THE BLUEGIGA BLE 112 MODULE

DEBUGGING ( UPLOADING A FIRMWARE INTO THE 128 KB FLASH MEMORY OF BLE112)

To get started, go to the [Create Bluegiga Tech Forum Account page](http://www.bluegiga.com/create_account) and create an account if you haven’t already done so, then go to the [support page](http://www.bluegiga.com/support) and login.  Once you’ve logged in, head to the [BLE112 section](http://techforum.bluegiga.com/ble112) then head to the “Software Releases” section and download the Bluegiga-ble-1.2.1-91**Software Development Kit** software package.  This contains the windows drivers for the BLED112 USB dongle, sample BGScript projects as well as the necessary tools required for compiling the BGScript projects into hex files to be flashed onto the BLE112.

Next, download the following documentation:

* **Profile Toolkit developer guide**
* **BGScript developer guide**
* **BLEGUI User Guide**

The Profile Toolkit developer guide provides details regarding the structure of a typical BGScript project, including instructions on how to use the binary utilities from the Bluetooth Smart 1.1 (Beta) Software Development Kit to compile the BGScript into a hex file.  Make sure you read this guide before anything else

The BGScript developer guide contains an explanation of the BGScript syntax as well as useful BGScript examples.  Read this guide after checking out the Profile Toolkit guide.

The BLEGUI User Guide contains information to use the the BLEGUI tool included with Bluegiga-ble-1.2.1-91Software Development Kit.  This is an extremely useful tool which allows you to attach to the BLED112 USB Dongle and use it to quickly test other BLE112 modules.

Hardware configuration-

Requirements:

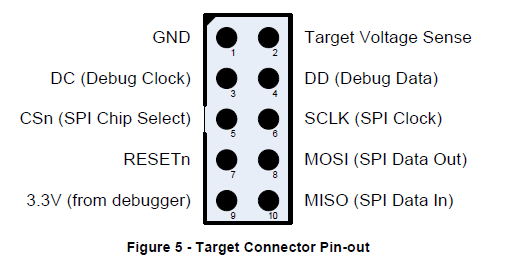
1. CC debugger (the one I am using is from Texas instruments)

2. Ble112 module from Bluegiga

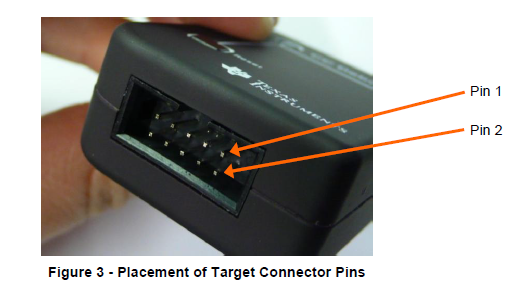
3. connecting wires and some LED’s and resistors

CONNECTIONS-

1. pins on the cc debugger are shown in this picture:



The pin numbering is as follows:

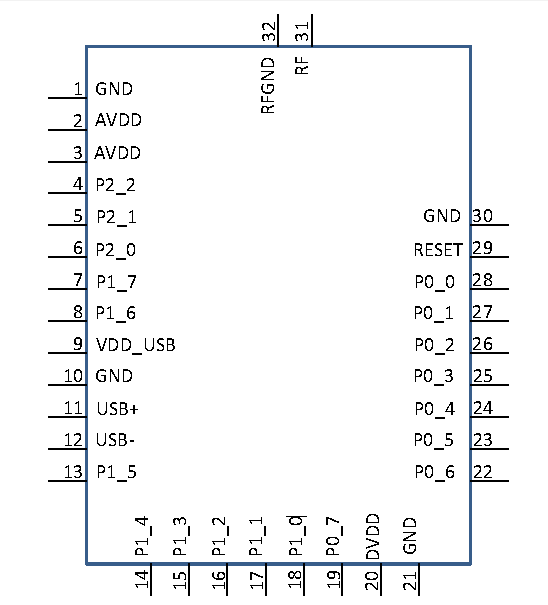


1. Soldering your ble112 module onto a board would be useful as the module is very small to work with.

< pic of the module soldered>

< attach the ble112 datasheet herewith>

1. the pins on the ble112 module are depicted here



4. We only need pin numbers 1, 2, 3 , 4, 7 and 9 from the cc debugger for this purpose.

5. Connect pins 30, 1, 10, 21 of the BLE112 module with pin number 1 of the cc debugger.

6. Then connect the pins 2, 3 ,9and 20 of the ble module with pins 2 and 9 of the cc debugger.

7. Connect pin number 4 of the ble module with the pin number 3 of the cc debugger.

8. Connect pin number 5 of the ble module with the pin number 4 of the cc debugger.

9. Finally connect pin number 29 of the ble module with pin number 7 of the cc debugger.

10. Now connect the cc debugger to your pc via a USB cable and u must see that the LED on debugger turns GREEN. If it does not then check ur connections, there are loose connections , I faced this problem, soldering would really help.  
11.With this your hardware configuration is complete,now we move on to installing the required softwares.  
  
  
Software Configuration:

1. Bluegiga-ble-1.2.1-91.exe

<attach the setup file if possible>  
Install this executable file in your system and this will install all the required softwares and the documents in your system.

2.BLE Software update tool:

A] After you attach the cc debugger with your pc you should see the cc debugger under the ports menu in your Device manager.As the system automatically installs all the necessary drivers.

B] Now open the BLE SW update software and in the ports menu your device must be shown, refresh if not.

C] Now click the info button, it tells u about the 3 parameters your module has:

#MAC address-actual address of the module which it uses to identify itself while advertising.

#License Key-Provided by the manufacturing company.

#Serial Number-Self explanatory

D] To upload a Project file browse to the project.hex or the project.bgproj file in the example folder under bluegiga, and click update.

E] This way you can flash .hex files into ur ble112 module.

\*NOTE: Do not use TI Softwares such as the Smart RF Flash programmer to flash into your modules as it tends to erase the MAC address in your device.

F] For burning the project files and creating out.hex files by using bgbuild.exe the procedure is as follows:

(the out.hex files are needed only when u are using the TI flash programmer, I do not recommend that but still I am giving the procedure to do that)

Open up a command prompt in Windows and navigate to the project directory, in my case, I stuck it in “**C:\Bluegiga\ble-1.2.1-91\example\toggle\_led**“

|  |  |
| --- | --- |
| Once there, execute the following command:  ..\..\bin\bgbuild.exe project.xml |  |

Now, if you open the directory that the **project.xml** file was in, you’ll notice there’s a file named **out.hex**. This is our bgscript that’s been compiled into a hex file that we can flash to the BLE112.

The next step is to open up the [Texas Instruments SmartRF Flash Programmer](http://www.ti.com/tool/flash-programmer), set the flash image path to the **out.hex** file, then click on “Actions: Erase and Program”, then hit the “Perform Actions” button, and u should see the status bar completed.

BLINKING AN LED ( THE TOGGLE LED EXAMPLE )

Now we will try to blink an LED using the input/output pins of our module.We will be using BGScript scripting language to do so, currently we are making a fully standalone device i.e. it does not need any external microcontroller to work as the ble112 modules already has a 8051 MCU in it which can be used for running simple applications such as this.

So what exactly is BGScript and why is it important? BGScript is a freely available proprietary language developed by Bluegiga for use with their BLE112 modules.  It allows us to write programs for the BLE112 without using the very expensive IAR compiler.  This, of course is very appealing, because IAR costs about $4,000 AUD, which is far beyond the typical hobbyist budget.  BGScript is a nice alternative for applications that don’t need the full complexity of a C based program.

For blinking an led you first have to understand about the various types of files and tools that we will be using, open the toggle\_led example and u would see various xml files and hex files .These all are required for proper functioning of our application.

1. Gatt.xml: for now you just need to know that the gatt.xml file gives us access to the GATT functionality of our module. It has the service ,characteristics descriptions .Also the GAP service is enabled using this document.  
   Do not panic if you were unable to get any of the above things, they are quite complex and I would repeat them in description again.
2. Hardware.xml: this file manages all the hardware interfaces of your module, the low power sleep modes, endpoints etc.
3. Cdc.xml: this file is about configuring the connection parameters of your module, the number of connections, type of connection etc.
4. Project.xml: this is the final file which uses all the above mentioned files and creates an output.hex file which has to be uploaded into the module. We have to mention the names of all the above created files in this so that it can access those files.
5. Out.hex : it is the hex file created after using BGbuild.exe as an analogy u can say like after burning the project.xml file we get this file.
6. Bgdemo.bgs: this is the file which contains our bgscript code. BGScript is is a simple BASIC like scripting language developed by the bluegiga people ,this makes developing applications very simple and we do not have to get into all that protocols for now.

Now before doing anything else first of all flash this Toggle\_led example into the firmware of ur module . Then connect a led on the pin number 19 of ble112 i.e. P0\_7.  
The LED should now toggle every second.

Now before knowing about other files you must get a good understanding of the code, the BGScript. I would walk u through each and every line of the code, come along.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# This example script will flash an LED connected to P0.7 every second

# Boot event listener

event system\_boot(major ,minor ,patch ,build ,ll\_version ,protocol\_version ,hw)

#Set timer to generate event every 1s

call hardware\_set\_soft\_timer(32768, 1, 0)

# configure P0.7 as output

call hardware\_io\_port\_config\_direction(0, $80)

# Disable P0.7 pin. Parameters: I/O port to write to 0/1/2,

# bitmask of pins to modify,

# bitmask of pin values to set

call hardware\_io\_port\_write(0, $80, 0)

end

dim result

dim port

dim data

#Timer event listener

event hardware\_soft\_timer(handle)

call hardware\_io\_port\_read(0, $80)(result, port, data)

if data & $80 then

call hardware\_io\_port\_write(0, $80, 0)

else

call hardware\_io\_port\_write(0, $80, $80)

end if

end

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Comments-all the lines of code that are written after a hash “#” represent the commenting in the code.

As the first comment says we will be able to toggle a led on P0\_7 pin every one second.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*# Boot event listener

event system\_boot(major ,minor ,patch ,build ,ll\_version ,protocol\_version ,hw)

…………

………..

end

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

From the Bluegiga Bluetooth Smart Software API Documentation file, it tells us “This event is produced when the device boots up and is ready to receive commands”.  So this is the primary entry point where we can configure the BLE module, sort of like a “main” function in C code.

When we power on the module it starts the boot process, here we listen to this boot EVENT and can use it to call various functions when the module is powered on. Do not worry about the parameters that are written inside the braces they just represent the boot up conditions which are of no use right now. Just u have to know that whenever we want to listen to an event we use “event ” followed by the type of event. Here the type of event is “system\_boot”. Finally we have to end our event to start a new one, I have not tried calling an event inside an event, you can try that too.

Now we will call various functions inside our boot event.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#Set timer to generate event every 1s

call hardware\_set\_soft\_timer(32768, 1, 0)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The first parameter defines how often the timer is fired, in units of local crystal frequency. In other words, time = 1/32768 seconds. Since we’re passing 32768, we’ll get a timer period of 1 second.  If we wanted a 2 second period, we’d use 2\*32768, or 65536.  The parameter for this time period is a 32 bit unsigned integer, so the largest period we can have for a timer interrupt is (2^32-1)/32768 seconds, or 36.41 hours. If you need a timer interval longer than this, you’ll need to write your own custom handler to take care of it.  The last parameter of the function defines whether this is a one time event or a continuously firing timer.  Since we want this timer to fire continuously, we’ll pass a value of 0. If we wanted a single shot timer, we’d pass a 1 here.

Invoking this function starts a timer that will be triggered every one second, now how do we use this timer as an event ,and how can we call functions based on this timer will be discussed later.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* # configure P0.7 as output

call hardware\_io\_port\_config\_direction(0, $80)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Now we will configure P0.7 pin as an output, for this we call the hardware io port configuration function. The first parameter “0” defines the port number of the pin. The module has 3 ports 0,1 and 2 . out of which only ports 0 and 1 have the io abilities.

The second parameter $80 is a hexadecimal value which tells up about the configurations of all the 7 pins in port number 0. The $ sign here represents that we want to input a hex value. This parameter is a BITMASK . it is not tough to understand , if we represent $80 in binary form we get 10000000 i.e. 6 zeroes after 1 and then again a 0. For a bitmask the lastbinary digit is always a 0. So in all we can say that we have 6 zeroes for pins 1,2,3,4,5,6 and 1 for pin number 7. Here 1 represents that it is configured as an output and 0 represents that the pin has been configured as an input.

As an example if we want to configure pin number 4 and 6 as outputs and all others as inputs we will write 01010000 whose hexadecimal value is $50.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Disable P0.7 pin. Parameters: I/O port to write to 0/1/2,

# bitmask of pins to modify,

# bitmask of pin values to set

call hardware\_io\_port\_write(0, $80, 0)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This call function is used for writing into the io ports.

The first parameter is the port number, the second parameter is the bitmask of the pins you want to configure, and the last parameter is the value you want to set the pins to. By doing this we configure the output voltage of pin P0.7 to be 0.

And we end the boot event now.

Next, we need to set up a callback to handle the firing of the timer that we configured earlier. Before we can define the timer callback, we’ll need to create some variables which we’ll be referencing later on.  These need to be defined outside of the event block:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

dim result

dim port

dim data

#Timer event listener

event hardware\_soft\_timer(handle)

…….

End

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This event is for listening to the hardware\_soft\_timer function call ,this events gets executed whenever the timer is invoked, in this case the timer is invoked every 1 second.so the code inside this event would be executed after every second.

Now we will look inside this timer call back event.

For this example project, I want to toggle an LED every second.  So in our timer callback event, we’ll need to get the current value of the pin and if it’s high, we set it to low, and vice versa. So the first thing we need is the current state of the pin:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

call hardware\_io\_port\_read(0, $80)(result, port, data)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Reading the current state of all the pins. In binary 1 would mean pin at high and 0 would mean pin at low.

Now let us first talk about the first pair of braces. These represent the parameters that u are giving as input to the function. The 0 here stands for the port number . The second parameter represents the bitmask of the pins that we want to read.

The second pair of braces represents the outputs given by the function, this is why we had defined these variables. The first return value**result** is the error code from the call, where 0 represents success.  The second return value **port** indicates the port that was read, and the last parameter **data** is the pin state, which is an 8 bit value representing which pins are high and which ones are low.

Now that we’ve got the current status of the pin, we can use a simple if/else conditional to toggle it:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if data & $80 then

call hardware\_io\_port\_write(0, $80, 0)

else

call hardware\_io\_port\_write(0, $80, $80)

end if

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

First parameter port number, second parameter bitmask of the pins to be configured, third parameter represents the state of the pins that u want them to be in, here we want pin 7 to be HIGH and all other outputs low (of any) so 10000000 i.e. $80.

Now you have got a fair idea about how the bgscript works, more functions and examples are present in the documentation.

For now do not worry about the other files, I will explain them at the correct time.

Our next target will be to change the blinking duration of the led and to have multiple led at other pins as well .

TUTORIAL #2

Now we will see how to change the toggle duration of the led and also will use other ports and pins.

The duration of toggle depends on the value we input at the time of call hardware soft timer. In the above example we set the parameter of duration as 32768.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

call hardware\_set\_soft\_timer(3\*32768, 1, 0)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The frequency of the inbuilt crystal is 32.768 kHz or its time period is 1/32768 seconds.

The value which we enter as the first parameter will be multiplied by the time period and the led will toggle for the resulting number of seconds.

As an example if you want the led to be on for 3 seconds and then off for 3 seconds we give the value of the parameter as 3\*32768.

Similarly u can also have a time period less than 1 second by dividing 32768 by some number.

Now try doing this and see what is the smallest time period u can attain so that u can easily distinguish between a blinking led and an ever on led.

Configuring pins 2 and 5 as output and attaching led onto all of them

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

call hardware\_io\_port\_config\_direction(0, $24)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

First write the bitmask for the pins u want to configure as output. Output=1 and input=0 (default).

So here we get 7th=0, 6th=0, 5th=1, 4th=0, 3rd=0, 2nd=1, 1st=0 , 0th=0 .

We get 00100100. Now find out the hex value of the above binary value , it is 24.

You can also change the port number by changing the first parameter. But remember you cannot use port number 2 in ble112 for io operations.

With this u have configured pins 2, 5 as outputs and now moving ahead we will see how to make the led on these 2 pins blink alternatively.

First the source code:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# This example script will flash LED connected to P0.2 and P0.5 alternatively every 0.5 second

# Boot event listener

event system\_boot(major ,minor ,patch ,build ,ll\_version ,protocol\_version ,hw)

#Set timer to generate event every 0.5s

call hardware\_set\_soft\_timer(32768/2, 1, 0)

# configure P0.2 and P0.5 as output

call hardware\_io\_port\_config\_direction(0, $24)

# Setting the initial values as 0

# I/O port to write to 0,

# bitmask of pins 2, 5 to modify,

# bitmask of pin values to set .

call hardware\_io\_port\_write(0, $24, 0)

end

dim result

dim port

dim data

#Timer event listener

event hardware\_soft\_timer(handle)

call hardware\_io\_port\_read(0, $24)(result, port, data)

if data & $4 then

call hardware\_io\_port\_write(0, $2C, $20)

else

call hardware\_io\_port\_write(0, $2C, $4)

end if

end

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Nothing much to explain in this , but I would like you to try out different things yourself but remember we do not have “else if” statement in BGScript, its not a powerful language.

TUTORIAL #3

Now we will start using wireless BLE communication. We would use ble112 as a server and TI CC2540 dongle as a client.

The TI CC2540 dongle is a usb supported device which can be easily plugged into any of your pc’s usb ports and it starts working. We will basically use it for discovering other advertising ble devices in the range and connect with them and control them.

This tutorial’s purpose is to teach how to make your ble112 module an advertising device via BGScript and to establish connection between these 2 modules.

We will be using the software called Btool to operate the TI dongle. Install the CC-254x-1.4.0 setup file into your pc and u are good to go.

Now it’s the time I should talk about the hardware.xml file and the gatt.xml file.

HARDWARE.XML

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**<?xml version="1.0" encoding="UTF-8" ?>**

**<hardware>**

**<sleeposc enable="true" ppm="30" />**

**<usb enable="false" endpoint="none" />**

**<txpower power="15" bias="5" />**

**<usart channel="0" mode="spi\_master" alternate="2" polarity="positive" phase="1" endianness="msb" baud="57600" endpoint="none" />**

**<script enable="true" />**

**<pmux regulator\_pin="7" />**

**</hardware>**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**The language used is basic xml. Under the hardware tag we define various hardware parameters and configure them here. As I told you we can either use BGScript for programming a standalone module or we can also use An external MCU for our module, in that case we have to use BGAPI protocol and we cannot use the BGSCRIPT . But here we are using the Bgscript language only as we have a standalone device so we make the “script enable “ tag as true.**

**We also have the option of controlling the power transmitted in the TX , using the “tx power” tag . but that is slightly advanced topic .**

**The ble112 module also supports USB and UART connectivity with a host. We have to configure whether we use these in our present application or not . More on connectivity is discussed later.**

**The rest are out of discussion right now. U would be wondering I did not really tell u much about above file, but believe me It is not important to know every thing here itself, u should just get an idea that some file called an hardware.xml exists and is very important .**

**Next is the gatt.xml file. But before going into gatt file I want to talk about profiles ,services and characteristics.**

**I will explain these three by taking an example of a man whose name is X. So this man X lives in Mumbai , goes to IIT Bombay and likes to watch football. The things that I just told u about X come under the Profile of X. Profile basically defines the type of person the man X is.**

**Under the profile of X we define the services that X can provide. For instance take the service of sports, X can play various sports so it can provide its service in various teams. X also provides other services such as Teaching and working. Let us explore the sports service of X, under sports X can play football, cricket and table tennis. These represent the characteristics of X under the service sports.**

**So in all we have characteristics->services->profiles**

**Our ble112 module also has these characteristics->services->profiles which are configured in the gatt.xml file. That is we can choose which characteristics should be there in our module and under which service the characteristics should come.**

**Many characteristics ,services and profiles have been already defined by the Bluetooth SIG people, we have the liberty to choose from them for our application.**

**To dishtinguish between these the SIG group has provided every characteristic Service and profile with a UUID (universally unique identifier), which is a 16 bit number.**

**Now we can move onto the gatt file.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**<?xml version="1.0" encoding="UTF-8" ?>**

**<configuration>**

**<service uuid="1800">**

**<description>Generic Access Profile</description>**

**<characteristic uuid="2a00">**

**<properties read="true" const="true" />**

**<value>Bluegiga Find Me</value>**

**</characteristic>**

**<characteristic uuid="2a01">**

**<properties read="true" const="true" />**

**<value type="hex">4142</value>**

**</characteristic>**

**</service>**

**<service uuid="1802" advertise="true" >**

**<description>Immediate Alert Service</description>**

**<characteristic uuid="2a06">**

**<properties write\_no\_response="true" />**

**<value length="1" />**

**</characteristic>**

**</service>**

**</configuration>**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**The typical structure of a gatt file can be easily seen from the above xml code. Under the configuration tag we define our services and under the services tag we define the characteristics and their UUID’s . The UUID of the respective service is also written adjacent to the service.**

**The first service defined has a UUID of 1800 as shown. This service is globally known as the GAP Service (Generic access profile), this is not an ordinary service it’s a special one. If we want our device to be advertising then we have to include the GAP service.**

**Under this service the first tag <description> is for writing the name of the service.**

**Then we define the characteristics under this service, we have the liberty to define any number of available characteristics under any defined service. Here we have 2 characteristics with UUID’s 2a00 and 2a01 .**

**There is one more service with UUID 1802 which also have advertising enabled true. This service just an ordinary service , not a special one like the GAP service. The name of this service is immediate alert service and it has one characteristic defined under itself.**

**Now let us go into detail about the characteristic tag, under the characteristic tag we have defined the properties of that particular characteristic . Under the properties tag we control the permissions offered by the respective characteristic, such as read/write/notify/indicate/constant etc. i.e. we have the liberty to completely shape up our characteristic’s properties. Then under the tag <value> we provide the value to the characteristic , the value can either be a hexadecimal value or a string.  
we have to initially define what kind of value we want and what length of value we want:**

**<value type=”hex” length=”6”>VALUE</value> for choosing a hex value of length 6**

**<value type=”UTF-8” length=”4”>VALUE</value> for choosing a string of length 4**

**So far so good, but now for making our module visible that is in advertising mode we also have to configure the BGScript , the bgscript code that we used in the toggle LED example will not work in advertising. We have to make a few changes.**

**To set the module into advertising and bondable mode we have to call the following functions:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**event system\_boot(major ,minor ,patch ,build ,ll\_version ,protocol\_version ,hw)**

**call gap\_set\_mode(gap\_general\_discoverable,gap\_undirected\_connectable)**

**call sm\_set\_bondable\_mode(1)**

**end**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**The first function sets the module in advertising mode under the parameters “general discoverable” and “undirected connectable”. General discoverable here means that our device can be easily discovered by any device which is scanning for ble devices. And The connection can be directed or undirected, in a directed connection the module knows the scanning device to which it wants to get connected and only that particular scanning device can pair up with our module, like a one to one communication, others can’t connect. Whereas in an undirected connection any scanning device can make a connection with our device .**

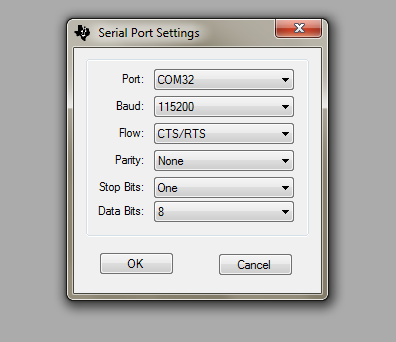
**The second function invokes the security manager and sets the device in bondable mode, yes we need permissions from the security manager every time we invoke any pairing/bonding/encrypting. The parameter (1) inside this function is for switching on bondable mode, if we want to switch off bondable mode then we pass a (0) here.**

**As you can see these functions are called at system boot, so whenever the module will start it will begin advertising and would be ready to connect.**

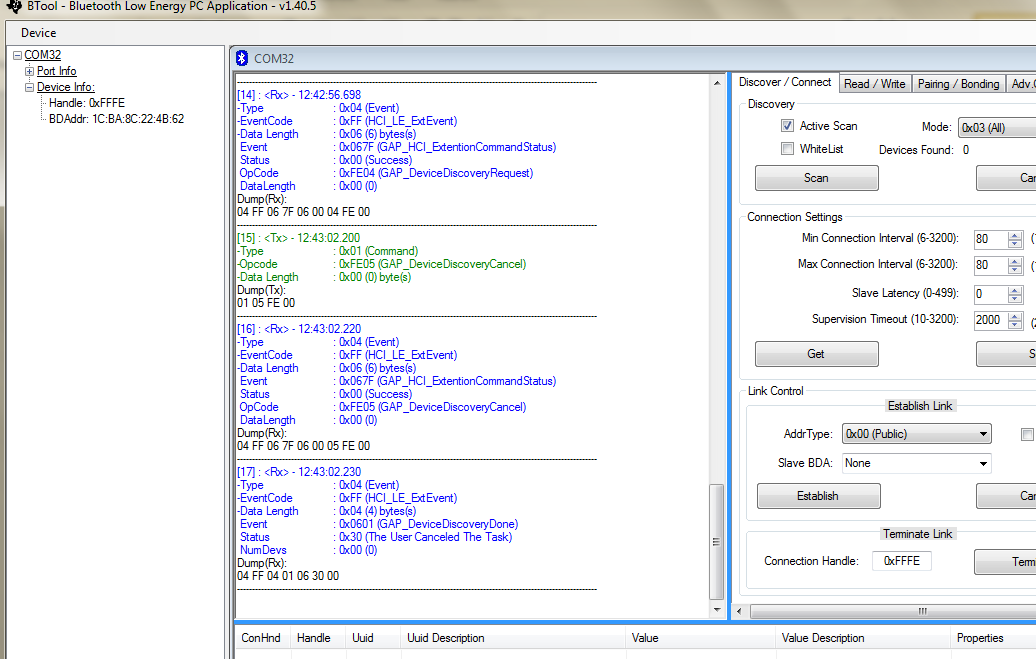
**This completes configuring our advertiser but we need to also know about how to listen to these advertising packets. So now we will move on to the TI CC2540 dongle configurations and understand how to work with the BTool.**

**For that you must install it first, the steps are given above. After installing you should see a Btool icon on your desktop , insert your TI dongle into usb ports and open the Btool.**

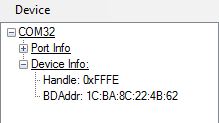
**The first thing you will see after starting btool is**

****

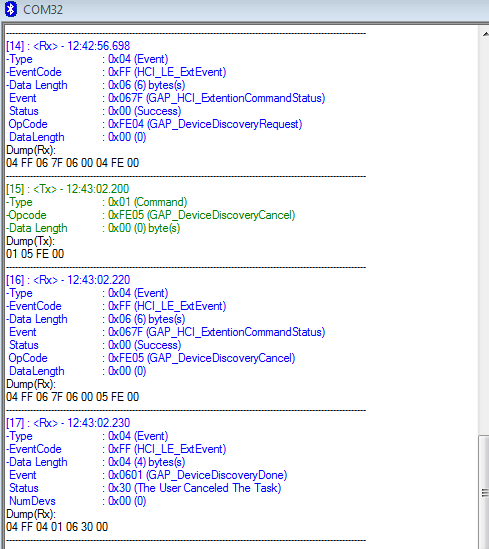
**Select the correct com port and the above settings and click ok.**

****

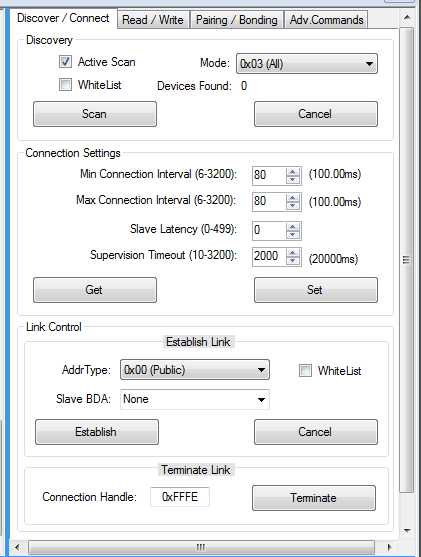
**Now let us look at various parts of btool one by one.**

****

**The left most window tells you about the com port, the device info and the info of the device to which your dongle is paired. Currently it is not paired to any device so u cannot see any device there.**

****

**The middle part shows you the description of the data sent and received between the devices. Hexadecimal numbers following the dump are actually the data packets, whose description is given above dump in blue script.**

****

**The right most part is the command window from here u can send commands and adjust settings. What the btool does is , it simplifies the commununication protocols for us and converts them into simple graphical user interface.**

**Now power on your bluegiga module( after flashing the new code into it) and press the scan button in btool. The btool scans for 10 seconds everytime and then u must see one device found. The MAC address of your bluegiga module is shown in the Slave BDA column below.**

**To establish a connection between these two modules press the establish button and then u should see that your devices are successfully paired and u can see your ble112 in the left most menu.**

**There is also a terminate button to terminate the connection and there is something called as connection handle. We will talk about handles and changing characteristics in later tutorials.**

**TUTORIAL #5**

**Objective of this tutorial is to change the blinking time of an LED wirelessly from the client by changing a characteristic value.**

**You can have your LED or any other actuators connected to any of the pins in ports 0 and 1 but not port 2 pins, as they do not support interrupts in current versions of ble112.**

**To start with connect both of your devices ble112 and TI dongle and connect them as per the instructions given in previous tutorial. Now we will look at the attribute read/write operations that can be performed form out TI dongle (the client) . That is we have the liberty to read and write attribute values of the server(ble112) from the client by using Btool.**

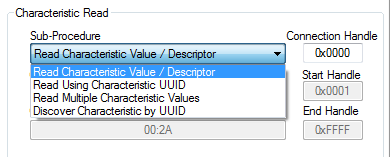
**Assuming that till now to connect you have already scanned for ble devices and established a connection between both of your devices, now go to the Read/Write column.**

**Upon connection the client assigns a connection handle for every connection that it makes, you can see the connection handle here in the read/write window. At present ble112 devices which are configured as “master” can support a maximum of 8 connections only. There is no limit on the number of connections in the actual definition of BLE but the blue giga people have a limitation for now.**

**There is a notable difference between a “characteristic” and an “attribute” and both of these terms cannot be interchanged. Actually when we define a characteristic in the gatt file we assign 4 attributes to completely define that characteristic. The 4 attributes that are defined for a characteristic are**

1. **Characteristic Declaration: used for writing and displaying the UUID.**
2. **Characteristic Value: in this attribute we input the value of the characteristic.**
3. **Characteristic Description: used for describing the characteristic with a string/name. and this name does get displayed to the client unlike the description for a service, that is the description that we write below a service declaration does not gets advertised and is only for commenting purpose.**
4. **Characteristic Configuration: used for enabling notifications and/or indications in the server device. To enable notifications we write 0x0001 in this characteristic and to enable indications we write 0x0002. It should be noted that the only difference between notifications and indications is that indications need to be responded to , that is a scan response is generated when client receives an indication whereas notifications might go unnoticed.**

**First looking at the read commands :**

****

**Inside the characteristic read menu on clicking the sub-procedure menu you will find 4 options:**

**1.Read char. Value /Descriptor-this option can be used to find the value of an attribute if you know the attribute handle. Again I should remind you that by using this we tend to read an attribute’s value form the attribute handle that we input. We do NOT read the characteristic, it should be made very clear.**

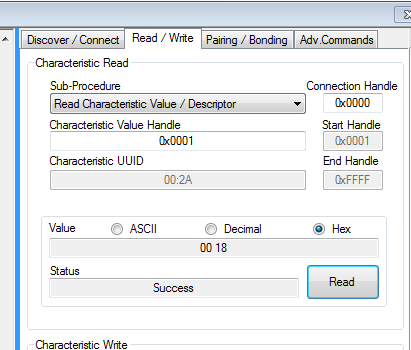
**2.Read using characteristic UUID: this option lets us read only the valuee of the characteristic whose UUID we enter, in simple terms it reads the value of the attribute number 2 in a characteristic as described above.**

**3.Read Multiple Characteristic Values: this is just an extension to the previous command, we can enter more than one characteristic UUID’s in here and we get the value of all of them at a time.**

**4.Discover characteristic using UUID: type just the UUID of a characteristic and you can get all the attribute values associated with the characteristic.**

**After selecting all the parameters press READ and the value appears in the value window, which also has the option ASCII ,Decimal and HEX.**

**The status window gives you the status of the executed command. “success” represents that the read was perfectly executed.**

****

**In the above example I have read the value of the 1st attribute that is the UUID declaration for the GAP service.**

**Now moving on to writing characteristic values. Input the attribute handle and the connection handle and then choose the type of value you want to input. We cannot write more than 20 bytes data into an attribute. input the data and then press the “write” button, if everything was fine you should see success in the status box.**

**But definitely you will be able to write to a characteristic value only if it has a “write” property enabled in gatt.xml !**

**And also one cannot change the UUID of any service/characteristic form the client so do not even try to write the UUID attribute handle.**

**Also you are not allowed to change the string description of the characteristic. You can only change the value of the characteristic and the characteristic configuration handle value.**

**Currently the application and the configuration files that you have flashed in your ble112(which were same as previous tutorial) do not have any characteristic which has a permission to write to . So we will now modify our code so that we can control the blinking time of an LED wirelessly.**

**We only need to change some parts of Gatt.xml and Bgscript(used in tutorial 2 and 3).  
In the gatt.xml file we only need to add one more service which has only one characteristic which stores the value of time interval we want to set. And also remove the immediate alert service defined earlier as it is of no use right now.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**<?xml version="1.0" encoding="UTF-8" ?>**

**<configuration>**

**<service uuid="1800">**

**<description>Generic Access Profile</description>**

**<characteristic uuid="2a00">**

**<properties read="true" const="true" />**

**<value>Bluegiga Find Me</value>**

**</characteristic>**

**<characteristic uuid="2a01">**

**<properties read="true" const="true" />**

**<value type="hex">4142</value>**

**</characteristic>**

**</service>**

**<service uuid="180f" advertise="true">**

**<description>timer</description>**

**<characteristic uuid="2a19" id="xgatt\_timer">**

**<properties read="true" notify="true" write="true" />**

**<value variable\_length="true" length="20" type="utf-8" />**

**</characteristic>**

**</service>**

**<configuration>**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**For now I am using a predefined service, which has been defined for some other use by the developers, it has some other name, I have modified its characteristics and that is all we need for now. You do not need to worry about this, I will tell you about self defined services and characteristics later.**

**advertise="true" using this tag here makes the UUID of the timer service to be advertised with all the other attributes.  
the id="xgatt\_timer" is a really useful thing when it comes to using characteritic values in the BGScript code. We have the read write and notify properties enabled for this characteristic.**

**In the value tag we jave defined what can be the maximum mength of the data and that the data can be variable in length. UTF-8 represents a format in which we can enter strings and numbers.**

**Doing this completes our gatt.xml file configuraton, now moving on to the script.**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# This example script will flash an LED connected to P0.7 every second and we can change the timing of flashing wirelessly from our ble client.

dim timer

dim result1

dim v\_len1, v\_data1(20)

# Boot event listener

event system\_boot(major ,minor ,patch ,build ,ll\_version ,protocol\_version ,hw)

call gap\_set\_mode(gap\_general\_discoverable,gap\_undirected\_connectable)

call sm\_set\_bondable\_mode(1)

call flash\_ps\_load($8000)(result1, v\_len1, v\_data1(0:v\_len1))

call attributes\_write(xgatt\_timer,0,v\_len1,v\_data1(0:v\_len1))

end

event attributes\_value(connection, reason, handle, offset, value\_len, value\_data)

if handle=$0003

call flash\_ps\_save($8000, value\_len, value\_data(0:value\_len))

call attributes\_write(xgatt\_timer,0,value\_len,value\_data(0:value\_len))

timer= value\_data(0:value\_len)

#Set timer dependent on our variable “timer”

call hardware\_set\_soft\_timer(timer, 1, 0)

# configure P0.7 as output

call hardware\_io\_port\_config\_direction(0, $80)

# Disable P0.7 pin. Parameters: I/O port to write to 0/1/2,

# bitmask of pins to modify,

# bitmask of pin values to set

call hardware\_io\_port\_write(0, $80, 0)

end if

end

dim result

dim port

dim data

#Timer event listener

event hardware\_soft\_timer(handle)

call hardware\_io\_port\_read(0, $80)(result, port, data)

if data & $80 then

call hardware\_io\_port\_write(0, $80, 0)

else

call hardware\_io\_port\_write(0, $80, $80)

end if

end

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The first most important change that we have made is the attribute value change event. This event is called automatically every time any attribute’s value is changed. And then we can make our code to differentiate between the change in different attributes by using the if statement.  
here I know the attribute handle in which my characteristic value is stored, so by using “if handle=$0003” I make sure that the code inside this if statement gets executed only when there is some change in that particular attribute. If u want to change multiple attributes then simply use multiple if statements.

This event gives us the attribute handle whose value has been changed and also gives us the new value which we can use. We store this changed value in a buffer value\_data(20), buffers are almost same as arrays.

We use this value and store it in our “timer” variable, which will control the blinking time of the led. The rest is the same inside this event as we did in the previous tutorials just replace 32768 form the hardware\_soft timer function call with the variable “timer”.

Now you are in a perfect stage to be told about the Persistent Store or the PS store. It is just a chunck of flash memory reserved for storing data that is used by the application from time to time. The PS store has various keys from memory address $8000 to $ 807F , at each memory address you can store data upto 32 bytes. Whenever you reset your ble112 module the code gets started again and it would eventually forget the attribute value which was written to it. So to make It remember the change we first store the attribute’s new value which we have written into the ps store by using the function:

call flash\_ps\_save($8000, value\_len, value\_data(0:value\_len))

The first parameter is the memory location where we want to store the attribute’s value. The third parameter is the value inputted to be stored and the second parameter is the length of the data. As value\_data is a buffer we use this kind of notation to use this.

The we call the attribute write function to write the value into the attribute.

call attributes\_write(xgatt\_timer,0,v\_len1,v\_data1(0:v\_len1))

Here we use the ID of the characteristic to point to the following characteristic.

But till now what we have discussed will not full fill our needs, we also need the ble112 device to load from the saved memory the attribute’s value whenever the module is restarted. To do this we call the flash load function from in the boot event and then again call the attribute write function here itself.

call flash\_ps\_load($8000)(result1, v\_len1, v\_data1(0:v\_len1))

call attributes\_write(xgatt\_timer,0,v\_len1,v\_data1(0:v\_len1))

The parameter in the first bracket corresponds to the memory location from where we want to load the value, a zero value of result1 tells us that the command was perfectly executed. We define new variables for data and its length in which we store the contents we load. Then we write this data again to the attribute.

This completes our current tutorial, and yeah, it is obvious but I should tell you that the first time you start the code the led would not toggle as no there is no value to “timer”, to make it work enter a value in the write command menu.  
also if you flash the module again then all the values in the ps store get erased, but it remains intact if you restart.

TUTORIAL #6

In this tutorial we would learn how to connect a ble112 device to the pc via UART and use it in BLEGUI software. Basically now we will eliminate the use of TI CC2540 and use a ble 112 device to do the same work with a new software.  
The hardware connection for connecting module via UART is very easy but there was no proper documentation on the internet to find it out, so It took me a lot of time to find it out.

As I told earlier you can either program your ble device via BGScript or you can use BGAPI commands, to use BGAPI you have to connect your module via UART to the pc. This tutorial is just an intro to the BGAPI mode of commanding and will not go into using the ANSI C library for BGAPI.

The Hardware configuration:

1. To connect via UART you will first need a FTDI chip FT232 RL (USB to Serial converter basically)  
   whose one end gets connected to the pc via a USB and the other end has outputs such as the RX,TX,3.3 V,GND and CTS,RTS. It is okay if you have a chip with only CTS as we do not really have to do anything with the RTS.
2. As you know the ble112 device needs 3.3 V for operation this FTDI chip would provide it, connect the power and the GND to the power and GND of your module which u had assembled for the led tutorial.
3. Connect RX of the FTDI port to TX of the ble112 device that is pin number 24 and TX of FTDI to RX of module that is pin 23.
4. Connect the CTS of pin of the FTDI to the common ground trough a 1kilo ohm resistor and let the RTS pin float independently i.e. do nothing to the RTS pin.

This completes the hardware configuration but this is not all. We have to configure the hardware.txt file in our code and make it feasible for UART communication.  
What we have basically done is faked flow control, so we are in a way using flow control and not using it too!

The hardware file configured for UART communication at a BAUD rate of 9600 looks like

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* <?xml version="1.0" encoding="UTF-8" ?>

<hardware>

<sleepoc enable="true" ppm="30" />

<usb enable="false" endpoint="none" />

<txpower power="15" bias="5" />

<usart channel="1" alternate="1" baud="9600" endpoint="api" mode="uart" flow="true" />

<sleep enable="false" />

<wakeup\_pin enable="false" />

</hardware>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The only highlight of the above file is the <usart> tag in which we have properties like:

1. Channel=”1” alternate=”1” : this sets pin number 23 and 24 as the communication pins for UART. We have 4 pairs of RX and TX pins for usage and we are currently using 23 and 24.
2. Baud=”9600” sets the baud rate at which the communication takes place , a low baud rate gives more stable and reliable data transfer.
3. Endpoint=”api” mode=”uart” configures the connection to be a uart connectionwith the endpoint set to api mode.
4. Flow=”true” is used for enabling the flow control which is a must.

Now save this file and flash it to module’s memory. Then connect the module with the pc and start the BLEGUI software, find your device in the ports menu, press attach and you are good to go. The rest is quite similar to the Btool, there is no need for me to tell you how to scan for devices, how to connect etc. as you can see everything is based on graphical user interface.

You also have the option of bypassing the GUI and using actual API commands written in hex values under the commands LOG.

For knowing more about the hexadecimal method and different functions you can have a look into the BGAPI configuration guide.  
so far so good for this tutorial, now what is left has to be completely done by you i.e. try to see all the characteristics in BLEGUI , try to change the values and do all that you have done with the Btool!

TUTORIAL 7

From this tutorial we will start working on a project entitled “Vehicle Identification”, basically we would install ble servers in every vehicle, these servers would advertise important information about the vehicle which include-

1. Vehicle Number
2. Vehicle Make
3. Vehicle type
4. Vehicle model
5. Owner name
6. Owner contact details
7. Insurance company details
8. Date of expiry of insurance
9. Toll data
10. Toll balance available
11. Last recharge date

We want the server to advertise this data to a client so we first configure the gatt.xml file of the server and add new characteristics, in total 16 of them.

We can either add the characteristics either under a single newly defined service or we may even have multiple services, I prefer to have just one single service as it is less of a headache and also there is no use of having multiple characteristic as we do not have the options of controlling the properties under a service.

If we had the option of controlling the properties of all the characteristics inside a service by controlling the properties of the service itself then definitely it would have been really good to have multiple characteristics.

Well the plan is as follows, customer or the vehicle owner buys vehicle and with it he buys a ble module, just the same way he has to apply for vehicle number he applies for the configuration of his ble device. As u can see the ble server even has the vehicle number in it and many other important information so it is very useful. The ble module then goes to the government official who has the responsibility and the power to configure the module or in simple words to add relevant data into the characteristics. He has special access to change the characteristics of the server which only an authenticated government official has.

Actually there are characteristics that have to be changed on a periodic basis and for these characteristics we will have to have other mechanisms. As an example consider the balance amount characteristic, definitely it cannot be a fixed/constant characteristic, it has to be changed time to time, the vehicle owner has to recharge his vehicle now in a same way he recharges his cell phone. Now vendors/retailers will buy credits from the government and they will have the authority to add balance to the vehicle owner’s ble server. So u can see that the characteristics will differ in their properties and usage, we will talk in detail about every characteristic later.

Here is the first gatt file which is the most basic one, in it all the characteristics are advertisable without any security considerations, that is anyone can view the characteristics and write into them without having to enter and security key. Slowly we are going to insert security exceptions into our application but for now this is good.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<?xml version="1.0" encoding="UTF-8" ?>

<configuration>

<service uuid="1800">

<description>Generic Access Profile</description>

<characteristic uuid="2a00">

<properties read="true" const="true" />

<value>Vehicle Iden.</value>

</characteristic>

<characteristic uuid="2a01">

<properties read="true" const="true" indicate="true" />

<value type="hex">0832</value>

</characteristic>

</service>

<service uuid="180f" advertise="true">

<description>Battery</description>

<characteristic uuid="2a19" id="xgatt\_battery">

<properties read="true" />

<value type="user" />

</characteristic>

</service>

<service uuid="47a9adfc-f7aa-11e3-865e-b2227cce2b54" >

<description>Vehicle Attributes</description>

<characteristic uuid="49c80d26-c536-4753-8bf5-16bd9574daba" id="vehicle\_number">

<description>Vehicle number</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="3ef805c3-6771-43f3-ba2d-90926dcf1f3e" id="vehicle\_make">

<description>Vehicle Make</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="158a5f06-b54f-4ad0-987c-23c3d5eb210f" id="vehicle\_type">

<description>Vehicle Type</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="17def42a-0e99-4d3f-a979-75632f70d7ab" id="vehicle\_model">

<description>Vehicle Model</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="2324d59a-d9f6-4cdb-a1b1-3ee3ca7c6c36" id="insurance\_company">

<description>Insurance Company</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="636d8b48-82e1-434a-9a1f-d6fb477544ec" id="insurance\_policy\_number">

<description>Insurance Policy Number</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="11500dde-1f52-4b1d-bb70-21e662759543" id="insurance\_expiry\_date">

<description>Insurance Expiry Date</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="b4528177-b13e-4c60-8ac0-8f50ce83dbe3" id="vehicle\_owner">

<description>Owner Name</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="bb98d960-5d32-482a-a789-cb8b2d2e553d" id="contact\_number\_one">

<description>Owner Contact Number 1</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="d14b2c77-4a89-4cea-af8b-d88ad3361043" id="contact\_number\_two">

<description>Owner Contact Number 2</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="27af3f6c-dea1-4ef0-a586-bcaf114bdd16" id="owner\_email">

<description>Owner Email</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="9a0687cf-906a-4f8e-9098-c381dd1f4acb" id="current\_balance">

<description>Current Vehicle Balance</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="0be2350a-f581-41d0-89e0-ba4fadd608c3" id="last\_tollcross\_date">

<description>Last Toll crossed date</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="f94333cb-2002-46a1-ba9c-76ba8fcd1efb" id="last\_tollcross\_code">

<description>Last Toll crossed code</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="7f588466-a4c4-4cef-9f52-4c97d3fd81d9" id="last\_tollcross\_amount">

<description>Last Toll Amount</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="9ef38934-6d5a-4c62-964e-0270d2fd9530" id="last\_recharge\_date">

<description>Last Recharge Date</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

</service>

</configuration>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The first step was to generate 128 Bit UUID’s for our 1 service and 16 characteristics. It should be noted that the 16 bit UUID’s that you can see in the GAP service have been previously defined by the Bluetooth SIG group and we the common people have the access to define only 128 bit long UUID’s which can be generated from [www.uuidgenerator.net](http://www.uuidgenerator.net)

So I generated 17 UUID’s and then used them here, u do not separately need to generate UUID’s for yourself until and unless u want to define a completely new characteristic. There is nothing special about UUID’s these are just randomly generated numbers through some machine.

So we defined one service and named it as “vehicle attributes”, but this description would not be advertised, only the UUID of the service could be advertised and the service would be seen as Unknown service.

Then we define 16 characteristics inside this service, give them their UUID’s and also give them an ID, this id can be any word which you want, completely your wish.

It should be noted that the description/name that we give to the characteristic is advertised and u can very well see this name. Then we give read, write and notify properties to all the 16 characteristics for now.

The code under the <value> tag says that the characteristic value is of the type UTF-8 (string and numbers both can be fed into this) and also says that the data length can be variable but not more than 20 which is the maximum length possible (obviously now u know that one letter/number takes up one byte in the memory).

Now the changes that we do in the BGScript have already been learnt by you !  
actually in the tutorial in which we blinked led wirelessly I bypassed a simpler method and taught you a bit more advanced method using ps store which is going to be used now. So now you do not need to learn anything new for creating BGScript file for this simple version of our project.  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

dim v\_data,v\_data0(20),v\_data1(20),v\_data2(20),v\_data3(20),v\_data4(20),v\_data5(20),v\_data6(20),v\_data7(20),v\_data8(20),v\_data9(20),v\_data10(20),v\_data11(20),v\_data12(20),v\_data13(20),v\_data14(20),v\_data15(20),v\_data16(20)

dim v\_len,v\_len0,v\_len1,v\_len2,v\_len3,v\_len4,v\_len5,v\_len6,v\_len7,v\_len8,v\_len9,v\_len10,v\_len11,v\_len12,v\_len13,v\_len14,v\_len15,v\_len16

dim result1,result2,result3,result4,result5,result6,result7,result8,result9,result10,result11,result12,result13,result14,result15,result16

event system\_boot(major ,minor ,patch ,build ,ll\_version ,protocol\_version ,hw )

call gap\_set\_mode(gap\_general\_discoverable,gap\_undirected\_connectable)

call sm\_set\_bondable\_mode(1)

call flash\_ps\_load($8000)(result1, v\_len0, v\_data0(0:v\_len0))

call attributes\_write(vehicle\_number,0,v\_len0,v\_data0(0:v\_len0))

call flash\_ps\_load($8001)(result2, v\_len1, v\_data1(0:v\_len1))

call attributes\_write(vehicle\_make,0,v\_len1,v\_data1(0:v\_len1))

call flash\_ps\_load($8002)(result3, v\_len2, v\_data2(0:v\_len2))

call attributes\_write(vehicle\_type,0,v\_len2,v\_data2(0:v\_len2))

call flash\_ps\_load($8003)(result4, v\_len3, v\_data3(0:v\_len3))

call attributes\_write(vehicle\_model,0,v\_len3,v\_data3(0:v\_len3))

call flash\_ps\_load($8004)(result5, v\_len4, v\_data4(0:v\_len4))

call attributes\_write(insurance\_company,0,v\_len4,v\_data4(0:v\_len4))

call flash\_ps\_load($8005)(result6, v\_len5, v\_data5(0:v\_len5))

call attributes\_write(insurance\_policy\_number,0,v\_len5,v\_data5(0:v\_len5))

call flash\_ps\_load($8006)(result7, v\_len6, v\_data6(0:v\_len6))

call attributes\_write(insurance\_expiry\_date,0,v\_len6,v\_data6(0:v\_len6))

call flash\_ps\_load($8007)(result8, v\_len7, v\_data7(0:v\_len7))

call attributes\_write(vehicle\_owner,0,v\_len7,v\_data7(0:v\_len7))

call flash\_ps\_load($8008)(result9, v\_len8, v\_data8(0:v\_len8))

call attributes\_write(contact\_number\_one,0,v\_len8,v\_data8(0:v\_len8))

call flash\_ps\_load($8009)(result10, v\_len9, v\_data9(0:v\_len9))

call attributes\_write(contact\_number\_two,0,v\_len9,v\_data9(0:v\_len9))

call flash\_ps\_load($800a)(result11, v\_len10, v\_data10(0:v\_len10))

call attributes\_write(owner\_email,0,v\_len10,v\_data10(0:v\_len10))

call flash\_ps\_load($800b)(result12, v\_len11, v\_data11(0:v\_len11))

call attributes\_write(current\_balance,0,v\_len11,v\_data11(0:v\_len11))

call flash\_ps\_load($800c)(result13, v\_len12, v\_data12(0:v\_len12))

call attributes\_write(last\_tollcross\_date,0,v\_len12,v\_data12(0:v\_len12))

call flash\_ps\_load($800d)(result14, v\_len13, v\_data13(0:v\_len13))

call attributes\_write(last\_tollcross\_code,0,v\_len13,v\_data13(0:v\_len13))

call flash\_ps\_load($800e)(result15, v\_len14, v\_data14(0:v\_len14))

call attributes\_write(last\_tollcross\_amount,0,v\_len14,v\_data14(0:v\_len14))

call flash\_ps\_load($800f)(result16, v\_len15, v\_data15(0:v\_len15))

call attributes\_write(last\_recharge\_date,0,v\_len15,v\_data15(0:v\_len15))

end

event attributes\_value(connection, reason, handle, offset, value\_len, value\_data)

if handle=$000c

call flash\_ps\_save($8000, value\_len, value\_data(0:value\_len))

call attributes\_write(vehicle\_number,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0010

call flash\_ps\_save($8001, value\_len, value\_data(0:value\_len))

call attributes\_write(vehicle\_make,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0014

call flash\_ps\_save($8002, value\_len, value\_data(0:value\_len))

call attributes\_write(vehicle\_type,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0018

call flash\_ps\_save($8003, value\_len, value\_data(0:value\_len))

call attributes\_write(vehicle\_model,0,value\_len,value\_data(0:value\_len))

end if

if handle=$001c

call flash\_ps\_save($8004, value\_len, value\_data(0:value\_len))

call attributes\_write(insurance\_company,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0020

call flash\_ps\_save($8005, value\_len, value\_data(0:value\_len))

call attributes\_write(insurance\_policy\_number,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0024

call flash\_ps\_save($8006, value\_len, value\_data(0:value\_len))

call attributes\_write(insurance\_expiry\_date,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0028

call flash\_ps\_save($8007, value\_len, value\_data(0:value\_len))

call attributes\_write(vehicle\_owner,0,value\_len,value\_data(0:value\_len))

end if

if handle=$002c

call flash\_ps\_save($8008, value\_len, value\_data(0:value\_len))

call attributes\_write(contact\_number\_one,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0030

call flash\_ps\_save($8009, value\_len, value\_data(0:value\_len))

call attributes\_write(contact\_number\_two,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0034

call flash\_ps\_save($800a, value\_len, value\_data(0:value\_len))

call attributes\_write(owner\_email,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0038

call flash\_ps\_save($800b, value\_len, value\_data(0:value\_len))

call attributes\_write(current\_balance,0,value\_len,value\_data(0:value\_len))

end if

if handle=$003c

call flash\_ps\_save($800c, value\_len, value\_data(0:value\_len))

call attributes\_write(last\_tollcross\_date,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0040

call flash\_ps\_save($800d, value\_len, value\_data(0:value\_len))

call attributes\_write(last\_tollcross\_code,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0044

call flash\_ps\_save($800e, value\_len, value\_data(0:value\_len))

call attributes\_write(last\_tollcross\_amount,0,value\_len,value\_data(0:value\_len))

end if

if handle=$0048

call flash\_ps\_save($800f, value\_len, value\_data(0:value\_len))

call attributes\_write(last\_recharge\_date,0,value\_len,value\_data(0:value\_len))

end if

end

event connection\_disconnected(handle,resultx)

call gap\_set\_mode(gap\_general\_discoverable,gap\_undirected\_connectable)

end

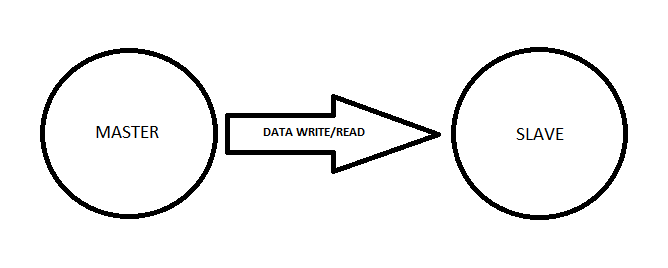
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*This might seem a big code but it is due to the fact that we have 16 characteristics so we need more lines of code.

What we have done here is exactly the same what we did in tutorial number 5, just the difference is that we have done it 16 times and there we did it just once.

I hope you do not have any problem in the above script as it has been already covered in tutorial 5.  
now we would move on to differentiating our characteristics and improving their security functions from the next tutorial.

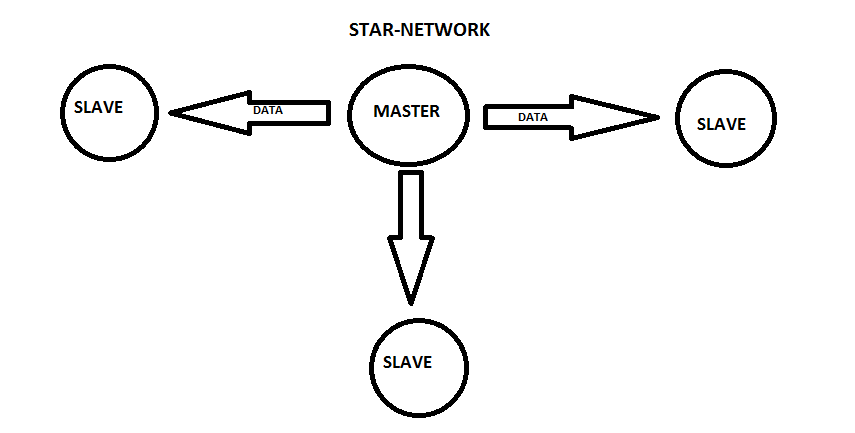
TUTORIAL #8

Putting the security configurations of the above project on hold, now we are going to start to build networks using ble devices. First we will configure the devices in a kind of linear network, in which each device has to be fed with the address of the devices to its left and the devices to its right and also with its own address. It should be kept in mind that the current ble methods only support one to one and star networks, but we can build other types of networks in pseudo forms by using switching between the master and slave modes, as the switching between different modes is quite fast so it gives us an illusion that we are actually using linear networks. But we can do data transfers with this in the same way as in linear networks.

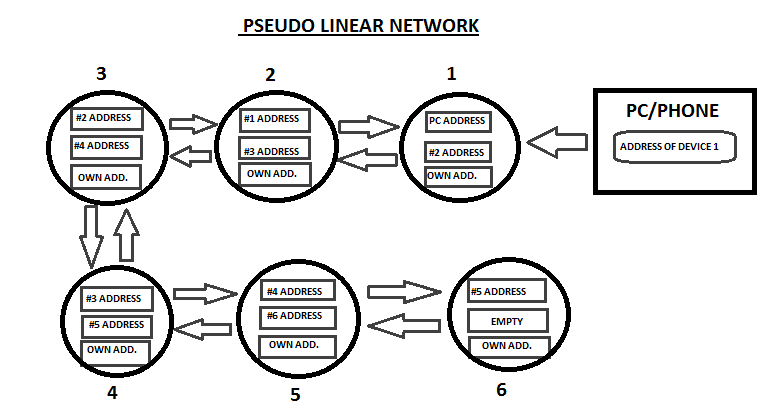


ONE TO ONE CONNECTION

The principle we are going to use is that these devices can switch between master and slave modes pretty quickly, so actually we get to read write data from both devices, just we have to change the previous master to slave and the previous slave to master. A master can have up to 8 slaves with whom it can get connected, but one slave can have only one master( note that this limitation of 8 slaves is with bluegiga modules, actually according to Bluetooth 4.0 specifications we can have unlimited slaves).



Now the next figures explain about the pseudo linear network that we are going to build.



As u can see every device in the network contains the address of both of its immediate neighbours, so in all it knows its own position in the network.

There has to be three kinds of devices in this network:

1. The first device which is connected to the PC/phone.
2. The devices that are in between any two Bluetooth devices.
3. And the one in the end of the chain.

The gatt.xml file is same for all these three kinds of devices but the BGScripts slightly differ for them.

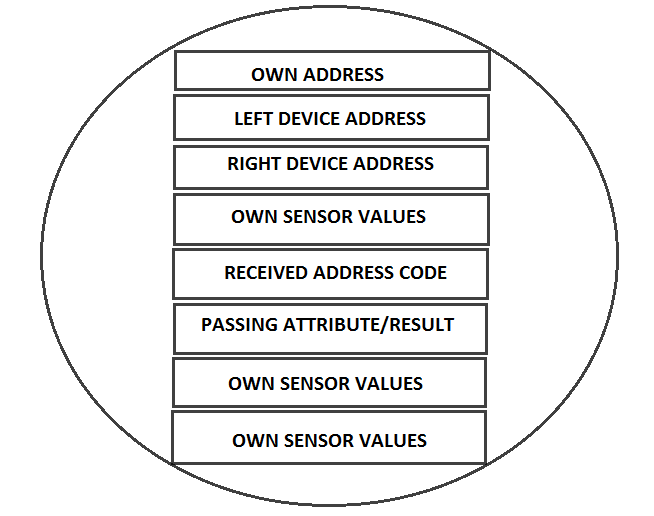
These devices in the current example code have 3 characteristics in which their sensor values can be stored, our main aim is to have a network in which we feed the address (of the device from which we want to read the characteristic value) and the characteristic handle (which we want to read) from the pc/phone and then we this data must travel through the network unless it reaches the concerned device and it acknowledges the data and then sends the respected characteristic value to the host.

Actually the device which is closest to the PC cannot write anything to the PC, only the PC can read information from this device number 1. So finally the respected characteristic value reaches to device 1 and we can read it from there.

The devices have one characteristic defined in which the then master writes the address code which is travelling through the network to the then slave.

All the devices also have one more characteristic defined which contains this passed on result/value.

This figure explains all the characteristics contained in all the devices.



Now to explain the mechanism better I we talk about what is happening inside when the PC asks for characteristic value of handle $12 from device number 4 (as an example).

1. Initially all the devices are in connectable, discoverable advertising mode. Currently these are advertising their own address and any scanning device can find them. Being in this scan able advertising mode ensures that these devices want to be paired as a slave.
2. Now the pc scans for device number 1 and establishes connection with it if it receives advertisement packets from 1. PC being the master and device 1 being the slave.
3. The PC writes the address of the device and the characteristic value handle into the “received address code” characteristic of device number 1. The format of writing from a TI Btool software is like D6:04:69:80:07:00:12, where the first 6 bytes represent the address 00 07 80 69 04 D6 written reversely and the last byte represents the characteristic handle in hex.
4. After the characteristic value is written into the perspective “address received” characteristic of device 1, it automatically disconnects with the PC.
5. Then device one matches the address received from PC with its own address, as it would definitely not match then device 1 would know that the PC does not want its address and it would now pass on this received address.
6. In order to pass on the received address it enters scanning mode, to become a master now!
7. As it has the address for device 2 fed inside it, it searches if device 2 is advertising or not, if yes then it connects “directly” to device 2. This direct connection is another feature which is available for us within BLE devices, according to this a master can look for advertising packets only from a selected number of devices whose address has already been fed into it and connect to them if it finds them advertising. Whitelist is a list of device address which are there for direct advertising, but we are not using whitelist here so do not worry.
8. So as device 2 was in advertising mode device 1 would receive these packets and then automatically connect to device 2 and write the received address code into the “ address received characteristic” of device 2.
9. Now device 2 upon receiving a write in its “received address” attribute would disconnect from device 1. Device 1 at this time would be programmed to enter advertising mode again but now it knows that it has passed the address from its right to its left( this right refers to the side from which the command came, and left the opposite to it) . You would be wondering what do I mean when I say that the device 1 now “knows that it has passed the address from its right to its left” well actually I have defined certain variables and I am using them as flags , that is I just change the values of certain variables and then every time I execute commands I execute them after checking these variables.
10. This time also the address does not match when device 2 checks it so it does the same process as device one and look and connect to device three and write the “received address code” onto device 3. And device 2 also goes to adverting mode same as device 1, ready to receive a value from left neighbour.
11. So what now? Yes, exactly. The address does not match with the address of device 3 also so it now connects and writes to device number 4 and gets into an advertising state same as devices 1 and 2.
12. This time the address would match when device 4 would compare with its own address. Then it would look for the last byte of the received code, as told earlier the last byte contains the attribute handle of the characteristic which the PC wants to read. It should be noted that the value that PC writes (the received address code) is stored in a BUFFER, in buffers we have the access to every byte individually just like arrays. So the last byte of the received address code is only seem if we have the initial 6 bytes of the buffer matching with the address.
13. Device number 4 now knows that it is the one whose value the PC wants, so now it would not write the value to device number 5 just devices 1,2 and 3 but it would enter scanning mode and scan for the device which is to the right, the device which wrote onto its “address received characteristic”.
14. As device 3 was advertising device 4 would easily receive its advertising packets and connect to it.  
    The difference in this connection from the last connection is that this time device 4 is the master and device 3 is the slave. After connection device 4 would write the characteristic value into the “passing attribute” characteristic of device 3.
15. Now as device 4 writes a value to a device which is to the right of it, it is programmed to disconnect with device 3 and get completely reset and start the general discoverable advertising again.
16. As device 3 receives a value in to its “passing attribute” characteristic it exits the advertising mode and starts scanning mode, to scan for the device right to it and directly connects. After getting connected to device 2 it writes into the “passing attribute” of device 2 and disconnects. After disconnection device 3 also gets completely reset.
17. Device 2 also does the same scans->connects->writes->disconnects->resets.
18. Now finally device 1 receives the value on its “passing attribute” and when it receives it also goes to complete reset. Mind you that the result value has been stored into the PS store in the flash memory of device 1 and does not gets erased upon reset, PS store gets reset only when we upload a new application.
19. So, how does the PC gets the value? Actually I have still not been able to figure out how to configure the PC as a slave device so what we have to do is to again connect to device 1 and read the value on the “passing attribute”.

This was a brief explanation of the algorithm we are going to use. Now we shall start with the actual code first we would configure the gatt.xml file for this.

The gatt file is same for all the devices.  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<?xml version="1.0" encoding="UTF-8" ?>

<configuration>

<service uuid="1800">

<description>Generic Access Profile</description>

<characteristic uuid="2a00">

<properties read="true" const="true" />

<value>Networking node #1</value>

</characteristic>

<characteristic uuid="2a01">

<properties read="true" const="true" indicate="true" />

<value type="hex">0832</value>

</characteristic>

</service>

<service uuid="180f" advertise="true">

<description>Battery</description>

<characteristic uuid="2a19" id="xgatt\_battery">

<properties read="true" />

<value type="user" />

</characteristic>

</service>

<service uuid="47a9adfc-f7aa-11e3-865e-b2227cce2b54" >

<description>attributes contained in node</description>

<characteristic uuid="49c80d26-c536-4753-8bf5-16bd9574daba" id="own\_sensor\_1">

<description>Own sensor value1</description>

<properties read="true" write="true" notify="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="3ef805c3-6771-43f3-ba2d-90926dcf1f3e" id="address\_received">

<description>address received</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="158a5f06-b54f-4ad0-987c-23c3d5eb210f" id="right\_device">

<description>Right device</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="17def42a-0e99-4d3f-a979-75632f70d7ab" id="left\_device">

<description>Left device</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="2324d59a-d9f6-4cdb-a1b1-3ee3ca7c6c36" id="passing\_attribute">

<description>Passing attribute</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="11500dde-1f52-4b1d-bb70-21e662759543" id="own\_sensor\_2">

<description>own sensor value2</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="b4528177-b13e-4c60-8ac0-8f50ce83dbe3" id="own\_address">

<description>own address</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

<characteristic uuid="9ef38934-6d5a-4c62-964e-0270d2fd9530" id="own\_sensor\_3">

<description>own sensor value3</description>

<properties read="true" notify="true" write="true" />

<value variable\_length="true" length="20" type="utf-8" />

</characteristic>

</service>

</configuration>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

As told earlier the BGscript files differ slightly for the three kinds of devices.  
but before moving on to the application something more on buffers:

1. To define a buffer of “n” bytes: buffer(n)
2. To access the (x-1)th element of a buffer: buffer(x:1)
3. These things are pretty similar to arrays.
4. To compare two buffers use the function:

Memcmp (buffer1( byte number to start comparing from),buffer2( byte no. ...), number of bytes to compare)

Now moving on to the application code, go to the attached bgscript files, their everything has been explained, it was not feasible to paste a properly commented and tabbed code.

First go through networking node #1, which contains the code for the device closest to the PC.

Then move on to networking node#2 which contains the code for the middle devices.

Finally have a look at networking node #3 for the device at the tail end.

After going through the codes and reading each and every line come back here and follow the next steps.

Now flash the codes into different devices, u must have at most 1 device which has to be next to the pc and at most one device , and u can have an infinite number of devices at the middle of the chain.

The main problem or rather setback of this network is that you will have to configure all the devices by entering details like the address of neighbour and own address.  
to do that connect to every device one by one with your PC (I am using the TI CC2540 dongle) and write onto the characteristic values, the left device address, the right device address, the own address, and also write different values into the own\_sensor characteristics of all the devices so that u can easily check the working of the network. Later on you can use these characteristics to store data from any sensor and access that data form anywhere.

Now after you have finished configuring all the devices connect to the first device and write the address code into the address received characteristic, followed by the address write the characteristic handle which you want to read. Then press write, the connection would be disconnected if u have entered an address other than this first device address. Wait for a minute (atmost) and let the data flow through the network.

Finally connect again to device 1 and read the attribute value at “passing attribute” characteristic, this is your required characteristic value !!

Try again and again and find out bugs in the application code if any.

The next target is to build networks (pseudo mesh) in which we do not have to configure the devices, simply use them.