Beacons for smartBASIC – BL600 and BT900

Walkthrough *v1.0*

INTRODUCTION

The goal of this document includes the following:

1. Introduce the basics of Bluetooth Low Energy
2. Demonstrate the use of *smart*BASIC to create BLE applications
3. Walk through creating an Eddystone-URL beacon as a demonstration

OVERVIEW

Bluetooth and specifically Bluetooth Low Energy (BLE) are defined in large and comprehensive specifications by the Bluetooth Special Interest Group. However, these specifications can be difficult to navigate when learning to develop Bluetooth products. This guide explains the basics of BLE and shows how to use *smart*BASIC to create a BLE application for Laird's interactive programmable modules.

To accomplish this, this guide walks through the steps to create an Eddystone-URL beacon. The Eddystone specification is defined by Google and can be tested against existing apps on BLE-enabled Android and iOS devices.

REQUIREMENTS

The following DVKs can be used with this guide:

- BT900 – DVK or USB dongle
- BL600 – DVK or USB dongle

Note: It is recommended that you update the firmware on your DVK to the latest version available from the respective product page on http://www.lairdtech.com/solutions/embedded-wireless.

The following is also recommended:

- UwTerminal – Available as a fee download from Laird
- Eddyurl.sb – The completed *smart*BASIC script created in this guide (to be used as a reference). This is found in Appendix I: Completed Eddystone Script.

This guide assumes that you are familiar with loading *smart*BASIC scripts onto your DVK or USB dongle. If not, see the UwTerminal Application Note.

BLUETOOTH LOW ENERGY (BLE)

Bluetooth Low Energy, also known as BLE or Bluetooth Smart, was introduced into the Bluetooth 4.0 core specification. BLE is a protocol that was originally designed by Nokia under the name "Wibree" where many of the protocol and physical transport layers were designed to use the lowest power possible until it was later adopted into the Bluetooth specification. This was originally intended to address a gap in the Bluetooth design to allow small devices that could run for months or years on a single battery.

BLE shares some of the lower-level aspects with Classic Bluetooth, but is overall a completely separate design that can either be used alongside Classic Bluetooth (dual mode), or in a single low power device without Classic Bluetooth.

One of the largest power savings comes from the change in topology between Classic Bluetooth and BLE. In Classic, a slave device must be constantly listening to receive incoming connections and therefore must have its radio hardware powered on. In BLE, the slave will intermittently transmit data over the air for a device to receive and can power down the radio hardware when not transmitting. With this change in behavior, the advertising device can massively reduce its average power consumption.

In this guide we will also look into connectionless advertising with the Generic Access Profile and how to implement such a case in *smart*BASIC.

*smart*BASIC

This guide assumes that you understand the fundamentals of *smart*BASIC and have already compiled and executed a "Hello World" script, or otherwise understand how to use and create functions and how to handle events/messages.

To keep this example brief, error checking has been omitted. While you should not encounter any errors in such a small example, it is highly recommended that in production applications you ensure that functions return a "Success" return code before continuing to the next step.

A complete example of this Eddystone beacon is supplied with this guide, but it is recommended that you write this script along with this guide to get a better understanding of the construction of a beacon.

Adverts

Advertising is one of the simplest data transfer use cases of BLE and is used when a device wants to repeatedly transmit a small amount of data (an advert can contain up to 31 bytes). One of the most common uses for advertising you may have heard of is acting as a beacon.

A beacon is a peripheral that can be placed in a location of interest and broadcasts adverts repeatedly at a set interval. A BLE scanner, such as a smart phone, can listen for these adverts and notify the end user or pass the advert data to an app.

Adverts are small blocks of bytes, which are broadcast over the air and are made up of one or more advert data elements. For example the location in a building, a URL, the current temperature, or special offers as a customer walks past a product on-shelf. Each individual advert data element begins with a single length byte, then a single tag byte, and then the payload.

Registered tag values are maintained by the Bluetooth SIG.

Advertisements are implemented using GAP, which stands for Generic Access Profile. GAP is a profile within Bluetooth and is used to manage advertising and connections in BLE. Although connections are a part of the generic attribute profile (GATT), they will not be explained in this guide.

There are two roles defined in GAP that are used in a simple connectionless advertising setup:

1. A "peripheral" device is a small, low-power device such as a beacon. It features limited capabilities in order to keep power usage low and connect to more powerful "central" devices. They can often run for months or years on a single battery depending on the configuration.
2. A "central" device is a more powerful device that is used to communicate with "peripheral" devices. A central device is usually a smart phones or a personal computer. The higher processing power means that a larger share of the complexity of BLE is handled by this central device. A central device will listen for peripherals that are advertising and inform the user or call an app when a peripheral is detected within range.

BLE devices available from Laird include:

- BL600: A programmable single mode BLE peripheral module
- BL620: A programmable single mode BLE central module
- BT900: A programmable dual mode module that supports central and peripheral BLE modes and Classic Bluetooth

An advert is made up of a number of Advertising Data (AD) elements, which are contained within an array of 31 bytes within the advert. These AD elements are key-value pairs, where the key describes the type of data that is paired with it.

A number of keys have been defined by the Bluetooth SIG and are used by the central device to parse the data within the 31 byte payload (available here). For example a key 0x10 defines that the paired value contains the Device ID, and a key 0x09 is followed by the name of the device.

If a central device requires more information from a peripheral it a sends a "Scan Request" to the peripheral which requests another 31 byte payload. This increases the amount of data that can be advertised.

Each advert is then repeatedly transmitted at a specified "advertising interval". This is a period of time that is selected by the developer to handle the balance between responsiveness and power consumption.

For example, if an interval of 50ms is selected, the frame will be advertised every 50ms and can be quickly received by a user's mobile device. This may be useful if a user is moving at a high speed past a peripheral, or if low latency is required. This increases the power consumption though, as a peripheral will often go into low power sleep mode between adverts, spending less time sleeping compared to the amount of time advertising.

Selecting a long advertising interval, such as one second, will greatly reduce the power consumption in situations where low latency is acceptable. One example would be advertising exhibits as a user is slowly walking around a museum.

Scan Requests/Responses

In situations where a chunk of 31 bytes is not large enough to contain all advertising data, or a central device wants further information about an advert it has received, a scan request can be performed.

A scan request is sent by a central device directly to a peripheral and requests that the peripheral sends a scan response containing another 31 bytes of data. This scan response is formatted in the same way as an advert and consists of a structure of AD elements and can contain any data type.

For example, a peripheral's scan response may include a string containing the device name that is not required in the advert.

A central device can also integrate a scan request while it is listening for adverts by performing active scanning. In this mode, when the central device receives an advert, it will automatically submit a scan request to the advertising peripheral. Listening for adverts without an additional scan request is known as passive scanning.

The majority of mobile devices perform active scanning, so it can often be useful to populate the scan response with a device name that can be displayed to the user on their device.

Scan won't be used in this example, so we will define our scan response in *smart*BASIC as an empty string.

About Eddystone

In this section, we'll create an Eddystone beacon with *smart*BASIC. For convenience, the overview of iBeacons is also provided in .

We will be defining a beacon device using the Eddystone-URL open standard. Eddystone is a standard defined by Google and is supplied free of royalties for developers to implement into their beacons.

Eddystone has growing support and is currently supported by iOS and Android devices with a number of companion apps such as Google Chrome or Physical Web.

The URL subset of the specification allows an encoded URL to be advertised and for users to be notified by their mobile device when within range of the beacon.

We will be using this URL specification as an example. Although it defines more than may be included in a custom advert, the resulting beacon will be compatible with a number of existing freely available apps. Therefore, no custom Android/iOS apps will need to be written for the purposes of this guide.

The Eddystone specification is open source and obtainable from GitHub, and the subset we will be using, Eddystone-URL, is available within the same project. Eddystone defines the format of the advert and URL defines the format of the AD element.

This Eddystone script we're creating here is a simple example of a beacon that can be used free of licensing and can be integrated into your own beacon enabled products.

Note: The Eddystone specification includes a couple of parameters in an advert that are used to increase compatibility with central devices but are not always required in a generic BLE advert. These parameters will be indicated throughout the guide and should be considered optional if implementing your own specification.

CREATING AN EDDYSTONE BEACON IN *SMART*BASIC

To begin your *smart*BASIC script, open a new text document and save it as “eddyurl.sb”. It is recommended that you write your scripts in the Notepad++ editor as Laird provides language definition files for Notepad++ that allows syntax highlighting for ease of coding.

Initial Defines

The first thing to do that should help with development of this script is to define a few parameters in accordance with the Eddystone specification. This is not yet specific to BLE but defines values used later in the script.

Enter the following into your empty script. If you have already read the Eddystone specification then you may already be aware of what these values represent. They will be explained later in this document.

```
#define ES_UUID   0xFEAA
#define ES_URL    0x0010
```

Initial Global Variables

Underneath that, declare the following global variables:

```
DIM rc            // Integer that stores return code from functions

DIM URL$ : URL$ = “lairdtech”// Defines the URL the be advertised

DIM advData$       // String used in construction of a BLE advert
DIM advRpt$        // String that holds the advert report when fully constructed
DIM scanRpt$       // String that defines the Scan response. As Scan is unused this
                  will remain empty
```

About Services and UUIDs

Services and UUIDs are not explained in this guide, but a brief definition will help to explain the following section and why they are used.

A connection can be made in BLE between a peripheral and central device, which allows data to be sent bi-directionally and securely. During transmission, related data values can be grouped into “services” which simply defines a set of one or more values and their types contained within the service.

For example, the heart rate “service” groups multiple values such as heart rate measurement and sensor body location.

A UUID is a 128-bit tag which is specified by a manufacturer to define the type of service being sent. This allows the remote device to parse the service it has received according to the specification of the UUID. As the number of possible values available in a 128-bit UUID, the Bluetooth SIG does not require a vendor to register their UUIDs to avoid clashes.

A vendor may also pay the Bluetooth SIG to register a 16 bit UUID and ensure that it never clashes with other products. For example, the 16-bit UUID of an Eddystone service is 0xFEAA and this will be used in the following script.

Creating an Advert Record

To construct a BLE advert we must first define the AD element that is contained within the advert.

Our URL and the accompanying Eddystone tags will be defined under the "Service Data" AD data type as specified in Bluetooth GAP Assigned Numbers found here.

This data type is used to supply arbitrary data to a BLE service. Although we will not be using any services, this tag allows us to include arbitrary data inside an advert.

1.11.2 Format

Data Type	Description
<<Service Data>>	Size: 2 or more octets The first 2 octets contain the 16 bit Service UUID followed by additional service data

Figure 1: Bluetooth 4.0 core specification Service Data

Figure 1 is from the Bluetooth 4.0 specification and defines that the first 2 bytes of a "Service Data" AD type must be the associated 16-bit UUID. As we have already defined the UUID in the Initial Defines of the *smart*BASIC script, it can be added to the currently empty AD.

The function BleEncode16 is used to encode a 16-bit integer value into an AD at a specified offset. The offset is defined as a multiple of 1 byte. As this is the first record, the offset is 0.

```
// Create the advert record

// Commit the Eddystone UUID to the advert at offset 0
rc = BleEncode16(advData$, ES_UUID, 0)
```

The remainder of the AD element is constructed in accordance with the Eddystone specification, another standard specification or custom specification.

In this case we will be populating the remainder of the "Service Data" according to the following Eddystone-URL Frame specification.

> **Note**: The following is required by the "Eddystone-URL Frame" specification and is not required in all adverts.

As we have already defined a 16 bit UUID as our first tag, our byte offset will be the values shown in Table 1.

Table 1: Frame Specification

Byte Offset	Field	Description
0	Frame Type	Value = `0x10`
1	TX Power	Calibrated Tx power at 0 m
2	URL Scheme	Encoded Scheme Prefix
3+	Encoded URL	Length 0-17

```
rc = BleEncode8(advData$, ES_URL, 2)    // Eddystone-URL Frame type 0x10
rc = BleEncode8(advData$,4, 3)          // TX Power hardcoded to 4 for demo purposes
```

The URL is defined in a shortened form to allow more characters to fit in a report using single non-ascii bytes to represent common prefixes such as “http://www.”

We will hardcode this to the following values for this example, but please refer to the Eddystone-URL specification for your use case.

- 0x00 = “http://www.”
- 0x00 = “.com”

We can now commit our URL to the AD.

```
rc = BleEncode8(advData$, 0x00, 4)                          // Prefix:  http://www.
rc = BleEncodeString(advData$, 5, URL$, 0, strlen(URL$))    // URL
rc = BleEncode8(advData$, 0x00, strlen(URL$) + 5)           // Suffix:  .com
```

The advData$ string has now been populated with the advert record data.

Creating an Advert Report

An AD element is contained inside an advert report and so the AD element must be committed into an advert report string. An advert report string has not yet been initialized, so that is the next step.

As this device will not be connectable, the default values of 0 are valid, but refer to the manual for your own applications.

Although the scan report is not being used in this example, an empty report must still be initialized.

```
rc = BleAdvRptInit(advRpt$, 0, 0, 0)     // Not connectable
rc = BleScanRptInit(scanRpt$)            // Init an empty string as the scan report
```

Before committing the AD element to the advert report, there is another Eddystone-specific parameter to add. This section may not be required when implementing another beacon specification.

In the Eddystone specification (not the URL specification), it is defined that the first parameter in the advert report must be a “Complete list of 16-bit Service UUIDs”.

The *smart*BASIC function, BleAdvRptAddUuid16 exists to add a list of up to 6 UUIDs to a report. If fewer than 6 UUIDs are being declared, then the value -1 is passed for unused UUIDs.

```
// Add service UUID to advert report
rc = BleAdvRptAddUuid16(advRpt$, ES_UUID, -1, -1, -1, -1, -1
```

We can now commit the record to the advert report. This will append the AD using the “Service Data” data type 0x16 as defined in the “Assigned Numbers” document and will calculate and update the length field in the advert.

Following this, the advert and scan report have been successfully constructed.

It is also recommended at this point to check the return value of the BleAdvRptAppendAD function, since if the string length exceeds the length of the advert, an error is returned.

```
// Append the record to the report using the Service Data type 0x16
rc = BleAdvRptAppendAD(advRpt$, 0x16, advData$)
```

PDU (Packet Data Unit)

A BLE peripheral includes a PDU (Packet Data Unit) in adverts that it broadcasts. This Advertising Channel PDU includes parameters that determine the capabilities of the device and whether a central device can scan or connect to the device.

The most common PDU types are ADV_IND and ADV_NONCONN_IND.

ADV_IND is defined as a "connectable undirected advertising event". This declares that the advertising device can be connected to and that it is undirected, i.e., not directed a particular central device (or broadcasting).

ADV_NONCONN_IND is defined as a "non-connectable undirected advertising event" and is the type that is commonly used in beacons. This defines that the device will not accept connections or scan requests and that the broadcast is undirected.

Beginning BLE Advertising

Now that a report has been defined, it can be committed to BLE baseband of the module. Following the commit, the module can then use that committed advert to begin advertising.

As described earlier, the interval defines how often the advert is broadcast. In this example the interval is 200mS. After you have completed this guide, feel free to change this interval value and observe how it affects the responsiveness of the mobile app. This is especially important when a mobile device is on the edge of the broadcast range as it may receive only a limited number of adverts.

The advert type is passed as a parameter to the BleAdvertStart function in *smart*BASIC, along with a string containing the address of the device the advert is directed towards. As the advert is undirected, this string will be empty.

```
// Commit the constructed reports to the BLE subsystem
rc = BleAdvRptsCommit(advRpt$, scanRpt$)

DIM addr$ : addr$ = ""       // Define an empty string for undirected advertising

// Begin advertising.
rc = BleAdvertStart(3, addr$, 200, 0, 0)       // 3=ADV_NONCONN_IND, 200mS
interval, 0 timeout
```

Arguments		
nAdvType	byVal *nAdvType* AS INTEGER. Specifies the advertisement type as follows:	
	0	ADV_IND – Invites connection requests
	1	ADV_DIRECT_HIGH_DUTY_CYCLE – Invites connection from addressed device using high duty cycle timing. nAdvInternal and nAdvTimeout are ignored and interval is set to 3.75ms and Timeout to 1.28 seconds as per the specification. See ADV_DIRECT_LOW_DUTY_CYCLE for an alternative.
	2	ADV_SCAN – Invites scan requests for more advert data
	3	ADV_NONCONN – Does not accept connections and/or active scans
	4	ADV_DIRECT_LOW_DUTY_CYCLE – Invites connection from addressed device using low duty cycle timing using nAdvInternal and nAdvTimeout specified

Figure 2: BLE Advertising Types (type 3 is used above)

Waitevent

Finally, we add a waitevent to the script. Although we have not registered any *smart*BASIC event handlers, adding this will ensure that the script does not exit and keeps running indefinitely.

```
waitevent
```

Testing Your Beacon

The script should now be ready to run on your Laird module. Compile and load the *smart*BASIC script onto the module and execute it. It is recommended that this be carried out via the UwTerminal program supplied by Laird (see the UwTerminal Application Note). This allows the process to be automated via the "right click menu/XCompile Load and Run".

The module should then begin advertising.

There are a number of apps that are available to test your Eddystone beacon on both Android and iOS. The app "Physical Web" is an open source app that has been developed by Google and is available on both Android and iOS and available from the Play Store and App Store. In this example, we'll use Physical Web to interact with the beacon.

Upon installing and running "Physical Web", your device will search for beacons and display them. You should see your URL listed within the app.

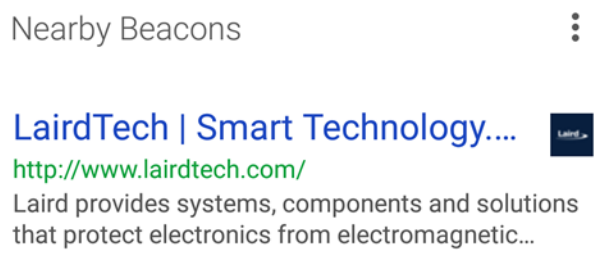

Figure 3: Laird beacon in Physical Web app.

If you are having issues with your beacon, other apps can be used that provide a more verbose interface that may be useful in debugging. One such app is "nRF Master Control Panel" from Nordic. If you are still experiencing issues, make sure to check the return codes of the functions in the scripts. Any unexpected return codes are documented in the *smart*BASIC manuals and may help diagnose the issue.

APPENDIX I: COMPLETED EDDYSTONE SCRIPT

For reference, the script we completed in this guide is shown in its entirety below.

```
#define ES_UUID      0xFEAA
#define ES_URL       0x0010


DIM rc               // Integer that stores return code from functions


DIM URL$ : URL$ = "lairdtech"              // Defines the URL the be advertised


DIM advData$         // String used in construction of a BLE advert
DIM advRpt$          // String that holds the advert report when fully constructed
DIM scanRpt$         // String that defines the Scan response. As Scan is unused this
                     will remain empty


// Create the advert record


// Commit the Eddystone UUID to the advert at offset 0
rc = BleEncode16(advData$, ES_UUID, 0)

rc = BleEncode8(advData$, ES_URL, 2)    // Eddystone-URL Frame type 0x10
rc = BleEncode8(advData$,4, 3)          // TX Power hardcoded to 4 for demo purposes

rc = BleEncode8(advData$, 0x00, 4)                          // Prefix:  http://www.
rc = BleEncodeString(advData$, 5, URL$, 0, strlen(URL$))    // URL
rc = BleEncode8(advData$, 0x00, strlen(URL$) + 5)           // Suffix:  .com

rc = BleAdvRptInit(advRpt$, 0, 0, 0)     // Not connectable
rc = BleScanRptInit(scanRpt$)            // Init an empty string as the scan report


// Add service UUID to advert report
rc = BleAdvRptAddUuid16(advRpt$, ES_UUID, -1, -1, -1, -1, -1


// Append the record to the report using the Service Data type 0x16
rc = BleAdvRptAppendAD(advRpt$, 0x16, advData$)


// Commit the constructed reports to the BLE subsystem
rc = BleAdvRptsCommit(advRpt$, scanRpt$)

DIM addr$ : addr$ = ""                        // Define an empty string for undirected
                                              advertising

// Begin advertising.
rc = BleAdvertStart(3, addr$, 200, 0, 0)     // 3=ADV_NONCONN_IND, 200mS interval, 0 timeout


waitevent
```

APPENDIX II: IBEACON OVERVIEW

An iBeacon consists of a BLE-enabled device that sends out an Apple, Inc. Manufacturer Specific Advertising Record, which will be henceforth referred to in this document as an AMSAD, formatted specifically to be interpreted by a receiving BLE device.

The Bluetooth specification requires that the AMSAD consists of 4 or more octets. The first octet is the overall length, the second octet shall always be 0xFF, the third and fourth octets are the manufacturer assigned Company ID for Apple Inc, and the rest is the iBeacon data. Apple has specified the rest of the octets to be grouped as: [Datatype] [Length] [UUID] [Major] [Minor] [Calibration Power], where the UUID is 16 octets long, Major and Minor are 2 octets long and the rest all single octets as shown in Figure 1:

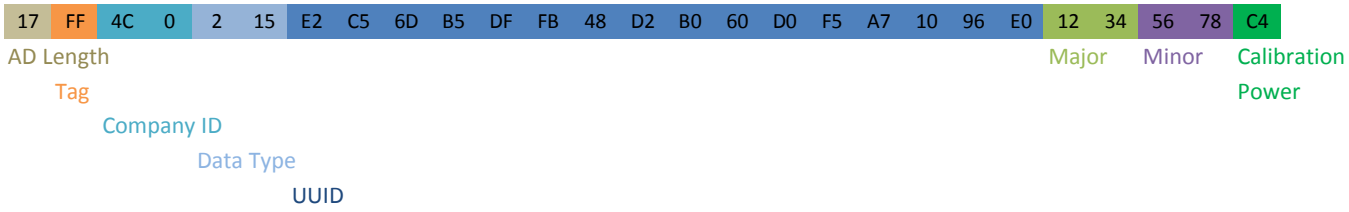

Figure 1: AMSAD Graphical Representation

The iBeacon smartBASIC sample app allows you to remotely configure the following parameters:

- UUID
- Major
- Minor
- Calibration Power
- Advertising Interval
- Advertising Timeout
- Time to remain connectable
- Actual Transmit Power Value

When not in a connection, the BL600 advertises with the AMSAD.

For a detailed walkthrough of setting up and running an iBeacon with the iBeacon sample app, see the BL600 iBeacon Sample App Application Note.

REVISION HISTORY

Version	Date	Notes	Approver
1.0	4 Dec 2015	Initial Release	Chris Ruppersburg