

Figure 13.5 shows the circuit diagram of the project. It is important to use an external +3.3V power supply for the ESP-01 board since the Arduino +3.3V power supply cannot provide enough current for the ESP8266. In this project the LM1086-3.3 power regulator chip is used to provide power to the ESP-01 board. Although this regulator can be connected to the +5V output pin of the Arduino Uno, it is recommended to connect it to an external +5V power supply since the ESP-01 current consumption could be high and it could overload the Arduino. Additionally, the GPIO0, RST, and CHD_PD inputs of the ESP-01 are all connected to +3.3V (you may consider connecting the RST input through a switch so that this pin is connected to GND when the switch is pressed, thus enabling the device to be reset if required).

The output voltage of an Arduino pin is +5V and this is too high for the inputs of the ESP-01. In this project a resistor potential divider circuit is used to lower the serial output voltage of the Arduino Uno to +3.3V before it is connected to the RX input of ESP-01. The TX output of ESP-01 is connected to an input pin of the Arduino which is configured as a serial input by the software. Pins 9 and 10 of the Arduino Uno are used as the serial RX and TX pins respectively. ESP-01 is not breadboard compatible and you may be required to purchase an adapter if you are planning to mount it on a breadboard.

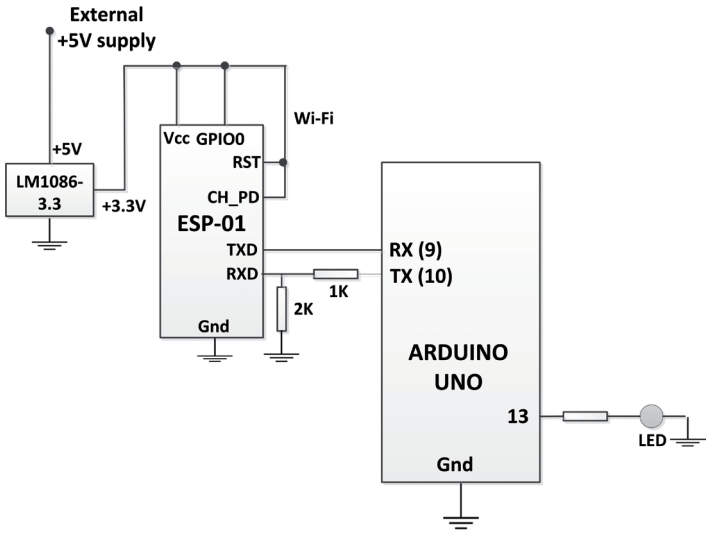


Figure 13.5 Circuit diagram of the project

13.3.5 Android Program

The Android program (program: UDP4) is very similar to the program given in Figure 11.16 (program: UDP2) where the commands **ON**, **OFF**, and **#** are entered into an Edit Text box on the Android mobile phone. In this program the commands are terminated with a dot character (".") and are then sent to the Arduino Uno. This is because the Arduino program reads the contents of a packet until a dot has been detected. i.e. the dot acts as the terminator character. Figure 13.6 shows the Android program listing.

```
#Region  Project Attributes

#Region  Activity Attributes

Sub Process_Globals
  Dim UdpSocket As UDPSocket
  Dim UdpPacket As UDPPacket
End Sub

Sub Globals
  Private Label1 As Label
  Private Sendto As Button
  Private Send As EditText
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("UDP4")
  Activity.Title = "ARDUINO LED CONTROL"
  If FirstTime Then
```

```
    UdpSocket.Initialize("UDP", 2000, 1024)
  End If
End Sub

Sub Sendto_Click
  Dim MyData() As Byte
  Dim msg As String
  msg = Send.Text & "."
  MyData = msg.GetBytes("UTF8")
  UdpPacket.Initialize(MyData, "192.168.1.160", 2000)
  UdpSocket.Send(UdpPacket)
End Sub

Sub UDP_PacketArrived(Packets As UDPPacket)
End Sub
```

Figure 13.6 Android program

13.3.6 Arduino Uno Program

The Arduino Uno program has been developed using the Arduino IDE. The program (LED-CONTROL) listing is shown in Figure 13.7. At the beginning of the program the **Software-Serial** library is included in the program, the interface between the Arduino and the ESP-01 are defined, and the LED is defined as 13 (i.e. port 13). Inside the setup routine the LED is turned OFF at the beginning of the program. Then the baud rate of the serial communication is defined as 115200 and the ESP-01 is reset.

The remainder of the program runs in an endless loop. Inside this loop the ESP-01 is configured and connected to the local Wi-Fi router. In this example, the name of the router is **BTHomeSpot-XNH** and its password is **49345abaab**, and these are setup by the AT command **CWJAP**. The AT command **CIPSTART** configures the ESP-01 to communicate via the UDP protocol where the port number is set to 2000.

When a packet arrives to the Arduino Uno, **esp01.available** becomes true. Array **cmd** is used to store the contents of the received packet. Function **readBytesUntil** reads the contents of the received packet until character dot has been received. If the packet contains the word **ON** then the LED is turned ON. Similarly, if the packet contains the word **OFF** then the LED is turned OFF. Command **#** terminates the Arduino Uno program so that it does not respond to UDP packets.

```
/*****************************************************************
 *              REMOTE LED CONTROL USING WI-FI
 *              ==============================
 *
 * In this project commands are received from an Android mobile
 * phone in order to control the built-in LED on the Arduino
 * board. Valid commads are ON, OFF, and #. Commands ON and
```

```
 * OFF turns the LED ON and OFF respectively. Command #
 * terminates the Arduino program.
 *
 * File: LEDCONTROL
 * Date: June, 2018
 * Author: Dogan Ibrahim
 ************************************************************/
#include <SoftwareSerial.h>         // Software serial library
SoftwareSerial esp01(9, 10);        // 9=RX, 10=TX
#define LED 13                      // LED on output port 13
int First = 1;
char cmd[10];


//
// Send a command to the ESP-01
//
void SendToEsp01(String cmd, const int tim)
{
  esp01.print(cmd);
  long int time = millis();
  while((time+tim) > millis())      // Wait required time
  {
    while(esp01.available())        // If response available
    {
      char ch = esp01.read();       // Read the response
    }
  }
}

//
// The Setup routine
//
void setup()
{
  pinMode(LED, OUTPUT);
  digitalWrite(LED, LOW);            // LED OFF to start with
  esp01.begin(115200);               // Baud rate
  SendToEsp01("AT+RST\r\n", 2000);   // Reset ESP-01
}

//
// Main program loop. Read the command from the ESP-01 through
// Wi-Fi, decode the command and implement the required control
// action
//
```

```
void loop()
{
  if(First == 1)
  {
    First = 0;
    SendToEsp01("AT+CWMODE=1\r\n", 5000);
    SendToEsp01("AT+CWJAP=\"BTHomeSpot-XNH\", \"49345abaab\"\r\n", 5000);
    SendToEsp01("AT+CIFSR\r\n", 3000);
    SendToEsp01("AT+CIPSTART=\"UDP\",\"0.0.0.0\",0,2000,2\r\n", 2000);
  }

  cmd[0] = 0;
  cmd[1] = 0;
  cmd[2] = 0;

  if(esp01.available())
  {
    esp01.readBytesUntil('.', cmd, 3);
    if(cmd[0] == 'O' && cmd[1] == 'N')
      digitalWrite(LED, HIGH);
    else if(cmd[0] == 'O' && cmd[1] == 'F' && cmd[2] == 'F')
      digitalWrite(LED, LOW);
    else if(cmd[0] == '#')
      exit(0);
  }
}
```

Figure 13.7 Arduino Uno program

Figure 13.8 shows the Android command to turn OFF the LED.

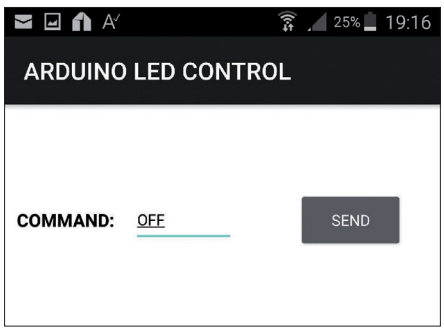


Figure 13.8 Command to turn OFF the LED

13.4 PROJECT 22 – Displaying the Temperature and Humidity

13.4.1 Description

In this project the ambient temperature and humidity are read by the Arduino Uno every second and are then sent to the Android mobile phone where they are displayed. The DHT11 sensor module is connected to the Arduino Uno to read the ambient temperature and the humidity.

13.4.2 Aim

The aim of this project is to show how the DHT11 sensor module can be used with the Arduino Uno to read the ambient temperature and the humidity, and also how to send this data to an Android mobile phone over a Wi-Fi link using the UDP protocol.

13.4.3 Block Diagram

Figure 13.9 shows the block diagram of the project. The communication between the Arduino Uno and the Android mobile phone is via the local Wi-Fi router, using the UDP protocol. In this project the ESP-01 Wi-Fi board is used to provide the Wi-Fi capability to the Arduino Uno board as in the previous project. The DHT11 sensor is connected to the Arduino Uno.

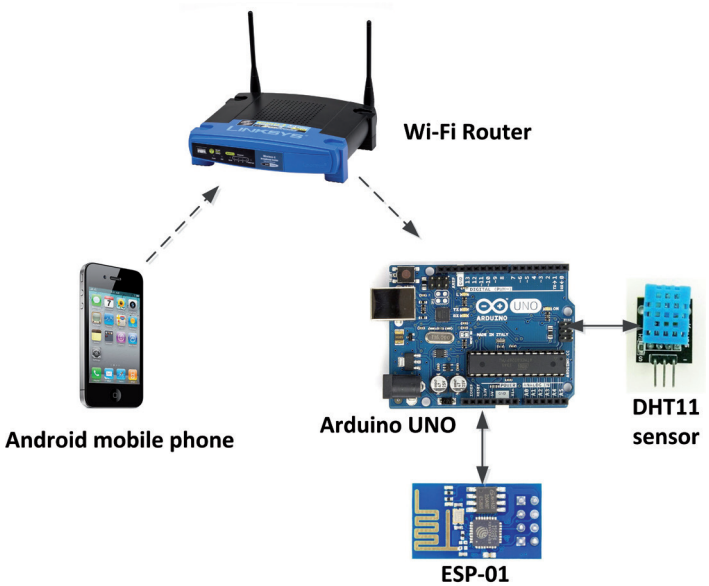


Figure 13.9 Block diagram of the project

13.4.4 Circuit Diagram

The ESP-01 is connected to the Arduino Uno as in the previous project. The standard DHT11 is a 4-pin digital output device (only 3 pins are used) as shown in Figure 13.10, having pins +V, GND, and Data. The Data pin must be pulled-up to +V through a 10K resistor. The chip uses capacitive humidity sensor and a thermistor to measure the ambient temperature. Data output is available from the chip around every second. The basic features of DHT11 are:

- 3 to 5V operation
- 2.5mA current consumption (during a conversion)
- Temperature reading in the range 0-50ºC with an accuracy of ±2ºC
- Humidity reading in the range 20-80% with 5% accuracy
- Breadboard compatible with 0.1 inch pin spacing

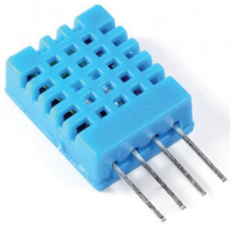


Figure 13.10 The standard DHT11 chip

In this example the DHT11 module with built-in 10K pull-up resistor, available from Elektor, is used. This is a 3-pin device with the pin layout as shown in Figure 13.11.

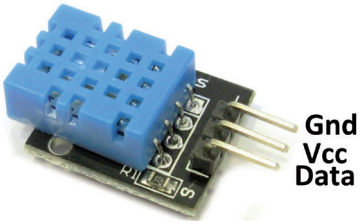


Figure 13.11 Elektor DHT11 module

Figure 13.12 shows the circuit diagram of the project. Here, the Data output of the DHT11 is connected to digital I/O pin 7 of the Arduino Uno.

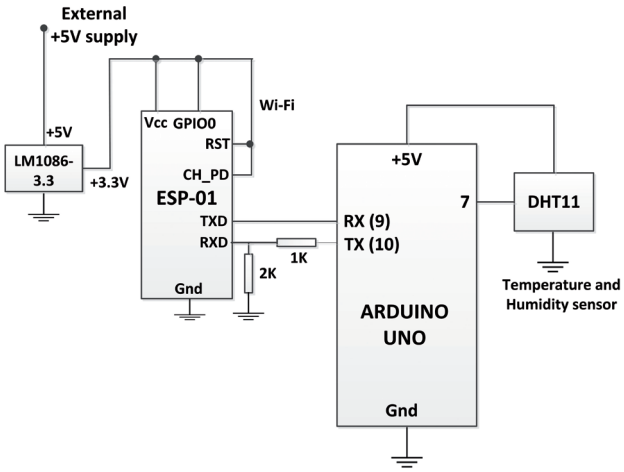


Figure 13.12 Circuit diagram of the project

13.4.5 Android Program

The B4A program consists of two labels and two Edit Text boxes. The labels display the text **TEMPERATURE:** and **HUMIDITY:**. The Edit Text boxes are named **Temperature** and Humidity and they display the temperature and humidity data received from the Arduino Uno. The steps to design the project are given below:

- Start the B4A
- Start the B4A-Bridge on your mobile phone and click START
- Connect to the mobile phone by clicking **Tools**-> **B4A Bridge** -> **Connect**
- Click **Designer** and then **Open Designer**
- Click **WYSIWYG** and then **Connect** to connect to the mobile device so that you can see the design on your mobile phone
- Set the **Color** in the **Activity Properties** to white
- Add two labels with **Text Colors** in black, **Sizes** 14, and set the **Texts** of these labels to **TEMPERATURE:** and **HUMIDITY:**
- Add two Edit Text boxes next to each label and name them as **Temperature** and **Humidity**, set their **Styles** to Bold, **Sizes** to 14, and **Text Colors** to black.
- Click **Tools -> Generate Members** and **Select All Views**. Exit after clicking **Generate Members**.
- Save the layout with the name **UDP5** and then close the window.

We can now develop the B4A code for the Android mobile phone. You should add the **Network** library to your project by clicking it at the **Libraries Manager** at the bottom right.

Insert the following code inside the **Activity_Create** to load layout UDP5, set the activity title to **TEMPERATURE AND HUMIDITY** and to initialize the UDP so that when a packet arrives the subroutine named **UDP_PacketArrived** is activated automatically:

```
Sub Activity_Create(FirstTime As Boolean)
   Activity.LoadLayout("UDP5")
   Activity.Title = "TEMPERATURE AND HUMIDITY"
   If FirstTime Then
     UdpSocket.Initialize("UDP", 2000, 1024)
   End If
End Sub
```

Create subroutine **UDP_PacketArrived** and insert the following code inside this subroutine. Here, received packet contents is stored in string variable **msg**. The program then

searches the comma (",") character inside this string where the position of the comma is stored in variable **p**. Functions **substring** are then used to extract the first part (before the comma) of the string which is the temperature and this value is displayed in Edit Text box called **Temperature**. The second part of the string (after the comma) is the humidity and is displayed in Edit Text box called **Humidity**:

```
Sub UDP_PacketArrived(Packets As UDPPacket)
    Dim msg As String
    Dim p As Int
    msg = BytesToString(Packets.Data, Packets.Offset, Packets.Length, "UTF8")
    p=msg.IndexOf(",")
    Humidity.Text = msg.SubString(p+1)
    Temperature.Text = msg.SubString2(0,p)
End Sub
```

Figure 13.13 shows the Android program listing (program: UDP5).

```
#Region  Project Attributes

#Region  Activity Attributes
 #FullScreen: False
 #IncludeTitle: True
#End Region

Sub Process_Globals
 Dim UdpSocket As UDPSocket
 Dim UdpPacket As UDPPacket
End Sub

Sub Globals
 Private Label1 As Label
 Private Temperature As EditText
 Private Humidity As EditText
 Private Label2 As Label
End Sub

Sub Activity_Create(FirstTime As Boolean)
 Activity.LoadLayout("UDP5")
 Activity.Title = "TEMPERATURE AND HUMIDITY"
 If FirstTime Then
   UdpSocket.Initialize("UDP", 2000, 1024)
 End If
End Sub

Sub UDP_PacketArrived(Packets As UDPPacket)
 Dim msg As String
```

```
  Dim p As Int
  msg = BytesToString(Packets.Data, Packets.Offset, Packets.Length, "UTF8")
  p=msg.IndexOf(",")
  Humidity.Text = msg.SubString(p+1)
  Temperature.Text = msg.SubString2(0,p)
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub
```

Figure 13.13 Android program

13.4.6 Arduino Uno Program

The Arduino Uno program uses a DHT11 library to read the temperature and humidity values from the sensor module. There are several such libraries available and the one used in this project is available from the following link:

http://www.circuitbasics.com/how-to-set-up-the-dht11-humidity-sensor-on-an-arduino/

Create a folder called **DHT11** under the Arduino libraries (e.g. under C:\Program Files (x86)\Arduino\libraries) and copy file **DHTLib.zip** file from the above link into this new folder. Then, start the Arduino IDE and click **Sketch -> Include Library** and select the **DHTLib.zip** file. We can now write our Arduino program.

Figure 13.14 shows the Arduino program listing (program: DHT11). At the beginning of the program libraries **SoftwareSerial** and **dht** are included in the program. **SoftwareSerial** is used for the serial communication with the ESP-01 Wi-Fi module. **dht** is used to read the temperature and the humidity from the DHT11 sensor module. Inside the **setup** routine the communication speed with the ESP-01 is initialized to 115200 baud and the device is reset. The remainder of the program is run inside an endless loop created using a **while** statement. Inside this loop the ESP-01 is connected to the local Wi-Fi router, and is configured to communicate using the UDP protocol. The IP address of the Android mobile phone ("192.168.1.78") and the UDP port number (2000) are defined. The program then reads the temperature and the humidity from the DHT11 sensor module (e.g. **DHT.temperature** and **DHT.humidity**). String variable **MyData** combines the temperature and the humidity such that the two are separated with a comma. Characters **C** and **%** are added to the end of the temperature and the humidity respectively (to indicate the Centigrade and relative humidity). The length of the packet (i.e. **MyData**) is then stored in variable **l**. Then, the size of the packet to be sent to the Android mobile phone is specified using the AT command **CIPSEND**. Finally, the packet is sent using a **print** statement. The above process is repeated forever after one second delay.

```
/*****************************************************************
 *              TEMPERATURE AND HUMIDITY
 *              ========================
 *
 * In this project a DHT11 type temperature and humidity sensor
 * is connected to port pin 7 of the Arduino Uno. The program
 * reads the ambient temperature and humidity and sends to the
 * Android every second which is displayed on the Android. The
 * data is sent over teh local Wi-Fi link using the UDP protocol
 *
 *
 * File: DHT11
 * Date: June, 2018
 * Author: Dogan Ibrahim
 *****************************************************************/

#include <SoftwareSerial.h>          // Software serial library
#include <dht.h>
SoftwareSerial esp01(9, 10);         // 9=RX, 10=TX
dht DHT;
#define DHT11_PIN 7                  // DHT11 pin

int First = 1;


//
// Send a command to the ESP-01
//
void SendToEsp01(String cmd, const int tim)
{
  esp01.print(cmd);
  long int time = millis();
  while((time+tim) > millis())       // Wait required time
  {
    while(esp01.available())        // If response available
    {
      char ch = esp01.read();       // Read the response
    }
  }
}

//
// The Setup routine
//
void setup()
{
```

```
  esp01.begin(115200);               // Baud rate
  SendToEsp01("AT+RST\r\n", 2000);   // Reset ESP-01
}

//
// Main program loop. Read the command from the ESP-01 through
// Wi-Fi, decode the command and implement the required control
// action
//
void loop()
{
  int rd;
  int Temperature, Humidity;
  String MyData;
  String MyLen;
  int l;

  if(First == 1)
  {
    First = 0;
    SendToEsp01("AT+CWMODE=1\r\n", 5000);
    SendToEsp01("AT+CWJAP=\"BTHomeSpot-XNH\", \"49345abaab\"\r\n", 5000);
    SendToEsp01("AT+CIFSR\r\n", 3000);
    SendToEsp01("AT+CIPSTART=\"UDP\",\"192.168.1.78\",2000,2000,2\r\n",
2000);
  }

  rd = DHT.read11(DHT11_PIN);
  Temperature = DHT.temperature;
  Humidity = DHT.humidity;
  MyData = String(Temperature) + " C," + String(Humidity) + " %";
  l = MyData.length();
  MyLen = String(l);
  SendToEsp01("AT+CIPSEND=",10);
  SendToEsp01(MyLen,10);
  SendToEsp01("\r\n",100);
  esp01.print(MyData);
  delay(1000);
}
```

Figure 13.14 Arduino program

Figure 13.15 shows the temperature and humidity displayed on the Android mobile phone screen.

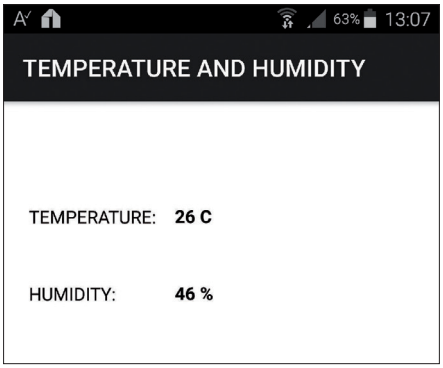


Figure 13.15 Android mobile phone screen

Chapter 14 • Android to Arduino SMS interface

14.1 Overview

In this Chapter we shall be developing a project to learn at how an Android device (mobile phone or tablet) can communicate with an Arduino Uno through SMS messages.

Before looking at the details of the project it is worthwhile to look at briefly the details of SMS messages and also how an GSM/GPRS shield can be used with an Arduino Uno to send and receive SMS messages.

14.2 SMS Messages

SMS (Short Message Service) is used to send and receive messages between mobile devices. It was first made available in 1992 and has been incorporated into the GSM standards and is currently supported by all GSM mobile phones. An SMS is short and it can contain up to 160 characters of data with 7-bit encoding (like English characters), and 70 characters of data if 16-bit Unicode encoding is used (like Chinese characters). In addition to text, binary data can also be sent and received using the SMS message services. Therefore, pictures, logos, ring tones and other binary data can be handled by SMS messages.

Some advantages of SMS messages are:

- SMS messages can be sent and received at any time of the day
- SMS messages can be sent to a phone which is offline. The GSM system will store the messages and send them to the correct phone when it becomes online
- SMS messages are supported by all mobile phones
- SMS messages can be sent and received in all languages
- Alerts and notifications, such as credit card transactions, stock market changes and so on can be sent using SMS messages
- Remote devices can be monitored and controlled using SMS messages
- SMS messages can be used as marketing and advertising tools
- SMS messages can be used as teaching tools

SMS messages are handled by the SMS center (SMSC). When an SMS is sent, it first reaches the SMS center where it is then forwarded to its destination if it is online, or it is stored if the destination device is offline and is delivered to the recipient when it becomes available. The network operator's SMS center number must be entered into a device before it can use the SMS services.

An SMS message is stored for a temporary period of time if the recipient device is offline. This time can be specified by the sender (called the validity period) where after this time the message will be deleted from the SMS center and the message will not be forwarded to its destination.

SMS messages can be sent to devices on the same network or on different networks. When different networks are used SMS Gateways are used to handle the messages. The messages can also be sent internationally. Messages can be sent and received in two formats using

the AT commands: by PDU (Protocol Description Unit) format, and TEXT format. The PDU format offers to send binary data in 7-bit or 8-bit mode and is helpful if we wish to send compressed data, or binary data. PDU format consists of hexadecimal string of characters, including the SMS service center address, sender number, time stamp etc. In Text format alphanumeric characters with up to 160 characters long with 7-bit encoding, and 140 characters long with 8-bit encoding can be sent and received.

14.2.1 Sending and Receiving in Text Mode

Sending

Sending an SMS in Text mode is very easy and an example is given in Figure 14.1 that shows how the message **Sending test SMS Message!** can be sent to mobile number **+447415987053**. It is assumed that the SMS service center number has already been loaded to the SIM card and that the password has been entered:

- Set the SMS format to Text:

```
AT+CMGF=1
```

- Set the character mode to GSM:

```
AT+CSCS = "GSM"
```

- Set the SMS parameters:

```
AT+CSMP = 17,167,0,241
```

- Set the recipient mobile phone number and the text message:

```
AT+CMGS="+447415987053"
>Sending test SMS Message![Ctrl-Z]
```

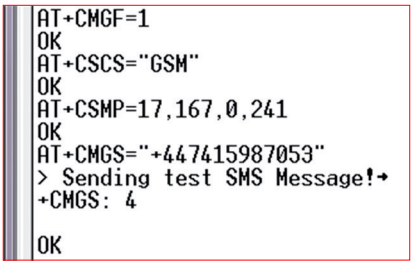


Figure 14.1 Sending an SMS text message

Notice that after entering the recipient phone number, the modem responds with character **>** and at this point the message to be sent must be entered. The message must be terminated with the **Cntrl-Z** character.

Let us look at the various commands used to send the message:

Command **AT+CMGF** selects the format of the message used with SMS messages. The command has a <**mode**> parameter which can take the following values:

<**mode**>
0 - PDU mode
1 - Text mode

Command **AT+CSCS** is used to set the character set to be used in text mode SMS messages. The valid character sets are:

"GSM" - GSM default alphabet
"IRA" - international reference alphabet
"8859-1" - ISO 8859 Latin 1 character set
"PCCP437" - PC character set
"UCS2" - 16-bit universal multiple-octet coded character set
"HEX" - Character strings consist only of hexadecimal numbers from 00 to FF

Command **AT+CSMP** is used to set various other parameters required before sending an SMS in text mode. This command has the following parameters:

<**fo**>,<**vp**>,<**pid**>,<**dcs**>

where,

<**fo**> - is set to 17 to indicate that the message is to go from a mobile device to a service center and the <vp> field is valid.

<**vp**> - selects the message validity period and a value between 144 to 167 selects the period as:

12 hrs + ((vp – 143) x 30 mins)

Thus, with a setting of 167, the message validity period is:

12 hrs + (167 – 143) x 30 mins = 24 hrs

<**pid**> - is the protocol identifier and the default is 0

<**dcs**> - is used to determine the way the information is encoded. This field is set to 241 which sets the message class to 1 and uses the default alphabet. Note that, setting the message class to 0 causes the message to be displayed immediately without being stored by the mobile phone (this is also called a flash message).

Receiving

The command **AT+CMGL** can be used to list all messages stored on the SIM card. As shown in Figure 14.2, entering command **AT+CMGL=?** Lists all the parameters of this command:

```
AT+CMGL=?
+CMGL: ("REC UNREAD","REC READ","STO UNSENT","STO SENT","ALL")

OK
```

Figure 14.2 Listing parameters of the AT+CMGL command

We can list all the received and read SMS messages on the SIM card by entering the command **AT+CMGL="REC READ".** We can also list the unread messages with the command **AT+CMGL="REC UNREAD".** Similarly, all the messages read and unread can be displayed with the command **AT+CMGL="ALL"**. The command **AT+CMGR=no** can be used to display a single message whose index number is **no**. The command **AT+CMGD=no** can be used to delete the message with index **no**. Command **AT+CMGD=1,1** can be used to delete all read messages on the SIM card.

14.3 Arduino SIM900 GSM/GPRS Shield

In this project we shall be using the popular SIM900 GSM/GPRS shield in an Arduino based project to receive an SMS message from an Android mobile phone in order to control a relay. It is worthwhile to look at the specifications of this shield. Figure 14.3 and Figure 14.4 show the front and back views of this shield. The SIM900 GPS/GPRS shield can be operated either with a PC or with a microcontroller such as the Arduino.

Figure 14.3 Front view of the SIM900 Arduino shield

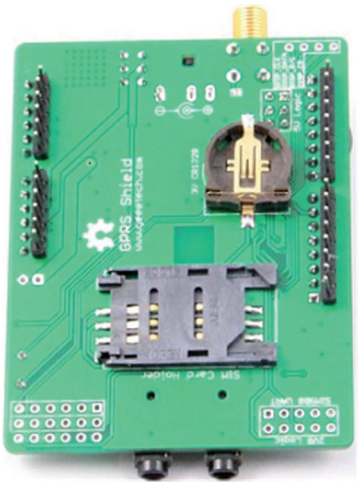


Figure 14.4 Back view of the SIM900 Arduino shield

At the front we have:

- SIM900 GSM/GPRS chip
- +5V Power connector
- Power select switch
- Microphone and speaker connectors
- Power key
- Antenna socket
- Network and status lights
- Serial port select jumpers (hardware or software)
- Edge connectors

At the back we have:

- Standard size SIM card holder
- Battery holder
- Pins to plug-in to the Arduino

The SIM900 GSM/GPRS shield is based on SIM900 module from SIMCOM and is compatible with Arduino and its clones. The shield provides the means to communicate using the GSM cell phone network. The shield allows us to achieve SMS, MMS, GPRS and Audio via UART by sending AT commands. The shield also has the 12 GPIOs, 2 PWMs and an ADC of the SIM900 module (They are all 2.8V logic compatible) present on-board.

The basic features of the SIM900 GSM/GPRS chip are:

- Quad-Band 850 / 900/ 1800 / 1900 MHz
- GPRS multi-slot class 10/8
- Class 4 (2 W @ 850 / 900 MHz)

- Class 1 (1 W @ 1800 / 1900MHz)
- Control via AT commands
- Short Message Service - so that you can send small amounts of data over the network (ASCII or raw hexadecimal).
- Embedded TCP/UDP stack - allows you to upload data to a web server.
- RTC supported.
- Low power consumption - 1.5mA in sleep mode

In addition, the shield offers the following features:

- Speaker and microphone jack sockets
- SIM card holder
- Power connector
- Serial communication select jumpers
- Antenna socket and antenna
- 12 General purpose I/Os (GPIOs)
- 2PWMs and an ADC
- Edge connectors for signal interface

Figure 14.5 shows the various components and jumpers on the front of the shield. When the board is to be operated without the Arduino it is necessary to power the board from an external +5V supply through the **power jack** connector. Make sure that the **power select** switch is set to use external power (switch position away from the board as shown in Figure 14.5). The board can be operated either using the software or the hardware serial communications ports. In this book we shall be using the software serial communication and this requires the two **serial port select** jumpers to be connected towards the board as shown in Figure 14.5. In this mode, pin D7 and pin D8 are the serial port receive (RX) and transmit (TX) pins.

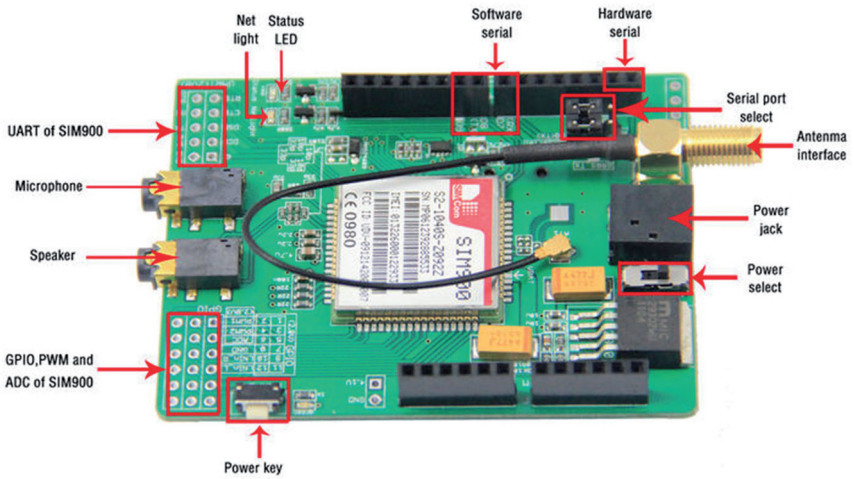


Figure 14.5 SIM900 GSM/GPRS shield in detail

It is important to note that you should have a valid SIM card installed in your SIM900 board before the board can be used in GSM/GPRS projects.

14.4 PROJECT 23 – Controlling a Relay by SMS Messages

14.4.1 Description

In this project a relay is connected to one of the digital input-output ports of an Arduino Uno computer. The relay is controlled remotely from an Android mobile phone by sending SMS text messages to activate and de-activate the relay.

14.4.2 Aim

The aim of this project is to show how the Android mobile phone can communicate with an Arduino Uno using SMS messages.

14.4.3 Block Diagram

Figure 14.6 shows the block diagram of the project. The SIM900 GSM/GPRS shield is plugged on top of the Arduino Uno. A relay is connected to one of the digital ports of the Arduino Uno.

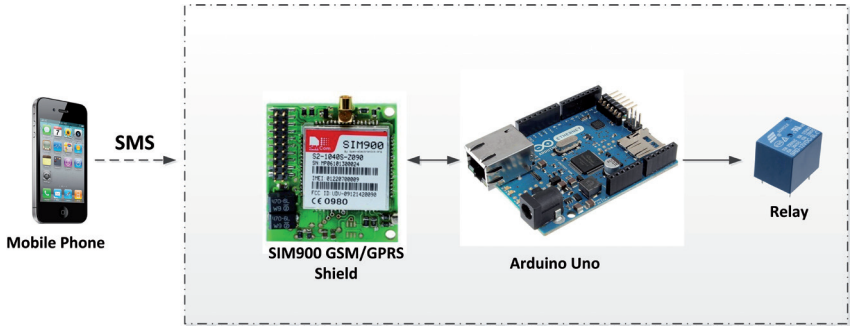


Figure 14.6 Block diagram of the project

14.4.4 Circuit Diagram

Figure 14.7 shows the circuit diagram of the project. In this project a relay module is used with built-in diode and transistor. The relay **S** pin is connected to digital port pin 2 of the Arduino Uno. Notice that power to the SIM900 module is supplied externally through a +5V, 2.5A power supply. This is necessary since the +5V output pin of the Arduino Uno cannot provide enough current to drive the SIM900 module.

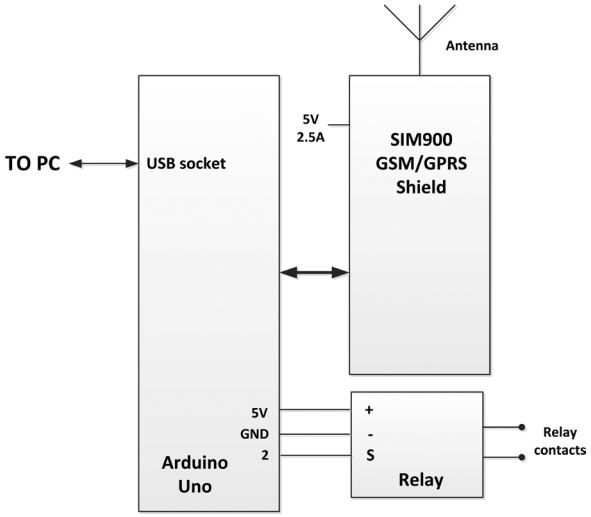


Figure 14.7 Circuit diagram of the project

14.4.5 Construction

As shown in Figure 14.8, the relay was mounted on a breadboard and jumper wires were used to make connections to the Arduino Uno.

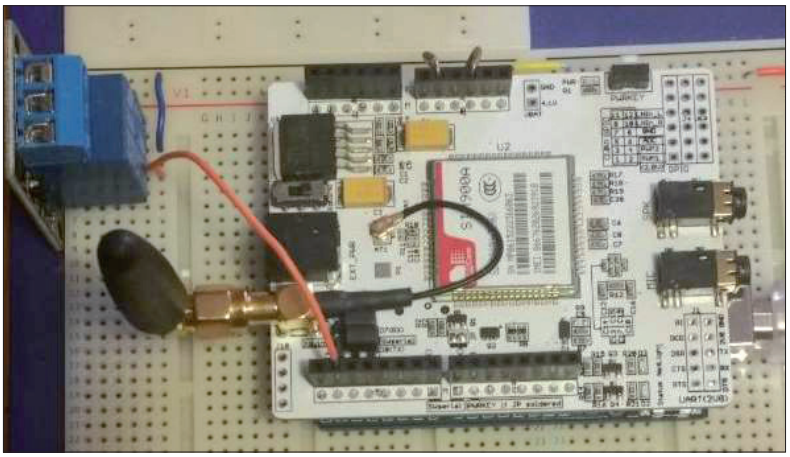


Figure 14.8 Construction of the project on a breadboard

14.4.6 Android Program

The Android program (program: SMSRELAY) is similar to the one given in Chapter 8, Project 9 where the acceleration was sent as an SMS message to a mobile phone. The complete program listing is shown in Figure 14.9. The program consists of a label, a Text Box, and a button. The label displays the text **COMMAND:**. The user command is entered into the Text Box named **Command**, and is sent to the Arduino Uno when the **SEND** button is clicked. Save the layout with the name **SMSRELAY**. Valid commands are **#ON** and **#OFF** to activate and de-activate the relay respectively. You should include the **Phone** and the

RuntimePermissions libraries in the program by selecting from the **Libraries Manager**. Inside the **Activity_Create**, load the layout and set the activity text to **SMS RELAY CONTROL**, and enable the runtime permissions for sending SMS:

```
Sub Activity_Create(FirstTime As Boolean)
    Activity.LoadLayout("SMSRELAY")
    Activity.Title = "SMS RELAY CONTROL"
    If rp.Check(rp.PERMISSION_SEND_SMS) = False Then
      rp.CheckAndRequest(rp.PERMISSION_SEND_SMS)
    End If
End Sub
```

Enter the following text inside the **SEND_Click** subroutine. Here, the command entered by the user is stored in string variable called **msg** and is then sent as an SMS to the specified mobile phone number, which was **07391846950** in this project:

```
Sub SEND_Click
    Dim msg As String
    msg = Command.Text
    MySms.Send2("00447415987053", msg, True, True)
End Sub

#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  Dim MySms As PhoneSms
  Dim rp As RuntimePermissions
End Sub

Sub Globals
  Private SEND As Button
  Private Command As EditText
  Private Label1 As Label
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("SMSRELAY")
  Activity.Title = "SMS RELAY CONTROL"
  If rp.Check(rp.PERMISSION_SEND_SMS) = False Then
    rp.CheckAndRequest(rp.PERMISSION_SEND_SMS)
  End If
End Sub
```

```
'
' Send the command to the Arduino Uno as an SMS text message
' The command is in the Command Text Box
'
Sub SEND_Click
 Dim msg As String
 msg = Command.Text
 MySms.Send2("00447391846950", msg, True, True)
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub
```

Figure 14.9 Android program

14.4.7 Arduino Uno Program

Figure 14.10 shows the Arduino Uno program (program: SMS). At the beginning of the program, library **SoftwareSerial** is included in the program and the connections with the GSM/GPRS modem are defined. Then, the AT commands used in the program are defined in various character arrays, and **Relay** is defined as 2 (it is connected to Arduino GPIO port 2). There are several functions in the program. Inside the **setup** function, Relay port pin is configured as OUTPUT and the relay is de-activated. The modem speed is then initialized to 19200 baud. Function **Send_To_Modem** sends the variable specified in its argument to the modem, including a carriage-return character. Function **Check_Response** checks the response received from the modem. If data is available from the modem, it is read and stored in character array **Buffer**. If the word **#ON** is received then variables **SMS_Arrived** and **Relay_State** are set to 1. If on the other hand the word **#OFF** is received then variables **SMS_Arrived** and **Relay_State** are cleared to 0. The main program is run in an endless loop. Inside this loop, all the existing received messages are initially deleted from the modem memory. Then the modem waits to receive an SMS message (i.e. **SMS_Arrived = 1**). Depending upon the content of the received message the relay is activated or de-activated.

```
/*****************************************************************
 *                CONTROLLING A RELAY BY SMS
 *                ==========================
 *
 *  In this project a relay is connected to digital I/O port
 *  pin 2 of the Arduino Uno. Additionally an SIM900 type
 *  GSM/GPRS shield is plugged on top of the Arduino Uno.
 *
 *  Notice that an external +5V poer supply must be connected
 *  to the shield. Also, make sure that the shield is compatible
```

```
 *  with your country as some shields are designed to work only
 *  in some parts of the world. If you receive Error 302 after
 *  sending the phone number, or, if the blue light on the
 *  shield is flashing every second at the same ON and OFF rates,
 *  then it is possible that your shield is not compatible with
 *  your country and you should update its firmware.
 *
 *  Make sure you press the ON/OFF switch for 2 seconds. If
 *  everything is working OK and the SIM card is registered
 *  to the network, then the lights on the shield should be:
 *
 *  Green (power):   Always ON
 *  Red (status):    Always ON
 *  Blue (netlight): Flashing, 64ms ON, 3 seconds OFF
 *
 *
 * File: SMS
 * Date: June, 2018
 * Author: Dogan Ibrahim
 ****************************************************************/

#include <SoftwareSerial.h>
SoftwareSerial GSM(7, 8);

//
// AT commands used
//
char No_Echo[] = "ATE0";
char Mode_Text[] = "AT+CMGF=1";
char Delete_All[] = "AT+CMGD=1,4";
char Get_Message[] = "AT+CNMI=1,2,0,0,0";
char SMS_Arrived;
char Relay_State;
//
int Relay = 2;                                  // Relay on pin 2
#define OFF LOW
#define ON HIGH

void setup()
{
  pinMode(Relay, OUTPUT);                       // Configure as output
  digitalWrite(Relay, OFF);                     // De-activate at start
  GSM.begin(19200);                             // Set GSM/GPRS speed
  delay(10000);
}
```

```
//
// This function sends a command to the GSM/GPRS modem, where the
// command is terminated with a carriage-return character
//
void Send_To_Modem(char *s)
{
    GSM.print(s);
    GSM.write(0x0D);
}

//
// This function reads the modem response
//
void Check_Response(void)
{
    char Buffer[255];
    int i = 0;

    while(!GSM.available())                  // Wait if no data is available
    {
      delay(1000);
    }

    while(GSM.available() > 0)               // If data is available
    {
       Buffer[i] = GSM.read();
       i++;
    }
    Buffer[i] = 0x0;                         // Terminate with NULL

    if(strstr(Buffer, "#ON") != 0)           // Command #ON detected
    {
       SMS_Arrived = 1;                      // Set the SMS_Arrived flag
       Relay_State = 1;                      // Set the Relay state to 1
    }

    if(strstr(Buffer, "#OFF") != 0)          // Command #OFF detected
    {
      SMS_Arrived = 1;                       // Set the SMS_Arrived flag
      Relay_State = 0;                       // Clear the Relay state
    }
}


//
// ********* This is the main program loop **********
```

```
//
void loop()
{
//
// Set NOECHO mode
//
    Send_To_Modem(No_Echo);
    delay(1000);
//
// Set modem into text mode
//
    Send_To_Modem(Mode_Text);
    delay(1000);

//
// Look for an SMS message from the mobile phone. The modem has been
// configured to work in Immediate Response mode, where the modem
// sends response as soon as an SMS message has been received
//
//
// Delete ALL messages to start with
//
   SMS_Arrived = 0;
   Send_To_Modem(Delete_All);
   delay(3000);
   Send_To_Modem(Get_Message);

    while(1)                            // Start of the endless program loop
    {
       Check_Response();                // Check if an SMS has been received
       if(SMS_Arrived == 1)
       {
          SMS_Arrived = 0;
          if(Relay_State == 1)
            digitalWrite(Relay, ON);           // Activate the relay
          else if(Relay_State == 0)
            digitalWrite(Relay, OFF);          // Deactivate the relay
//
// Delete ALL messages
//
          Send_To_Modem(Delete_All);
          delay(2000);
       }
    }
}
```

Figure 14.10 Arduino Uno program

Figure 14.11 shows an example command on the Android mobile phone to activate the relay.

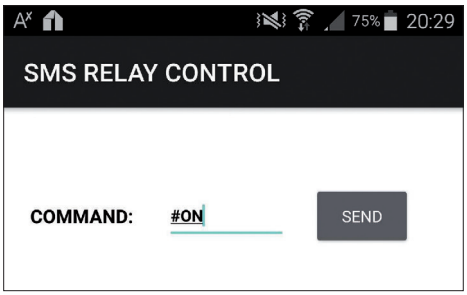


Figure 14.11 Command to activate the relay

Chapter 15 • Android to ESP32 WI-FI interface

15.1 Overview

In this Chapter we shall be looking at how an Android device (mobile phone or tablet) can communicate with an ESP32 processor over a Wi-Fi link using the UDP protocol. In this

Chapter we shall be programming the ESP32 processor using the Arduino IDE.
Before looking at the details of the Android to ESP32 interface, it is worthwhile to review some basic features of the ESP32 processor for those readers who may be new to the world of the ESP32.

15.2 The ESP32 Processor

The ESP32 processor has been developed by the Espressif, the developers of the ESP8266 processor. Although ESP32 has not been developed to replace the ESP8266, it improves on it in many aspects. The new ESP32 processor not only has Wi-Fi support, but it also has a Bluetooth communications module, making the processor to communicate with Bluetooth compatible devices. The ESP32 CPU is the 32-bit Xtensa LX6 which is very similar to the ESP8266 CPU, but in addition it has two cores, more data memory, more GPIOs, higher CPU speed, ADC converters with higher resolution, DAC converter, and CAN bus connectivity.
The basic specifications of the ESP32 processor are summarized below:

- 32-bit Xtensa RISC CPU: Tensilica Xtensa LX6 dual-core microcontroller
- Operation speed 160 to 240 MHz
- 520 KB SRAM memory
- 448 KB ROM
- 16 KB SRAM (in RTC)
- IEEE 802.11 b/g/ne/I Wi-Fi
- Bluetooth 4.2
- 18 channels x 12-bit ADC
- 2 channel x 8-bit DAC
- 10 touch sensors
- Temperature sensor
- 36 GPIOs
- 4 x SPI
- 2 x I2C
- 2 x I2S
- 3 x UART
- 1 x CAN bus 2.0
- SD memory card support
- 2.2V – 3.36V operation
- RTC timer and watchdog
- Hall sensor
- 16 channels PWM
- Ethernet interface
- Internal 8 MHz, and RC oscillator
- External 2 MHz – 60 MHz, and 32 kHz oscillator

- Cryptographic hardware acceleration (AES, HASH, RSA, ECC, RNG)
- IEEE 802.11 security features
- 5 μa sleep current

15.2.1 The Architecture of ESP32

Figure 15.1 shows the functional block diagram of the ESP32 processor (see **ESP32 Datasheet**, Espressif Systems, 2017). At the heart of the block is the dual-core Xtensa LX6 processor and memory. At the left hand side we ca see the peripheral interface blocks such as SPI, I2C, I2S, SDIO, UART, CAN, ETH, IR, PWM, temperature sensor, touch sensor, DAC, and ADC. The Bluetooth and Wi-Fi modules are situated at the top middle part of the block diagram. The clock generator and the RF transceiver are located at the top right hand of the block. Middle right hand is reserved for the cryptographic hardware accelerator modules such as the SHA, RSA, AES, and RNG. Finally, the bottom middle part is where the RTC, PMU, co-processor, and the recovery memory are.

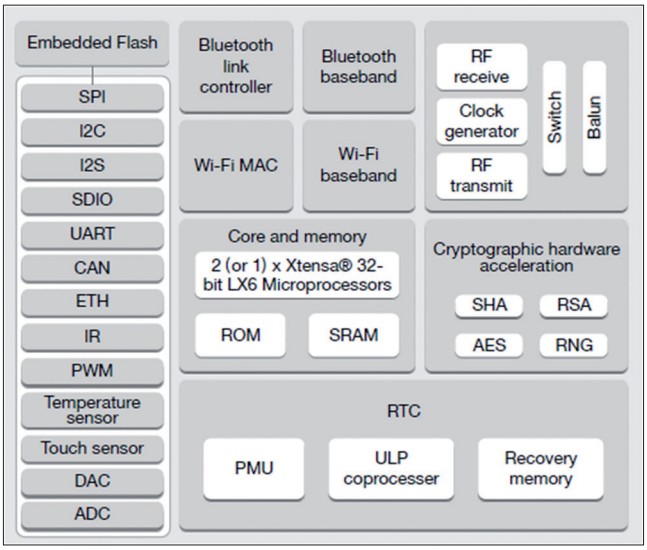


Figure 15.1 Functional block diagram of ESP32 processor

Figure 15.2 shows the system structure, consisting of two core Harvard architecture CPUs named PRO_CPU (for Protocol CPU) and APP_CPU (for Application CPU). The modules in the middle of the two CPUs are common to both CPUs. Detailed information about the internal architecture of the ESP32 can be obtained from the **ESP32 Technical Reference Manual** (Espressif Systems, 2017).

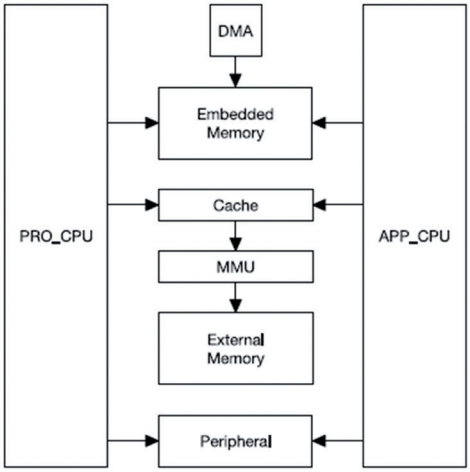


Figure 15.2 ESP32 system structure

15.2.2 ESP32 Development Boards

There are several development boards incorporating the ESP32 processor chip. Some popular ones are: SparkFun ESP32 Thing, Geekcreit ESP32, LoLin32, Pycom LoPy, ESP32 Test board, MakerHawk ESP32 board, Pesky Products ESP32 board, ESP32 DevKitC and others. In this Chapter we shall be using the highly popular ESP32 DevKitC development board. Before going into the details of our project, it is worthwhile to look at the specifications of the ESP32 DevKitC development board.

ESP32 DevKitC Development Board

ESP32 DevKitC is a small ESP32 processor based board developed and manufactured by Espressif. As shown in Figure 15.3, the board is breadboard compatible and has the dimensions 55 mm x 27.9 mm.

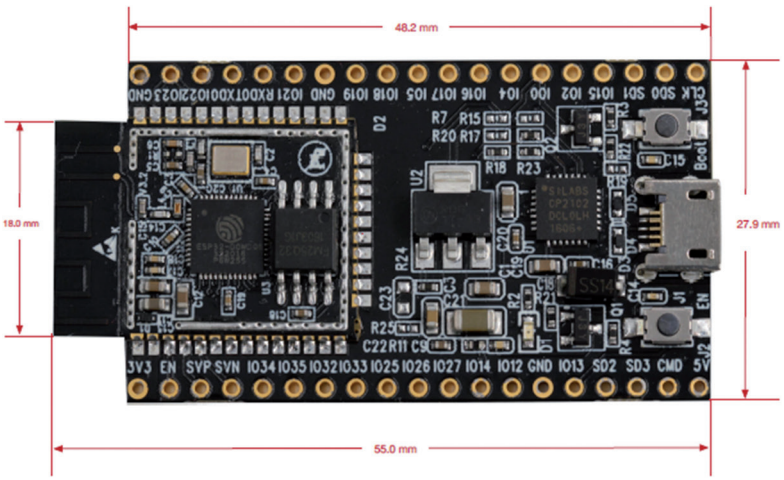


Figure 15.3 ESP32 DevKitC development board

The board has 19 connectors at each side for GPIO, clock, and power line interfaces. As shown in Figure 15.4, the two connectors carry the following signals:

Left Connector	**Right Connector**
+3.3V	GND
EN	IO23
SVP	IO22
SVN	TXD0
IO34	RXD0
IO35	IO21
IO32	GND
IO33	IO19
IO25	IO18
IO26	IO5
IO27	IO17
IO14	IO16
IO12	IO4
GND	IO0
IO13	IO2
SD2	IO15
SD3	SD1
CMD	SD0
+5V	CLK

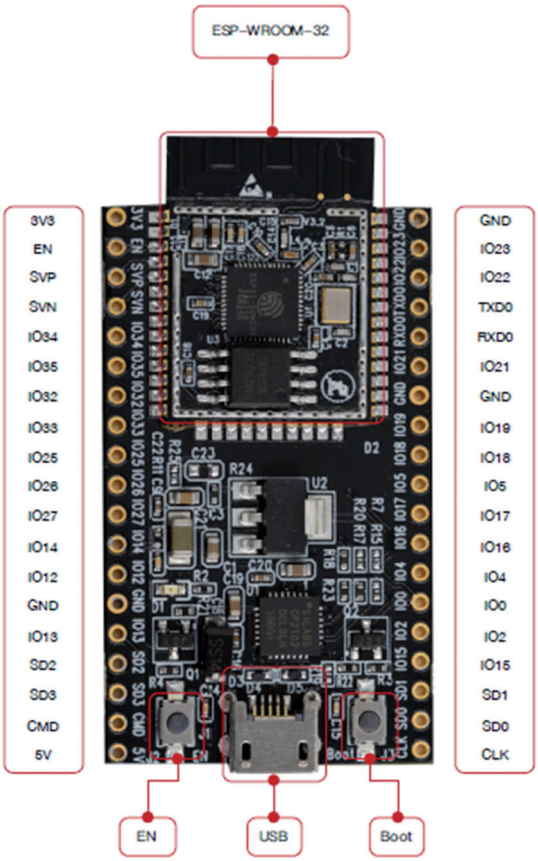


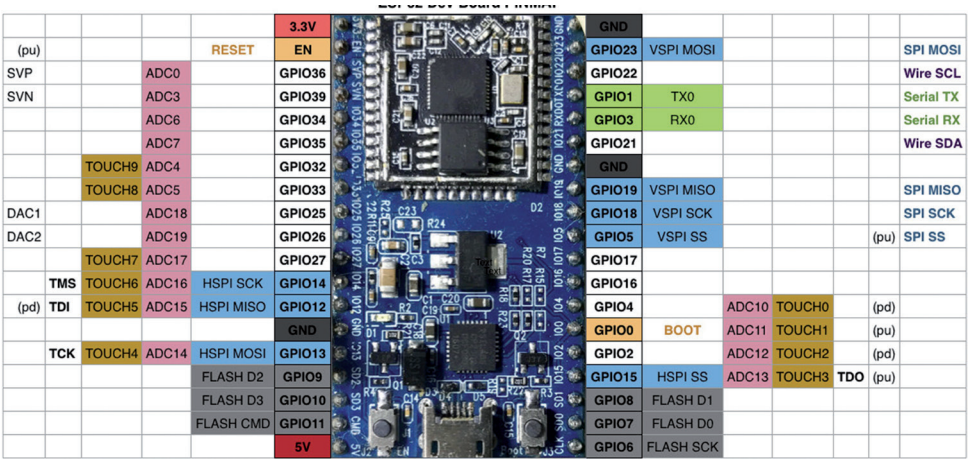
Figure 15.4 ESP32 DevKitC connectors

The board has a mini USB connector for connecting it to a PC. The board also receives its power from the USB port. Standard +5V from the USB port is converted into +3.3V on the board. In addition, two buttons are provided on the board named EN and BOOT, having the following functions:

EN: This is the reset button where pressing this button resets the board

BOOT: This is the download button. The board is normally in operation mode where the button is not pressed. Pressing and holding down this button and at the same time pressing the EN button starts the firmware download mode where firmware can be downloaded to the processor through the USB serial port.

The pins on the ESP32 DevKitC board have multiple functions. Figure 15.5 shows the functions of each pin. For example, pin 10 is shared with functions GPIO port 26. DAC channel 2, ADC channel 9, RTC channel 7, and RX01.



					3.3V		GND						
(pu)				RESET	EN		GPIO23	VSPI MOSI					SPI MOSI
SVP			ADC0		GPIO36		GPIO22						Wire SCL
SVN			ADC3		GPIO39		GPIO1	TX0					Serial TX
			ADC6		GPIO34		GPIO3	RX0					Serial RX
			ADC7		GPIO35		GPIO21						Wire SDA
		TOUCH9	ADC4		GPIO32		GND						
		TOUCH8	ADC5		GPIO33		GPIO19	VSPI MISO					SPI MISO
DAC1			ADC18		GPIO25		GPIO18	VSPI SCK					SPI SCK
DAC2			ADC19		GPIO26		GPIO5	VSPI SS				(pu)	SPI SS
		TOUCH7	ADC17		GPIO27		GPIO17						
	TMS	TOUCH6	ADC16	HSPI SCK	GPIO14		GPIO16						
(pd)	TDI	TOUCH5	ADC15	HSPI MISO	GPIO12		GPIO4		ADC10	TOUCH0		(pd)	
					GND		GPIO0	BOOT	ADC11	TOUCH1		(pu)	
	TCK	TOUCH4	ADC14	HSPI MOSI	GPIO13		GPIO2		ADC12	TOUCH2		(pd)	
				FLASH D2	GPIO9		GPIO15	HSPI SS	ADC13	TOUCH3	TDO	(pu)	
				FLASH D3	GPIO10		GPIO8	FLASH D1					
				FLASH CMD	GPIO11		GPIO7	FLASH D0					
					5V		GPIO6	FLASH SCK					

Figure 15.5 Multiple functions of each pin. Source: www.cnx-software.com

Note that GPIO34, GPIO35, GPIO36, GPIO37, GPIO38 and GPIO39 ports are input only and cannot be used as output ports.

The board operates with a typical power supply of +3.3V, although the absolute maximum is specified as +3.6V. It is recommended that the current capacity of each pin should not exceed 6 mA, although the absolute maximum current capacity is specified as 12 mA. It is therefore important to use current limiting resistors while driving external loads such as LEDs.

Depending upon the configuration the RF power consumption during reception is around 80 mA and it can be in excess of 200 mA during a transmission.

Powering Up The ESP32 DevKitC

Programming of the ESP32 DevKitC requires the board to be connected to a PC through its mini USB port. The communication between the board and the PC takes place using the standard serial communication protocol.

ESP32 DevKitC is preloaded with firmware that can be used to test the board. This firmware is activated by default when power is applied to the board. Before communicating with the board we have to run a terminal emulation software on our PC. There are several terminal emulation software available free of charge. Some examples are HyperTerm, Putty, X-CTU and so on.

15.3 PROJECT 24 – Controlling an LED by the ESP32 DevKitC

15.3.1 Description

In this project an LED is connected to the ESP32 DevKitC and is controlled remotely from an Android mobile phone over a Wi-Fi link using the UDP protocol. As in Project 18 in Chapter 11, Android commands **ON** and **OFF** turn the LED ON and OFF respectively. Command **#** terminates the program on the ESP32 DevKitC.

15.3.2 Aim

The aim of this project is to show how the Android mobile phone can communicate with an ESP32 DevkitC computer over a Wi-Fi link using the UDP protocol.

15.3.3 Block Diagram

Figure 15.6 shows the block diagram of the project. The communication between the ESP32 DevKitC and the Android mobile phone is via the local Wi-Fi router, using the UDP protocol. The ESP32 DevKitC has built-in Wi-Fi and Bluetooth capabilities. In this example project only the Wi-Fi is used.

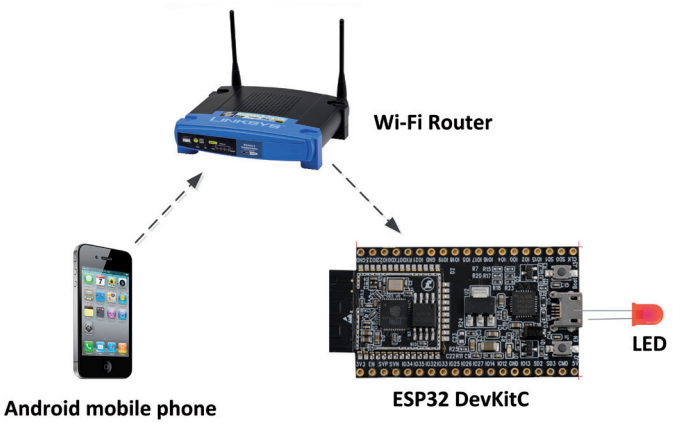


Figure 15.6 Block diagram of the project

15.3.4 Circuit Diagram

Figure 15.7 shows the circuit diagram of the project. The LED is connected to port pin GPIO23 through a 330 Ohm current limiting resistor.

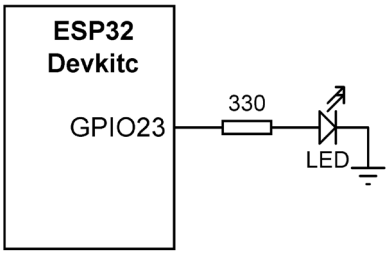


Figure 15.7 Circuit diagram of the project

15.3.5 Construction

As shown in Figure 15.8, the ESP32 DevKitC was mounted on a breadboard together with the LED and the current limiting resistor.

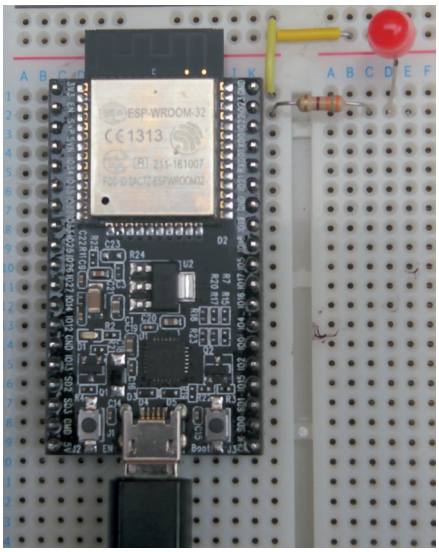


Figure 15.8 Construction of the project on a breadboard

15.3.6 Android Program

The Android program (program: UDP6) is basically same as the program given in Figure 11.16 (program: UDP2) where only the activity title name and the destination (ESP32 DevKitC) IP address are changed. The new activity title is set to **ESP32 LED CONTROL**, and the destination IP address is now **192.168.1.156**. Commands **ON** and **OFF** entered into an Edit Text box on the Android mobile phone to turn the LED ON and OFF respectively. Figure 15.9 shows the Android program listing. Figure 15.10 shows the ON command entered on the mobile phone to turn the LED ON.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  Dim UdpSocket As UDPSocket
  Dim UdpPacket As UDPPacket
End Sub

Sub Globals
  Private Label1 As Label
  Private Sendto As Button
  Private Send As EditText
End Sub

Sub Activity_Create(FirstTime As Boolean)
  Activity.LoadLayout("UDP6")
```

```
  Activity.Title = "ESP32 LED CONTROL"
  If FirstTime Then
    UdpSocket.Initialize("UDP", 2000, 1024)
  End If
End Sub

Sub Sendto_Click
  Dim MyData() As Byte
  MyData = Send.Text.GetBytes("UTF8")
  UdpPacket.Initialize(MyData, "192.168.1.156", 2000)
  UdpSocket.Send(UdpPacket)
End Sub

Sub UDP_PacketArrived(Packets As UDPPacket)
End Sub
```

Figure 15.9 Android program

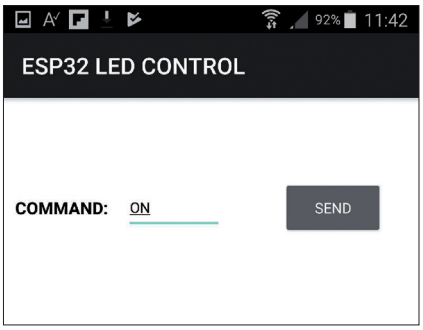


Figure 15.10 Command to turn ON the LED

15.3.7 ESP32 Program

The ESP32 program has been developed using the Arduino IDE. The program (ESP32LED) listing is shown in Figure 15.11. At the beginning of the program the WiFi libraries are included in the program, LED is defined as 23, the UDP port number is set to 2000, and an array called **Packet** has been created.

Then the name and password of the local Wi-Fi router are defined as constants. Subroutine **Connect_WiFi** is called from the setup routine to make a connection to the local Wi-Fi network. Additionally, the LED port is configured as output and the LED is turned OFF at the beginning of the program.

The remainder of the program is run in an endless loop. Inside this loop the program waits to receive UDP packets. After a packet is read, **if** statements are used to decode the contents of the packet. If the packet contains the word **ON** then the LED is turned ON. If on the other hand the packet contains the word **OFF** then the LED is turned OFF. This process is repeated forever.

It is assumed here that the reader has already configured the Arduino IDE to enable ESP32 programs to be developed and then uploaded to the ESP32 processor. Connect your ESP32 DevKitC development board to your PC through a mini USB cable. You should see the red power light turned ON on the board. Start the Arduino IDE and click **Tools -> Board** and select the **ESP32 Dev Module** as the module type. Click **Tools -> Port** and select the port where the ESP32 DevKitC is connected to. Click **Sketch -> Verify/Compile** to compile your code and if there are no errors click **Sketch -> Upload** to upload the compiled code to the ESP32 processor. Make sure that you hold the **Boot** button (**the one at the right hand side**) during the entire upload process. Release the **Boot** button and press the **Reset** button (**the one at the left hand side**). Start the Android program (Figure15.9) on your mobile phone and enter ON or OFF commands to control the LED connected to the ESP 32 DevKitC board.

```
/*********************************************************************
 *                ESP32 REMOTE LED CONTROL
 *                ========================
 *
 * In this project an LED is connected to port pin GPIO23 of the
 * ESP32 DevKitC development board. This LED is controlled by
 * commands sendt from an Android mobile phone. Valid commands are
 * ON and OFF to turn the LED ON and OFF respectively.
 *
 *
 * File: ESP32LED
 * Date: June, 2018
 * Author: Dogan Ibrahim
 *********************************************************************/
#include "WiFi.h"
#include <WiFiUdp.h>

#define LED 23                          // LED at port GPIO23
WiFiUDP udp;
const int Port = 2000;                  // UDP port number
char Packet[80];

//
// Local Wi-Fi name and password
//
const char* ssid = "BTHomeSpot-XNH";
const char* password = "49345abaab";

//
// This function connects the ESP32 to the local Wi-Fi router
//
void Connect_WiFi()
{
```

```
  WiFi.begin(ssid, password);
  while(WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
  }
}


//
// The Setup routine
//
void setup()
{
  pinMode(LED, OUTPUT);                  // Configure as output
  digitalWrite(LED, LOW);                // LED OFF at beginning
  Connect_WiFi();                        // Connect to Wi-Fi
  udp.begin(Port);
}


//
// Main program loop. Read the command from the received packet
// and then control the LED accordingly. Valid commands are ON
// and OFF
//
void loop()
{
  int PacketSize = udp.parsePacket();
  if(PacketSize)
  {
    udp.read(Packet, PacketSize);
    if(Packet[0] == 'O' && Packet[1] == 'N')
      digitalWrite(LED, HIGH);
    else if (Packet[0] == 'O' && Packet[1] == 'F' && Packet[2] == 'F')
      digitalWrite(LED, LOW);
  }
}
```

Figure 15.11 ESP32 DevKitC program

15.4 PROJECT 25 – Millivoltmeter

15.4.1 Description

In this project the voltage is measured (in millivolts) every second by one of the ADC inputs of the ESP32 DevKitC board and is sent to the Android mobile phone where it is displayed.

15.4.2 Aim

The aim of this project is to show how the ESP32 DevKitC can send UDP based data packets to the Android mobile phone over a Wi-Fi link using the UDP protocol.

15.4.3 Block Diagram

Figure 15.12 shows the block diagram of the project. The communication between the ESP32 DevKitC and the Android mobile phone is via the local Wi-Fi router, using the UDP protocol. A potentiometer is connected to one of the ADC inputs of the ESP32 DevKitC and the voltage is measured and sent to the Android mobile phone.

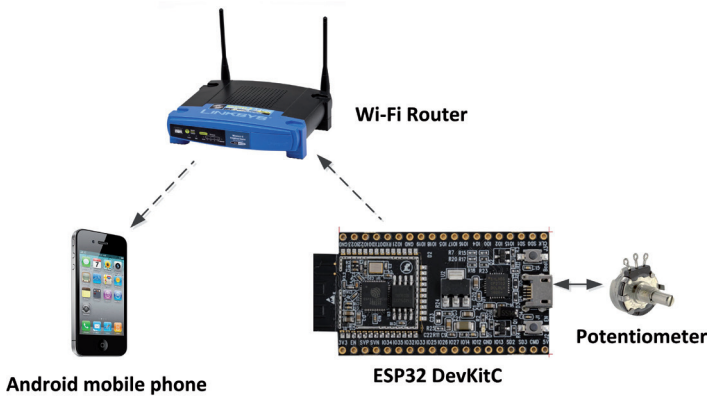


Figure 15.12 Block diagram of the project

15.4.4 Circuit Diagram

Figure 15.13 shows the project circuit diagram. The arm of a 10K potentiometer is connected to pin GPIO36 (one of the ADC channels) of the ESP32 DevKitC development board.

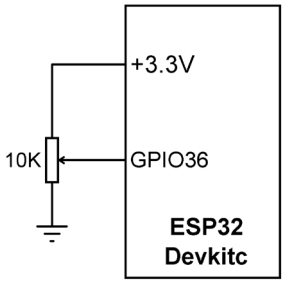


Figure 15.13 Circuit diagram of the project

15.4.5 Android Program

The Android program is very similar to the program in Project 19 (program: UDP3), except that here the activity title is changed. Figure 15.14 shows the Android program (UDP7) listing.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
 Dim UdpSocket As UDPSocket
 Dim UdpPacket As UDPPacket
End Sub

Sub Globals
 Private Label1 As Label
 Private Temperature As EditText
End Sub

Sub Activity_Create(FirstTime As Boolean)
 Activity.LoadLayout("UDP7")
 Activity.Title = "MILLIVOLTMETER"
 If FirstTime Then
   UdpSocket.Initialize("UDP", 2000, 1024)
 End If
End Sub

Sub UDP_PacketArrived(Packets As UDPPacket)
 Dim msg As String
 msg = BytesToString(Packets.Data, Packets.Offset, Packets.Length, "UTF8")
 Temperature.Text = msg
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
End Sub
```

Figure 15.14 Android program

15.4.6 ESP32 Program

The ESP32 program (program: VOLTS) listing is shown in Figure 15.15. At the beginning of the program the Wi-Fi libraries are added to the program and also **POT** is defined as 36 (i.e. GPIO36). Then the local Wi-Fi name and password are defined as constants. Inside the setup routine the program connects to the Wi-Fi. The remainder of the program is run in an endless loop. Inside this loop the ADC voltage is read from pin GPIO36 and is converted into millivolts. Notice that the ADC resolution is 12 bits (i.e. 4096 quantization levels) and the reference voltage is 3.3V. The voltage in millivolts is then converted into a string and stored in variable **millivolts** and is sent to the Android mobile phone (the IP address of the mobile

phone is "192.168.1.78"). The above process repeats every second where by rotating the potentiometer arm the voltage displayed on the Android mobile phone changes between 0V and 3300mV. Figure 15.16 shows a typical display on the mobile phone.

```
/*************************************************************
 *                 VOLTAGE DISPLAY
 *                 ===============
 *
 * This is a voltmeter project. The voltage is measured in
 * millivolts and is sent to the Android mobile phone where
 * it is displayed. A potentiometer is connected to pin GPIO36
 * to simulate external voltages. As the potentiometer arm is
 * varied the voltage is read and sent to the Android mobile
 * phone every second.
 *
 *
 * File: VOLTS
 * Date: June, 2018
 * Author: Dogan Ibrahim
 *************************************************************/
#include "WiFi.h"
#include <WiFiUdp.h>

#define POT 36                        // Potentiometer on GPIO36
WiFiUDP udp;
const int Port = 2000;                // UDP port number

//
// Local Wi-Fi name and password
//
const char* ssid = "BTHomeSpot-XNH";
const char* password = "49345abaab";

//
// This function connects the ESP32 to the local Wi-Fi router
//
void Connect_WiFi()
{
  WiFi.begin(ssid, password);
  while(WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
  }
}
```

```
//
// The Setup routine
//
void setup()
{
  Connect_WiFi();                      // Connect to Wi-Fi
  udp.begin(Port);
}


//
// Main program loop. Read the voltage at the potentiometer arm
// and then send it to the Android mobile phone every second
//
void loop()
{
  int Avolts = analogRead(POT);
  float mV = Avolts * 3300.0 / 4096.0;
  int MV = (int)mV;
  String millivolts;

  millivolts = String(MV);
  udp.beginPacket("192.168.1.78", Port);
  udp.print(millivolts);
  udp.endPacket();
  delay(1000);
}
```

Figure 15.15 ESP32 program

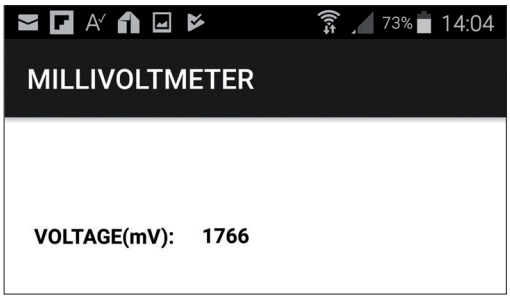


Figure 15.16 Display on the mobile phone

Appendix A • Using the Android emulator

The Android emulator is a Virtual Device Manager (AVD) that emulates an Android mobile phone. The AVD is provided by Google as part of their SDK to help users during the development of Android based applications. As we shall see later in this Appendix, the AVD is run from the B4A IDE by clicking **Tools -> Run AVD Manager**.

The emulator can be useful during program development where the programmer can emulate the operation of the real mobile phone on a PC screen. The emulator runs on a PC without the need of a real mobile phone. Although the emulator could be a useful tool, the author prefers to use a real mobile phone and install and test applications on a real device during program development. Perhaps the biggest advantage of using the emulator is that the developed program can be tested on different models of Android mobile phones with different sizes.

In this Appendix we will create a few very simple applications and then see how we can run them on the emulator.

Example A1
In this example the message **Hello from the emulator** will be displayed on the emulator screen.

The steps to start and use the emulator are described below:

- Start the B4A IDE
- Click **Tools -> Run AVD Manager**
- Create a new device and an API level
- You should see the Android emulator screen as in Figure A.1 with keys at the right hand side and the screen at the left hand side

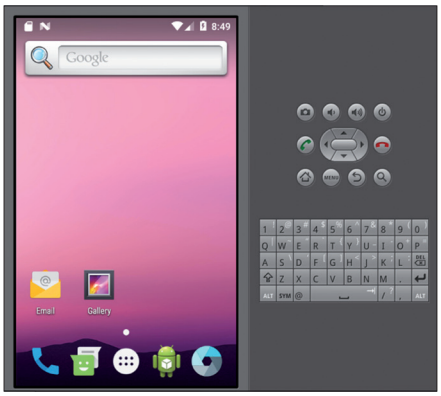


Figure A.1 The emulator screen

- Enter the following statement inside the **Activity_Create**:

```
Sub Activity_Create(FirstTime As Boolean)
    Msgbox("Hello from the emulator","Message")
End Sub
```

- Click **Run** (F5) to run the program and give a name to the program (e.g. hello)

- You should now see the message **Hello from the emulator** displayed in a msg-box on your emulator as shown in Figure A.2

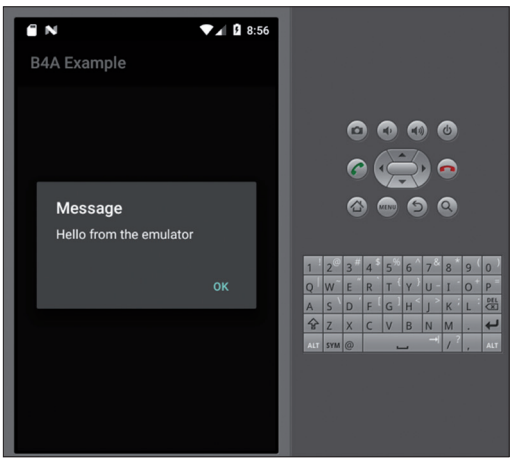


Figure A.2 Displaying the message in a msgbox on the emulator

Example A2
In this example we will place an Edit Text box and a button on the emulator screen. The example will display a random number between 1 and 100 when the button is pressed. The steps are given below:

- Start the B4A IDE

- Click **Tools -> Run AVD Manager**

- Open the Designer screen by clicking Designer -> Open Designer

- Click **WYSIWYG Designer -> Connect** in the designer screen to connect to the emulator

- Set the **Color** of **Activity Properties** to white

- Add a button and an Edit Text box. Give the names **Button1** and **Number** to the button and the Edit Text box respectively. Set the **Text Color** of the Edit Text box to black. Set the button text to **Generate** and its **Text Color** to white.

- You should see the button and the Edit Text box displayed on the emulator screen as in Figure A.3

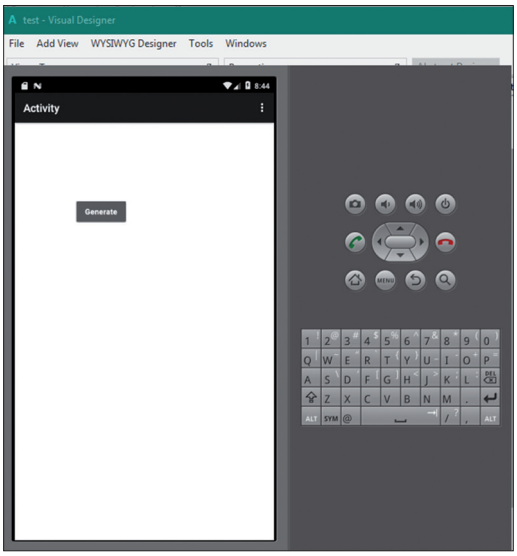


Figure A.3 Button and Edit Text box on the emulator screen

- Click **Tools -> Generate Members**, **Select All Views**, and enable **Click** for Button1 as shown in Figure A.4 so that the button responds to clicks. Click **Generate Members** and close the screen.

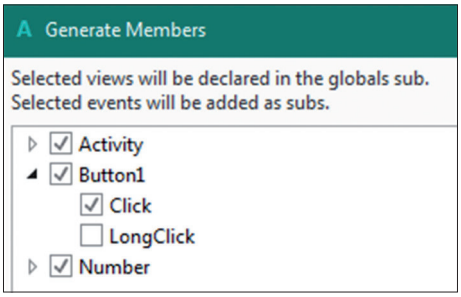


Figure A.4 Generate members

- Enter the following code inside subroutine **Button1_Click**. This code will generate a random number between 1 and 100 and display in the Edit Text box:

```
Sub Button1_Click
    Dim RandomNumber As Int
    RandomNumber = Rnd(1, 101)
    Number.Text = RandomNumber
End Sub
```

- Give a name to the layout (e.g. NUMBER) and Click **Run** (F5) to run the program
- Click the button and you should see a number displayed on the screen as shown in Figure A.5

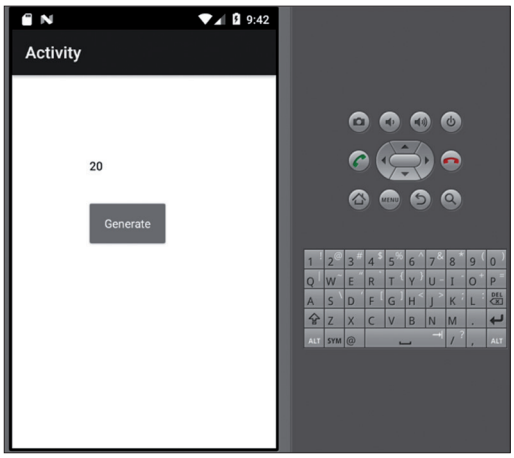


Figure A.5 Click the button to display a random number

The complete program listing is shown in Figure A.6.

```
#Region  Project Attributes

#Region  Activity Attributes


Sub Process_Globals
  'These global variables will be declared once when the application starts.
  'These variables can be accessed from all modules.
End Sub

Sub Globals
  Private Button1 As Button
  Private Number As EditText
End Sub

Sub Activity_Create(FirstTime As Boolean)
        Activity.LoadLayout("NUMBER")
End Sub

Sub Activity_Resume
End Sub

Sub Activity_Pause (UserClosed As Boolean)
```

```
End Sub


Sub Button1_Click
  Dim RandomNumber As Int
  RandomNumber = Rnd(1, 101)
  Number.Text = RandomNumber
End Sub
```

Figure A.6 Program listing

Appendix B • Publishing apps on Google Play

We may want to publish and perhaps sell our apps on the Google Play. In this Appendix we shall create a very simple application and see how it can be published on the Google Play website.

The example application developed in this Appendix is a simple dice. We will have a button and an Edit Text box (as in Appendix A, Example A2). Clicking the button will generate a random dice number between 1 and 6.

B.1 Developing the Application for Google Play

Every B4A application must have a unique Package Name. The default name is B4A. A Package Name can be set by clicking **Project -> Build Configurations** as shown in Figure B.1. Leave the **Configuration** field as **Default** so that the DEBUG or RELEASE mode are selected automatically depending on the current compiler setting. Give a name to field **Configuration Name**. The Package Name is important and should be a unique name in your projects. In this example we will set the Package Name to **dice.test.example**. The Package name must consists of at least two words, separated with a dot. Leave the **Conditional Symbols** filed as blank.

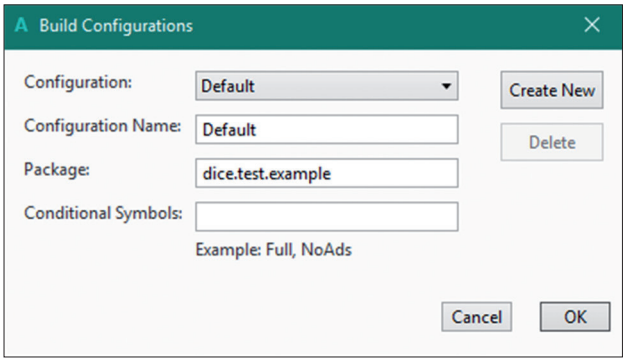


Figure B.1 Build configuration

You should then configure the **Project Attributes** and the **Activity Attributes** as desired. An application must also have an icon associated with it. There are several types of icons, such as tab, status bar, menu, launcher, and so on. In this example we shall use an icon compatible with the Google Play. The Google Play icon should be PNG file and have 512x512 pixels. The icon size should not be greater than 1MB. Application programs are available on the Internet free of charge for creating icons. There are also programs that can convert images to icons. In this example a dice icon is used with 512x512 pixels. Click **Project -> Choose Icon** and select the icon you wish to use in your application.

Now, create your dice application. Add a button and an Edit Text box such that when the button is pressed a dice number will be displayed between 1 and 6 in the Edit Text box. Name your application program and the designer layout as **exampledice**.

Now we should generate our APK file which contains the compiled project code and all the other necessary files for the project to run. Android requires that the apps must be signed before they can be installed. APKs can be signed using electronic certificates which contain a key and the identity of the owner. Two keys are used: a **private key** and a **public key**. The developed apps is signed using the private key, and is distributed with the certificate containing the public key. Click **Tools -> Private Sign Key** to create a private key (see Figure B.2). The country code in this figure should be selected from the following web site:

https://www.digicert.com/ssl-certificate-country-codes.htm

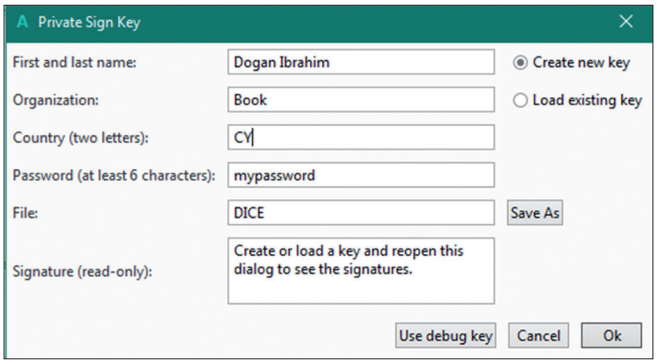


Figure B.2 Generating private key

Keys are stored in a **keystore** file. It is important that you remember the password you entered. If the file is lost then you will not be able to update your apps in Google Play website. Notice that B4A uses the same key for all of your B4A projects.

Now, you should compile your application in RELEASE mode. The APK file will be saved in the Objects folder of your application folder. Before publishing your apps on the Google Play website you have to one-time register with Google as a Developer. After registering you can publish as many apps as you like. To register as a Google Developer, login to Google, and go to Google Play Developer sign-in page. You can create a new Google developer account by entering the following site:

https://play.google.com/apps/publish/signup/

You should accept the Developer Agreement and continue to pay $25 as the registration fee. Click **PUBLISH AN ANDROID APP ON GOOGLE PLAY** as shown in Figure B.3.

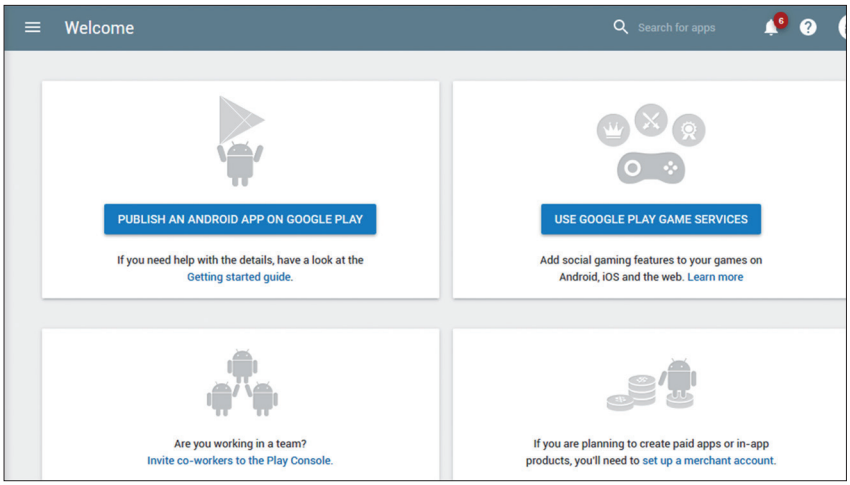


Figure B.3 Click to publish your apps

Click **Create Application** and enter the details as in Figure B.4. Here, the application is called **MyDice**. You should see the **Store listing** form in Figure B.5. Follow the instructions and enter short description, full description, screenshot of your apps, icon etc. At least two screenshots must be uploaded. Short description, full description, screenshot, icon, and feature graphic are shown in the final apps in Google Play website. Notice that the icon must be 32-bit 512x512 pixels PNG file, screenshot must have min length for any side 320 pixels and max length for any side 3840 pixels JPEG or 24-bitPNG file and the feature graphic must be 1024 width x 500 height JPG or 24-bit PNG file.

Figure B.6 shows the icon and the feature graphic used in this example (you can drag and drop at this stage). Select the application type, and category. You should select the **Content Rating** and answer all the questions to apply a content rating to your apps. Figure B.7 shows the **Store listing** form with the Content rating added. Enter the contact details and click the **Privacy Policy**. Click **SAVE DRAFT** after completing filling the form and you should see a green tick next to the section name.

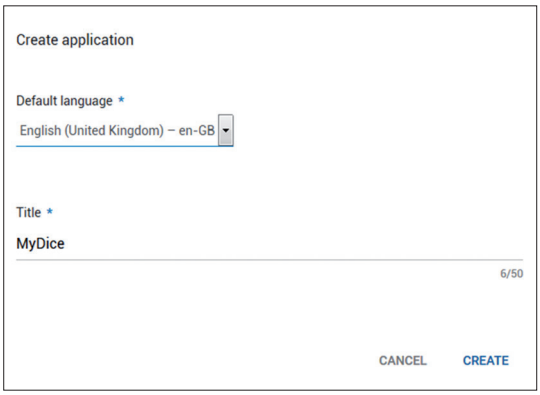


Figure B.4 Create an application

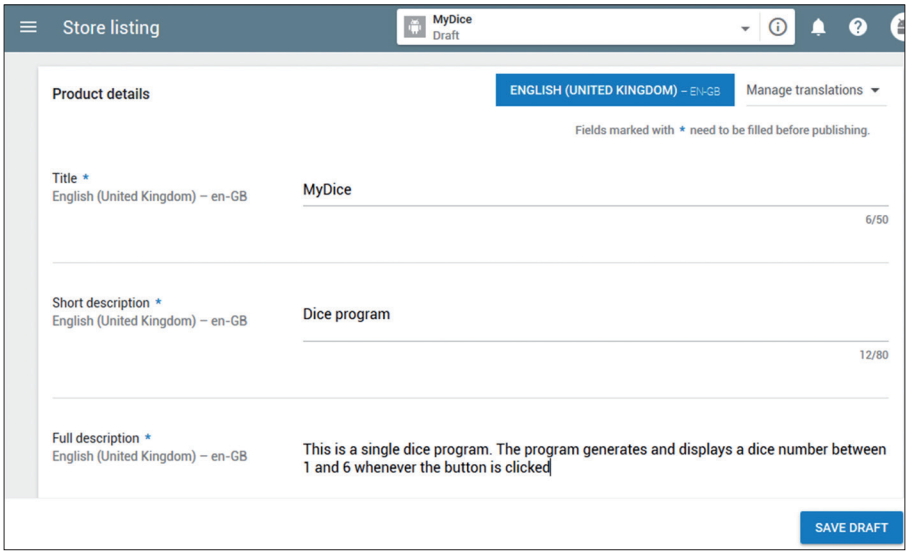


Figure B.5 Enter the required details

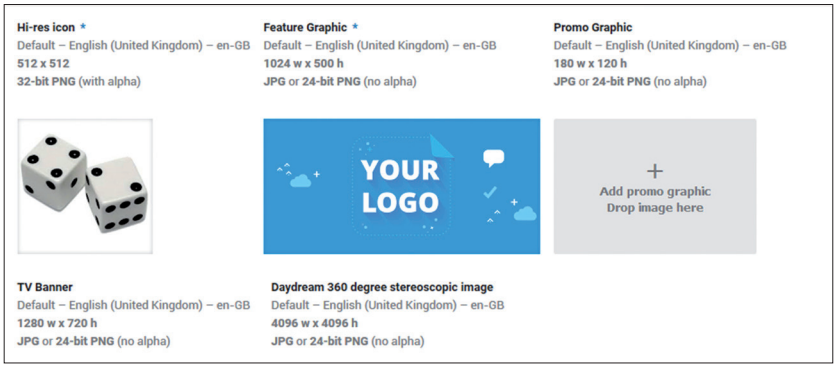


Figure B.6 The icon saved for the apps

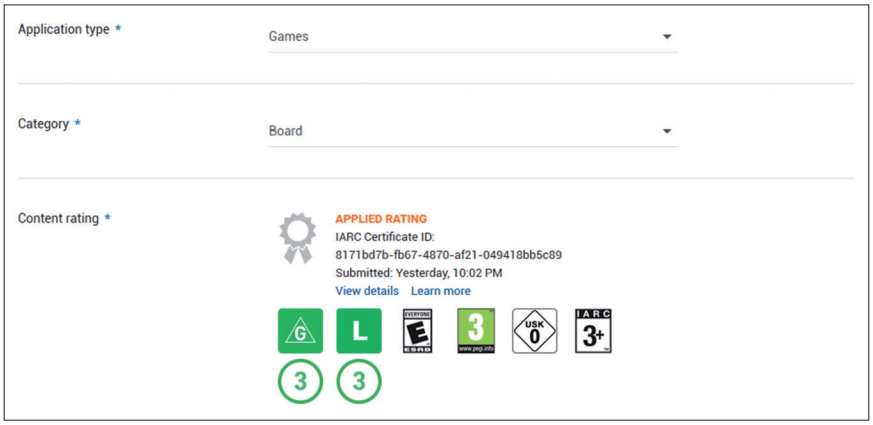


Figure B.7 Content rating added to the apps

Before you can roll out your release, make sure that you've completed your app's **Store listing**, **Content rating**, and **Pricing & distribution** sections. When each section is complete, you'll see a green tick mark next to it on the left menu. **Notice that you cannot release your apps before completing all these 3 sections. Check the green tick marks next to each section at the left hand side of the screen and *complete the sections until you see a green mark next to each section name*.**

In the **Pricing & distribution** section, specify whether your application will be free of charge, manage the countries allowed to install your application, and answer the other questions appropriately. After submitting the form you should see a green tick mark at the left of the section name.

Click **App releases** at the left hand pane of the screen. Click **MANAGE** in **production**. Click **CREATE RELEASE**. Drag and drop your APK file to **Android App Bundles and APKs to add**, or click **BROWSE FILES** and locate your APK file (see Figure B.8). Enter a **Release name** (**exampledice V1.0** in this example) and release notes for your apps. Click **SAVE** at the bottom right corner of the display. Click **REVIEW** to review details of your publication.

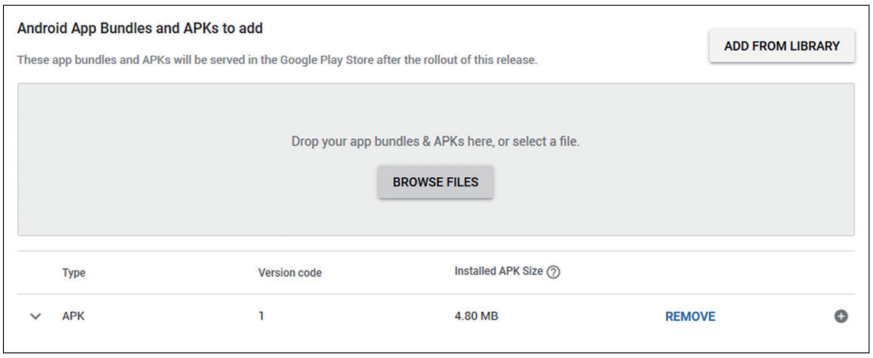


Figure B.8 Locate and upload your APK file

If you have uploaded your APK successfully and you see 3 green marks at the left of the sections named **Store listing**, **Content rating**, and **Pricing & distribution** then you should be ready to roll out and release your apps. **Confirm roll-out** and your apps will be published. You should see a release confirmation screen as in Figure B.9.

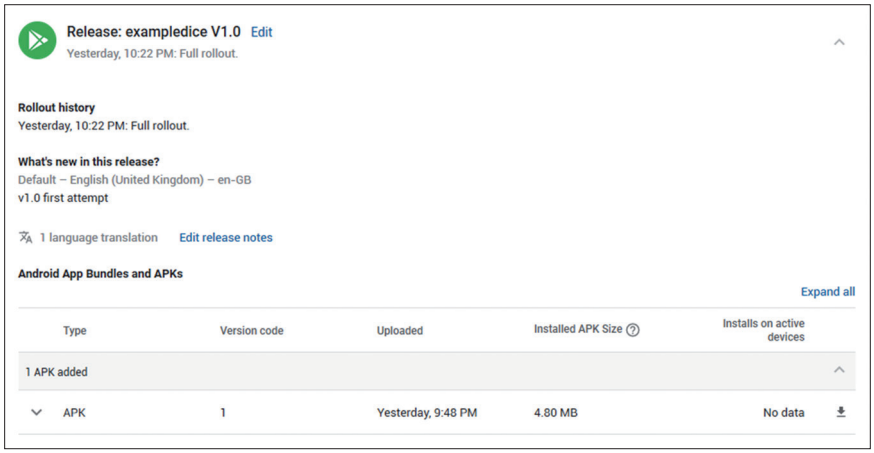


Figure B.9 Release confirmation screen

The apps should be available on Google Play after a short while. Figure B.10 shows our apps listed in Google Play (named: **MyDice**). Clicking the apps shows its details as in Figure B.11 and Figure B.12. Notice how the short description, full description, screenshots, icon, the feature graphic and the developer name are all displayed in the apps.

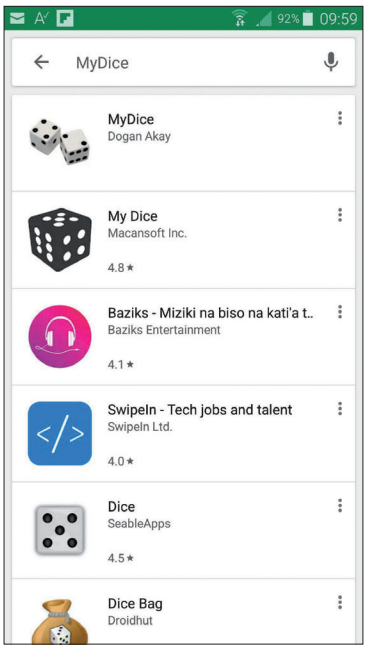


Figure B.10 Apps listed on Google Play

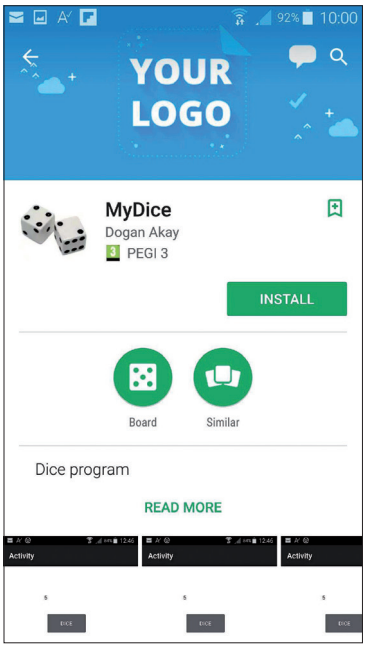


Figure B.11 Details of the apps

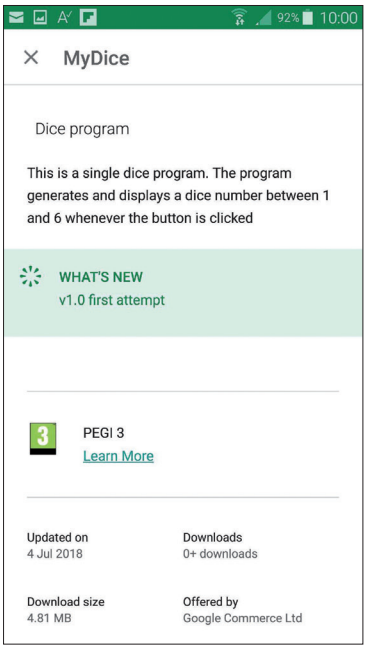


Figure B.12 More details of the apps

Index

WITH BASIC FOR ANDROID – B4A

ANDROID APP DEVELOPMENT
FOR ELECTRONICS DESIGNERS

Dogan Ibrahim

Prof Dr Dogan Ibrahim has a BSc degree in electronic engineering, an MSc degree in automatic control engineering, and a PhD degree in digital signal processing. He has worked in many industrial organizations before he returned to academic life. Prof Ibrahim is the author of over 60 technical books and over 200 technical articles on microcontrollers, microprocessors, and related fields. He is a Chartered electrical engineer and a Fellow of the Institution of Engineering Technology.

Pro makers, students, and hobbyists can develop apps for Android mobile devices using the Basic For Android (B4A) programming language and Integrated Development Environment (B4A IDE). Dr. Dogan Ibrahim begins Android App Development for Electronics Designers with a description of how to install the B4A on a PC. He then presents simple projects to introduce B4A's syntax and programming features.

Electronics designers will enjoy this book because it describes how an Android mobile device can communicate with a variety of hardware platforms — including Raspberry Pi, Arduino, and the ESP32 processor — over a Wi-Fi link or by using SMS text messages. Some of the projects show how data packets can be sent from a Raspberry Pi, Arduino, or ESP32 processor to an Android mobile phone and then displayed on the mobile phone.

All the Android projects in this book were developed using the B4A programming language. The Raspberry Pi projects were developed using Python. Arduino and the ESP32 processor projects make use of the popular Arduino IDE. Full program listings for all the projects as well as the detailed program descriptions are given in the book. Users should be able to use the projects as they are presented or modify them to suit to their own needs.

ISBN 978-1-907920-71-4

781907 920714

Elektor International Media BV

www.elektor.com