BLEGen — A Code Generator for Bluetooth Low Energy Services

P. Oliveira and P. J. Matos

***Abstract*—The BLEGen is a retargeting generation code tool to implement Bluetooth Low Energy services. Based on a specification of the main characteristics of the BLE services, the BLEGen is able to produce all the necessary C code to implement the services for the chosen microchip. The specifications are written using a domain specific language, which was designed to allow a very compact, easy to understand, highly focused on the BLE standard concepts and fully independent of the details and constraints of the target microchip. The BLEGen was conceived based on the builder design pattern to be a retargeting tool. This means that it is able to generate code for distinct target microchips, using the correspondent BLE software stack and taking advantage of the concrete microchip architecture. The BLEGen allows the implementation of new BLE services, reducing significantly the development time and the level of expertise, since it avoids a long learning period to understand the BLE stack used by each manufacturer and the microchip architecture.**

At this paper, it is presented the motivation, the domain specific language defined to describe the BLE services, the architecture of the BLEGen and the solutions adopted to make it a retargeting tool.

***Index Terms*—Bluetooth low energy, builder pattern, code generation.**

I. 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 [1].

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 [2] of net profit for the companies positioned in these technological areas. The Bluetooth Low Energy (BLE) [3] also known by Bluetooth Smart, is part of this technological revolution.

TABLE I: BLE Market Values [2]

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

The BLE is particularly useful and used on mobile devices, like smartphones and tablets, where the expertise of the software developers is focused on the application level (Android/Java, iOS/Objective-C, Windows Phone/C#, ...). By this point of view, dealing with the BLE is quite similar to many other event oriented solutions, is not very complex and it does not require special skills.

But the expertise to develop mobile software applications is only part of the knowledge required to implement a full solution [1]. Besides the software for the mobile device, the full solution can require the external device, the one that it is supposed to supply data to the mobile device or to be controlled by this one.

The BLE is based on a service-oriented architecture, where one of the devices assumes the role of server (also designated by *peripheral*) and the other assumes the role of slave (also designated by *central*). The server is the device that contains the data and the slave is the one that requests data to the server. The smartphones typically play the central role, monitoring and controlling one or more peripheral devices. The peripheral devices are typically programmed using low-level languages, like C and even assembly, where a strong knowledge about the hardware architecture and developing tools is fundamental. It is the kind of know-how that is very specific, very low level, typically constrained by the hardware resources, like timers, memory, real-time clocks, interrupts and many others things, and normally it is not very accessible for most mobile application developers.

The BLE standard includes a set of profiles for the peripheral devices, each one defining a service. The goal was to promote a standardize 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. The profiles defined on the BLE standard and supplied as implemented services by the hardware manufacturers help on

Manuscript received August 10, 2014; revised November 10, 2014.

The authors are with the Polytechnical Institute of Bragança at Portugal, Portugal (e-mail: poliveira@ipb.pt, pmatos@ipb.pt).

adaptation of the BLE technology, minimizing the required knowledge and sometimes avoiding the development of this component. As consequence, the market was quickly 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 microchip characteristics and resources; the BLE specification is not very accessible and uses a wide range of technologies; the available implementations of the BLE stacks 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 constraints and difficulties common to this level. Our experience shows us that to achieve a satisfactory level of expertise, in order to produce functional and consistent solutions, it can take several weeks or even months. Specially because, most microchips are very recent and, as consequence, there is a natural lack of documentation and technical support.

Confronted with all these problems, we implemented a code generator, the BLEGen to assist on the development of BLE services. Presently, the generator, which is a prototype, produces C code for only one family of microchips, the nRF51 - one of the most used BLE microchips on 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 microchips, is possible and not difficult to do.

The work developed around the BLGen aims to supply a tool to support the academic, research and industrial community on the development of new BLE services, without have to spend too much time learning the BLE implementation details, the hardware architectures specifications and how to use the developing tools. Based on a domain specific language, that requires a very short and higher-level description of the BLE services. This generator can produce highly normalized and optimized C code.

The BLEGen, that is currently a prototype, produces all the necessary code to implement BLE services for the SoftDevices software stack [4], [5], used on the family of chips nRF51 [6]. But it was conceived based on the builder design pattern to enforce the separation of the parsing process from the generation process. This solution, as it is explained in Section IV, has several advantages, namely, it makes more accessible and easier the maintenance of the BLEGen to generate BLE services for new microchips.

The BLEGen is a domain specific tool and it is the only tool known by the authors, available to support the development of BLE services.

This paper is organized as follows: Section II introduces the BLE, namely the GATT level that supports the services; Section III introduces the syntax defined for the code generator; Section IV explains the architectural design of the generator; and Section V wraps up the paper with the conclusions.

II. Bluetooth Low Energy

BLE technology is an extension of the Bluetooth 4.0 standard, introduced by SIG in late 2009 and optimized specifically for devices that use small batteries and require very low consumption [7]. Devices that support Bluetooth Low Energy are called Bluetooth Smart Devices and certified by the Special Interest Group (SIG). Operate in the same ISM (Industrial, scientific and medical) 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 [8]. 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 of consumption 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 lower data transmission rate (0.26 Mbit/s) and the radius of coverage is limited to 50 meters, half the maximum allowed by earlier versions [3]. Depending on usage, BLE can consume up to 100 times less than previous versions and, even in less favorable scenarios, the consumption is reduced at least to half. 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 [9]. Being a standard accepted by most manufacturers, including the main smartphones manufacturers, assures high levels of integration and interoperability.

A. Pairing, Connection and Transmission

The type of communication is master-slave, the master is 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 the slave devices that are on, the discovery mode will receive the messages, afterward they will reply, sending their address and class. A connection request can exist depending on the desired [10].

B. The BLE Protocol Stack

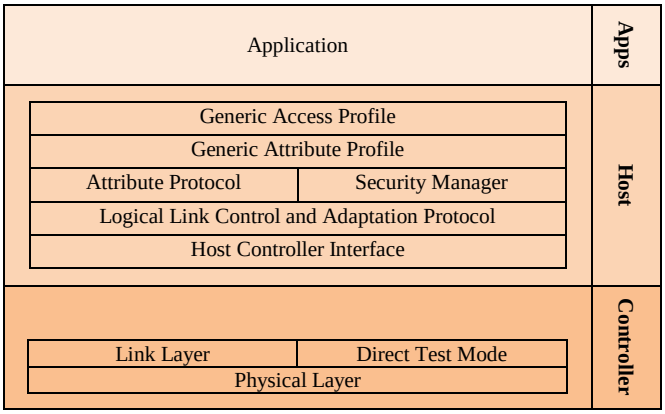
The BLE protocol stack is partitioned into controller and host. The controller is responsible for managing the lower layers of the stack, particularly the capture of physical packages and control of the radio frequency circuit. The host components are: logic control layer connection and adaptation (L2CAP), 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 Table II.

In this paper, we only address in detail the GATT layer, because it is the only one relevant for the purposes of this paper.

C. GATT

The Generic Attribute Profile (GATT), which is responsible for describing the different frameworks of services and is an extension of Attribute protocol (ATT) that is specific by 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.

TABLE II: BLUETOOTH LOW ENERGY STACK ARCHITECTURE



A GATT service 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 III.

TABLE III: 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 contain 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. An example of a complete GATT structure is show in the Table IV.

TABLE IV: GATT STRUCTURE

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

III. THE LANGUAGE SPECIFICATION

Before introduce 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 such 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 of each service. These two elements are specified as it is illustrated in Fig. 1, where WGEN is the name of the service and 0x2D26000057377FEE961BA8DB441BC2AC its address.

Syntax:

```
BLESERVICE ( prefix, address){
  predef
  dis_connection
  characteristics
}
```

Example:

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

Fig. 1. Service definition.

The *predef* is a block of C code surrounded by %{ // C code }%, that will be insert before the code generated for the BLE service. Users can use it to include the code that complements the BLE service, like definition of types (enumerations, structs and others), declarations of variables and constants and, implementation of functions. The service definition also allows to include the 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 in Fig. 2.

Syntax:

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

Example:

```
ONCONNECT:%{counter=0;}%
ONDISCONNECT:%{fclose(fp);}%
```

Fig. 2. OnConnect and OnDisconnect procedures definition.

The service definition can also contain the definition of one or more characteristics. For each one, 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), like it is shown in Fig. 3.

```
Syntax:
characteristic -> prefix ( address,
accesstype, typevalue, minnumber,
maxnumber)%{
// C code
}%
Example:
temperature( 0x20AA, [rn], uint8_t,
1, %{len}%)%{ ... }%
```

Fig. 3. Characteristic definition.

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 structure of the BLE service implemented is show in the Table V.

TABLE V: STRUCTURE BLE SERVICE WGEN

	Handle	UUID	Permissions	Value
Service	0x0001	Service	READ	WGEN
Characteristic	0x20AA	TEMP1	WRITE	20
Characteristic	0x20BB	TEMP2	READ/NOTIFY	30

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 surrounded by %{ //C expression }%, as it is illustrated at Fig. 3.

```
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}% );

  }
```

Fig. 4. Example of a BLE service specification.

At the end of the definition (see Fig. 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*, don t execute any kind of procedure. But once defined one, the user should define all the others, even when they don t use it (using an empty block). Fig. 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.

IV. THE ARCHITECTURE OF THE GENERATOR

To guarantee that the generator could be easily adapted to produce code for other BLE microchips/stacks, the authors used the builder design pattern [11] as it is illustrated at Fig. 5. This pattern is used whenever the same building procedure can be applied to build distinct products/outputs. The authors believe that this is the case of this generator. This pattern contributes to reinforce the separation between the parsing and the code generation, allowing inclusively changing the concrete builder at generation-time.

The implementation was done using bisonc++ [12] and C++. The main function calls the parser to collect all the necessary information from the specification and fulfill the data structures (represented in Fig. 5 by the *List* of *Service* objects). Afterward, based on the argument *target*, instantiates the concrete builder (*nRFBuilder/XBuilder*) that is able to generate the code for the desired microchip/stack. Then, it calls the *buildBLE()* method, of the created *Builder* object, passing the required data structures.

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

V. CONCLUSIONS

The BLEGen presented in this paper 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. But the most important contribution is that allows implementing a BLE service in few hours, without having to be an expert on the hardware architecture or on the implementation of correspondent BLE stack.

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 microchips/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 characteristic, and a small protocol. With small improvements, the proposed generator could easily supply this kind of solution, hiding all the complexity and unnecessary details.

The BLEGen was already used to implement several services, which will be available soon on the Android Market and Apple Store, and the goal is to make the BLEGen available as an open source project.

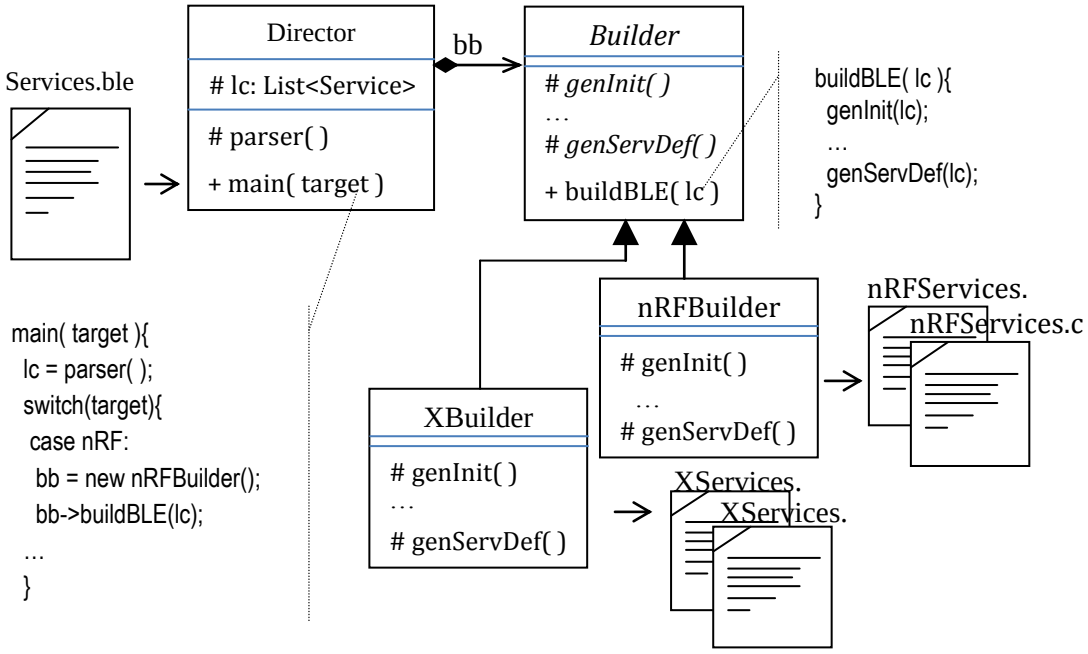


Fig. 5. Code generator architecture.

REFERENCES

[1] P. Oliveira, "Aplicação do Bluetooth Low Energy no controlo e monitorização de dispositivos de muito baixo consumo," Master Thesis, Escola Superior de Tecnologia e de Gestão, Bragança.

[2] J. Decuir, "Changing the way the world connects - Bluetooth 4.0: Low energy," *CSR*, 2010.

[3] *Specification of the Bluetooth System*, Bluetooth SIG, Inc., 2013.

[4] Introduction to S110 SoftDevice, *User Guide*, *Nordic Semi-Conductors*, 2013.

[5] *Creating Bluetooth Low Energy Applications Using nRF51822*, Nordic Semiconductor, 2013.

[6] *nRF51822 Multiprotocol Bluetooth 4.0 low energy/2.4GHz RF SoC - Product Specification2.0", Nordic SemiConductors*, 2013.

[7] Bluetooth Low-Energy: An Introduction. (2014). [Online]. Available: http://low-powerwireless.com/blog/2010/07/08/bluetooth-low-energy-an-introduction/

[8] N. Lee. (2011). Bluetooth 4.0: What Is It, and does It Matter? [Online]. Available: http://www.cnet.com/news/bluetooth-4-0-what-is-it-and-does-it-matter/

[9] J. Bradley, J. Barbier, and D. Handler, *Embracing the Internet of Everything to Capture Your Share of the $14.4 Trillion - White Paper*, Cisco, 2013.

[10] A. Huang and L. Rudolph, *Bluetooth Essentials for Programmers*, Cambridge University Press, 2007.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[12] F. Brokken. (2014). Bisonc++ V 4.08.00 User Guide. [Online]. Available: http://bisoncpp.sourceforge.net/bisonc++.html

P. Oliveira is currently a Ph.D. student in the Faculty of Science at University of Porto, Portugal. He received his CSE in computer engineering from Escola Superior de Tecnologia e Gestão de Bragança, Portugal in 2013 and M.Sc. in information systems from Escola Superior de Tecnologia e Gestão de Bragança, Portugal in 2014. His research mainly focuses on bluetooth low energy, behavior analysis and ambient intelligence.

P. J. Matos is an associate professor at the Department of Informatics and Communications of the Institute Polytechnic of Bragança. He concluded his PhD in 2005, master degree in 1989 and CSE degree in 1994 at University of Minho, Portugal. He is an author of several international refereed papers, participates regularly in international research projects, mainly with industrial partners, and his research interests include decision support systems, data mining, web semantic, collective intelligence and intelligent environments.