11th INTERNATIONAL CONFERENCE APPLIED COMPUTING

25 - 27 October 2014

CODE GENERATOR FOR BLUETOOTH LOW ENERGY SERVICES

ABSTRACT

The Bluetooth Low Energy (BLE) is an important part of the revolution that started around the internet of things, namely to connect smartphones to all kind of devices, like watches. The BLE is a service-oriented architecture, where one of the devices assumes the role of server (also designated by *central*) and the other assumes the role of slave (also designated by *central*). The server is the device that contains the data and the slave, the one that requests data to the server. The smartphones typically play the central role, motorizing and controlling one or more peripheral devices. The BLE standard includes a set of profiles, each one defining a service, and the idea was to promote a normalize set of services, that should be supplied by the hardware manufacturers, to promote the fast adoption of the BLE technology by the software developers community, namely the developers of the Android, iOS, Windows Phone and others mobile devices operating systems. As consequence, it was quite fast to the market be invaded by a significant number of software applications, even when most of the smartphones, tablets and computers available on the market are not yet prepared for this technology. But the expectations are very high and the number of profiles included on the standard is restricted and, of course, does not cover all the necessities. The implementation of new services is not a simple task. The technology is too fresh (there are few examples, documentation, support and experts available on the market); the implementation is very dependent of the chip characteristics and resources; the BLE specification is not very accessible and uses a wide range of technologies; the available implementations of the BLE stack use very distinct architectures, implying distinct ways of implement the services; and all the code is implemented at a very low level, with all the natural constrains and difficulties common to this level. Confronted with all these problems, the authors of the paper implemented a code generator to assist on the development of BLE services. Presently, the generator, which is a prototype, produces C code for only one family of chips, the nRF51 - one of the most used BLE chips of the market. But the code generator architecture, based on the builder design pattern, ensures that the expansion of the generator, to produce code for other chips, is possible and not difficult to do. The implementation details of this generator are explained along this paper.

KEYWORDS

Code generation, Bluetooth Low Energy, Builder pattern

1. INTRODUCTION

The internet of things, the ubiquitous computing and the sensor networks are the new technological revolution that already started and that will change our world. In less than ten years the expectation is that almost everything around us will be interconnected. It will not be only the computer, the mobile or TV. The dog, the garbage collector, the car, the wallet, the kids, everything will be connected, transparently for us and using smart solutions that will put the technology to our service in a way never achieved before. Our world will suffer deep changes that will increase our life quality as individuals, but also as society. It is expected to reach unparalleled levels of efficiency, in all aspects of our reality. This revolution already started. There are many cases of success in the industry, agriculture, public services, security, leisure, and on many other areas.

Table 1. BLE market values (Decuir, 2010)

Area	Total Market (devices)
Mobile Phone Accessories	> 10 Billions
Smart Energy (meters and monitors)	~1 Billion
Automation	>5 Billions
Health, Sport and Fitness	> 10 Billions
Assisted living	> 5 Billions
Tags Animals (Food Safety)	~3 Billions
Intelligent Transport Systems	> 1 Billion
M2M (devices connected to the Internet)	> 10 Billions

Recent studies point to more than 50 billion devices connected until 2020. A market that represents, in the next 10 years, 14.4 trillion dollar (J. Bradley, 2013)of net profit for the companies positioned in these technological areas. The Bluetooth Low Energy (BLE) (Anon., n.d.) also known by Bluetooth Smart, is part of this technological revolution. It is increasing the number of chips with this technology, however, the implementation of solutions is not trivial, because the standard involves many details and the implementation is done to a very low level, being necessarily constrained by the resources available, like timers, memory, real-time clocks and others. By the author's experience, achieve a satisfactory level of expertise to produce functional and consistent solutions, can take several weeks or even months. The technology is also very recent and as consequence there is a natural lack of documentation and technical support. This paper presents the implementation details of a code generator for BLE services. This code generator, that is presently a prototype, can produce the C files with all the necessary code to implement BLE services for the *SoftDevices* (nRF51 SDK, 2014) software stack, used on the family of chips nRF51. The code generator was conceived based on the builder design pattern enforcing the separation of the parsing process from the generation process. This solution has several advantages, as it is explained in Section 4. The paper is organized as follows: section 2 introduces the BLE, namely the GATT level that supports the services; section 3 introduces the syntax defined for the code generator; section 4 explains the architectural design of the generator; and section 5 wraps up the paper with the conclusions.

2. BLUETOOTH LOW ENERGY

BLE technology is an extension of the Bluetooth 4.0 standard and was introduced by SIG in late 2009 and optimized specifically for devices that use small batteries and require very low consumption (Lee, 2011). Devices that support Bluetooth Low Energy are called Bluetooth Smart Devices and certified by the SIG. Operate in the same ISM band than traditional Bluetooth devices, and this is being divided into 40 channels, with 3 for the process of advertising and 37 for data communication. The major advantage compared with traditional Bluetooth is a significant reduction in consumption. This is possible due to the simplification of the search and connection process to devices. Another key feature for this decrease is its reduced activity window, sending small data packets for a few seconds and going into standby mode, the remaining time. Compared to previous Bluetooth versions, are used smaller data packets (2971 bits), a data transmission rate lower (0.26 Mbit/s) and the radius of coverage is limited to 50 meters, half the maximum allowed by earlier versions. Reinforcing, depending on usage can point to a consume up to 100 times smaller, and even in the worst scenario half the consumption of previous versions. This provides greater autonomy, being shown in studies

that the battery life may be many times larger than the device, allowing solutions that would otherwise be unfeasible. The standard also requires that the physical devices operate at very low voltages allowing the use of simple watch batteries. The manufacturers are also providing this technology at a low cost and in very compact solutions, which allows the existence of devices slightly larger than a coin. Being a standard accepted by most manufacturers, including the main smartphones manufactures, assures high levels of integration and interoperability.

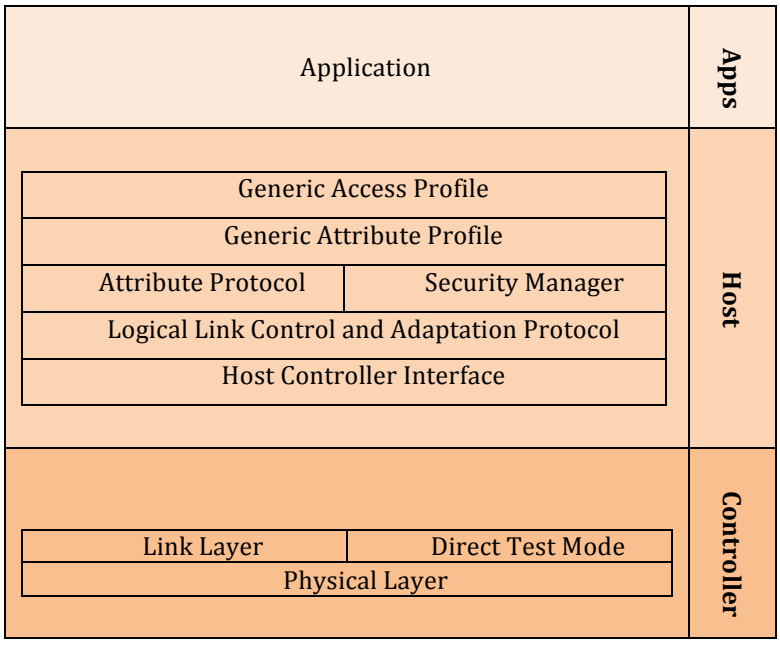
Pairing, connection and transmission

The type of communication is master-slave, the master being responsible for the establishment of the network connection. To establish a connection between devices, the master device sends a discovery message, using broadcast for devices in range, switching between the frequencies used by Bluetooth. All slave devices, with the active mode of discovery hear the discovery messages, and send a response to the master device, containing the address and device class. A connection request can exist depending on the desired (A. Huang, 2007).

The BLE Protocol Stack

The BLE protocol stack is partitioned into controller and host. The controller is responsible for managing the lower layers of the stack, particularly the capture of physical packages and control of the RF circuit. The other components are: logic control layer connection and adaptation (L2CAP), Generic Access Profile (GAP), Security Manager (SM), Attribute protocol (ATT) and the Generic Attribute Profile (GATT). For a better understanding, the entire stack can be seen in the following Table 2.

Table 2. Bluetooth Low Energy Stack Architecture



In this article, we only take in detail the GATT layer, because it will be the only relevant for that purpose.

GATT

The Generic Attribute Profile (GATT), which is responsible for describing the different frameworks of services and is an extension of ATT that is specific to the BLE 4.0. It provides the interface to the application layer through the application profiles. Each application profile defines the formatting of data and how they can be interpreted by the application. The profiles increase energy efficiency by reducing the amount of data to be exchanged. These are designed for specific functionality, e.g. there are heart rate, glucose, notification alerts and several other profiles. This makes it easier for developers to create applications for purposes of specific features by using pairs of default values/attributes found in each profile.

A GATT service it is a collection of related characteristics that work together to perform a specific function. Each GATT service has a number of characteristics. The characteristics storing useful values for services and its permissions. For example, the thermometer service includes characteristics for a temperature measurement value that is read-only, and a time interval between the measurements to be read/written. An example of a GATT service as well as their characteristics is shown in the Table 3.

Table 3. Example of a GATT service

Handle	UUID	Description	Value
0x0100	0x2800	Thermometer service definition	UUID 0x1816
0x0101	0x2803	Characteristic: temperature	UUID 0x2A2B
0x0102	0x2A2B	Temperature value	20 degrees
0x0104	0x2A1F	Descriptor: unit	Celsius
0x0105	0x2902	Client characteristic configuration descriptor	0x0000
0x0110	0x2803	Characteristic: date/time	UUID 0x2A08
0x0111	0x2A08	Date/Time	1/1/1980 12:00

Each characteristic has at least two attributes: the main (0x2803), which defines the universally unique identifier (UUID), and the attribute value. They may also be other extra attributes called descriptors, which serve, for example, to identify the measurement unit or any other information relevant of the characteristic. The GATT knows that the handle 0x0104 is a descriptor that belongs to feature 0x0101, because this is not the attribute value, as the attribute value is known to be 0x0101. Each service may define their own descriptors, but the GATT defines a standard set of descriptors that cover most of the cases, for example: numeric and presentation format, readable description, the valid range or extension properties.

Table 4. GATT structure

	Handle	UUID	Permissions	Value
Service	0x0001	Service	READ	HRS
Characteristic	0x0002	CHAR	READ	HRM
Characteristic	0x0003	HRM	READ/NOTIF	80 bpm
Descriptor	0x0004	DESC	READ	NOTIFY

3. THE LANGUAGE SPECIFICATION

Before explain the details that involve the language specification, it is important to say that the present prototype does not include solutions for all available features of the BLE stack. The support for new features will require some changes on the syntax, on the build routine and on the concrete builders. But the essential of the syntax and architecture will remain untouched. Each service is identified by an address composed by 32 hexadecimal digits. To make the generated code more legible, it is used a name that works like a prefix for the functions, structures and other elements. These two elements are specified as it is illustrated on Figure 1.

```
Syntax:
service –> BLESERVICE ( prefix, address){ predef dis_connection
characteristics }

Example:
BLESERVICE("WGEN", 0x2D26000057377FEE961BA8DB441BC2AC){ ...}
```

Figure 1. Service definition

Associated to the service, it is possible to have pre definitions (*predef*), which are described as C code surround by %\{ // C code }% and procedures that must be executed on the connection and on the disconnection events (*dis_connection*). These procedures should be defined using C code, like it is showed on Figure 2.

```
dis_connection –> connection | disconnection
connection –> ONCONNECT:%{ // C Code %}
disconnection –> ONDISCONNECT:%{ // C Code %}
```

Figure 2. OnConnect and OnDisconnect procedures definition

For each characteristic, it is request a prefix to be used as distinctive element on the structures, functions and other components. Each characteristic has also an address with 32 digits that only differs from the service address in four digits. To avoid the introduction of the 32 digits, the user only has to supply these four digits (as a hexadecimal value). The BLE supports five distinct access types to the characteristics: *read* (r); *write* (w), *write without response* (o), *notify* (n) and *indicate* (i). The *read* and *write* are, respectively, to read and write the characteristic; the *write* without response} is similar to the *write* but there is not any confirmation at the application level; the *notify* access is used to notify the slave whenever the value of the characteristic is changed on the stack; and the *indicate* access is similar to the notification, but there is a confirmation of the message deliver. The user can use several access types for each characteristic. It is also necessary to define the type of value associated to the characteristic and the interval of accepted number of values. The type of value is defined using the equivalent identifier of C language (uint8_t, float, char, ...). The interval is defined based in two integer or using C expressions surround by %{ //C expression }%, as it is illustrated at Figure 3.

```
Syntax:
characteristic –> prefix ( address, accesstype, typevalue, minnumber,
maxnumber)%{
```

```
// C code
}%

Example:
temperature( 0x20AA, [rn], uint8_t, 1, %{len}%)%{ ... }%
```

Figure 3. Characteristic definition

At the end of the definition (see Figure 3) there can be zero or more blocks of C code surrounded by *%{ //C code }%*, one per type of access used for the characteristic. Each one corresponds to the code that will be executed when is done the access by the correspondent type. Notice however that some access types, like the notify, indication and sometimes the read, do not execute any kind of procedure. But once defined one, the user should define all the others, even when they do not use it (using an empty block). Figure 4 shows a full example of a BLE service specification that will result, after being processed, into two files (*.h and *.c) with more than 250 lines of code.

```
BLESERVICE ("WGEN", 0x2D26000057377FEE961BA8DB441BC2AC){
PREDEF:%{
typedef struct per{
uint8_t var1;
uint8_t var2;
} Period;
}%
ONCONNECT:%{counter=0;}%
CHARACTERISTICS:
temp1( 0x20AA, [w], uint8_t, 1, 1)%{x=10;}%
temp2( 0x30BB, [rn], Period, 1, %{len}% );
}
```

Figure 4. Example of a BLE service specification

4. THE ARCHITECTURE OF THE GENERATOR

To guarantee that the generator could be easily adapted to produce code for other BLE chips/stacks, the authors used the builder design pattern (E. Gamma, 1994) as it is illustrated at Figure 5. This pattern can be used whenever the same building procedure can be applied to build distinct products/outputs. The authors believe that is the case of this generator. This pattern contributes to reinforce the separation between the parsing and the code generation, allowing inclusively to change the concrete builder at generation-time.

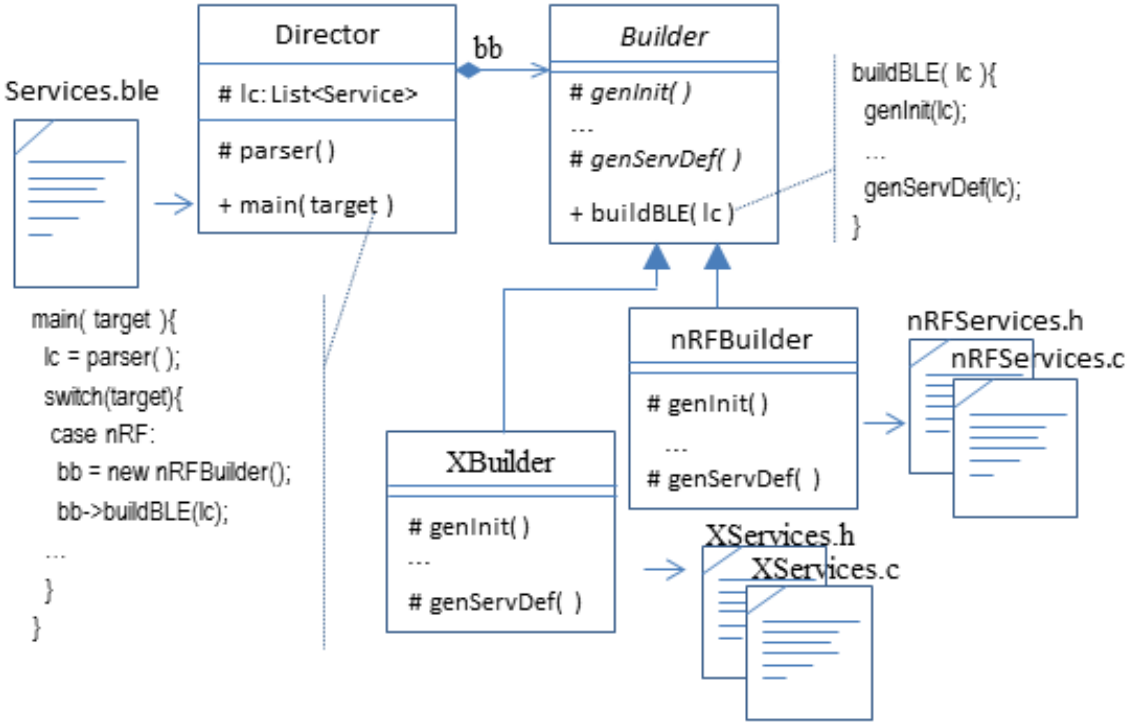


Figure 5. Code generator architecture

The main function, which is a real function and not a method like it is represented at the Figure 5 (the implementation was done using bisonc++ (Brokken, n.d.) and C++), calls the parser to collect all the necessary information from the specification and fulfill the data structures (represented at the Figure 5 by the *List* of *Service* objects). Afterward, based on the argument *target*, instantiates the concrete builder that is able to generate the code for the desired chip/stack. Then it calls the *buildBLE()* method, of the created *Builder* object, passing the require data structures.

The *Builder* class, which is an abstract class, defines the method *buildBLE()* that drives the building process. It also imposes that the concrete builder classes, like *nRFBuilder* and *XBuilder*, implement the methods required for the building process (represented at the Figure 5 by *genInit()* and *genServDef()*). Each of these methods is responsible for the generation of part of the final code.

5. CONCLUSIONS

The code generator presented at this paper that probably will be named by BLEGen simplifies significantly the task of implement the BLE services, reduces the developing time and the probability of errors, normalizes the generated code and hides lot of details that are not necessary to many developers. As a prototype, it must be submit to more tests, namely developing new concrete builders to see if the method that drives the code generation is enough generic and flexible to cover other BLE chips/stacks. It is also important to implement the missing features, like a solution to define the security settings of the services or the possibility to associate descriptors to the characteristics. It would be nice to supply more evolved models of iteration between peripheral and central units (client

and server). For example, the BLE is based on the client-server architectural pattern, but does not allow to directly pass parameters to the server, the requests are based only in the server id. The implementation of a solution that simulates requests with parameters is possible but requires more than one characteristics and a small protocol. The proposed generator can easily supply this kind of solution, hiding all the complexity and unnecessary details.

REFERENCES

A. Huang, L.R., 2007. *Bluetooth Essentials for Programmers*. Cambridge University Press.

Anon., n.d. *Bluetooth low energy Stack*. [Online] Available at: https://developer.bluetooth.org/TechnologyOverview/Pages/BLE.aspx [Accessed 2014].

Brokken, F., n.d. *Bisonc++ V 4.08.00 User Guide*. [Online] Available at: http://bisoncpp.sourceforge.net/bisonc++.html [Accessed 2014].

Decuir, J., 2010. *Changing the way the world connects - Bluetooth 4.0: Low Energy*. CSR.

Donovan, J., 2014. *Bluetooth Low-Energy: An Introduction*. [Online] Available at: http://low-powerwireless.com/blog/2010/07/08/bluetooth-low-energy-an-introduction/.

E. Gamma, R.H.R.J.J.V., 1994. *Design Patterns – Elements of reusable object-oriented software*. Addison-Wesley.

J. Bradley, J.B.D.H., 2013. Embracing the internet of everything to capture your share of the $14.4 trillion - White Paper., 2013. Cisco.

Lee, N., 2011. *Cnet*. [Online] Available at: http://www.cnet.com/news/bluetooth-4-0-what-is-it-and-does-it-matter/.

Nordic Semiconductor, 2014. *Creating Bluetooth® Low Energy Applications Using nRF51822*. [Online] Available at: http://www.nordicsemi.com/eng/nordic/download_resource/24020/3/37348252.

nRF51 SDK, 2014. *Introduction to the S110 SoftDevice*. [Online] Available at: https://devzone.nordicsemi.com/documentation/nrf51/4.1.0/html/group__nrf518__lib__ble__s110__intro.html.