DTAG100 User Reference Guide

Firmware Version V2.4.x

Introduction

The DTAG100 is an intelligent USB-connected device that can act as a variable NFC tag and Bluetooth (BLE) beacon. Depending on the model purchased, you can have one or both of these proximity technologies installed as interface modules, connected to the DTAG100 base unit.

The DTAG100 enables any connected host system to change the NFC tag and/or BLE beacon data at any time via USB, using a simple platform-independent method that doesn't require complex coding or APIs.

It provides a quick and easy way to add mobile interactivity to digital signage players, kiosks, ticketing and point-of-sale systems, without having to worry about the complexities of interoperating across multiple technologies, operating systems or mobile device types.

The DTAG100 can also be configured in advance, using a PC, and then left to run in stand-alone mode for as long as needed, by simply attaching it to any standard USB power supply.

It can also be securely upgraded in the field, making it a flexible and future-proof component of any system requiring mobile interactivity.

Using this guide

This document is aimed at first-time users of the DTAG100 as well as mobile app and system developers, and contains the information you need to get started with both the NFC and BLE functions. The two technologies are covered in separate sections, although some features and facilities are common to both. You will also find information on the many configuration options available, and on how upgrade your DTAG100 unit with new firmware when available.

Advanced applications, including how to send data in both directions between your host system and mobile apps, are also covered, with source code examples and further support available on request.

Demonstration apps and firmware updates

We have written various demonstration apps illustrating different aspects and use cases of the DTAG100 across both NFC and Bluetooth beacons – please visit www.dtag100.com/demo for more information and downloads.

Please also ensure that you have downloaded and installed the latest firmware from this location.

Getting technical support

For further information and technical support please contact us:

Email: support@dotorigin.com

UK telephone: +44 (0)1428 685 861

US telephone: +1-888-262-9642

How it works

When connected to a PC or another host computer system, the DTAG100 appears as a generic mass storage device (ie like a memory stick) and, optionally, as a keyboard input or virtual serial COM port device.

In order to configure your DTAG100, you simply edit or create certain named text files, and the on-board processor will automatically read these files and adjust the NFC tag, BLE beacon and/or BLE attribute data accordingly.

Certain other files are generated automatically by the DTAG100, for instance to provide you with status and event log information that can be read as and when needed by the attached host.

This approach allows you to easily define the data that is transmitted to mobile phones, both manually and programmatically, using a platform-independent approach. It also provides several options for phones to send data back to software running on the host computer, for example by the DTAG100 emulating a barcode scanner.

The DTAG100 will automatically detect whether it has NFC and/or BLE interface modules attached, and will configure itself accordingly. It is also fully upgradable in the field, using a similar file-based method for distributing firmware updates in a secure manner.

Essential NFC operation

When fitted with its NFC interface module, the DTAG100 appears as a standard NFC Forum tag, which can be read by any phone or other device capable of reading NFC tags. It does not act as a reader, and therefore doesn't use tag emulation or peer-to-peer modes, which can be unreliable. Instead it relies on a special dynamic NFC tag chip that implements the same short range radio interface as a standard tag, but allows the DTAG100 processor to provide the actual tag data 'on demand' rather than having it permanently programmed into memory.

The actual contents of the tag cannot be written or changed through the RF interface, but can be changed via the USB interface at any time. The DTAG100 supports all standard NDEF tag formats, as well as custom binary formats, extra-long tags and other advanced features including two-way data transfer.

Most device configuration and NFC-specific functionality is provided by reading and writing certain named text files on the DTAG100's mass storage – primarily '**config.txt**' and '**tagdata.txt**'. These files can be changed manually using a text editor, or programmatically by software running on the host.

During normal operation, the four LEDs on the front of the DTAG100 will flash in sequence. They will all light up for a few seconds when the RF field from a phone or another NFC reader is detected, after which the stored tag data will usually be read and the relevant action taken.

If you have just received an NFC-equipped DTAG100, try putting your mobile phone over the central antenna area to see what happens. As long as your mobile has NFC turned on, you should see a browser window open showing the Dot Origin web site, because of the default tag data stored on the DTAG100 when it is shipped.

Editing tagdata.txt

The easiest way to change the NFC tag data that the DTAG100 broadcasts is to plug the DTAG100 into your computer, browse to the DTAG100 mass storage drive, and open the **tagdata.txt** file in a text editor (eg Windows Notepad). For a standard NFC smart poster, you'll want to provide a URI (a web address for example) and a short bit of text describing the URI. You can include additional pieces of data in the NFC message by adding up to two external type records. External type records have a type and payload, both of which must be provided. Here is an example **tagdata.txt** that includes all three records.

```
!DTAG:MyTagName
SmartPoster
Ext1Type=appdomain.com:theapp
Ext1Payload=apppayload
URI=http://www.dotorigin.com
Text=This is Dot Origin
Ext2Type=android.com:pkg
Ext2Payload=com.yourdomain.yourapp
```

When using SmartPoster mode, the three records of Ext1, URI/Text, Ext2 are all optional and will always be portrayed in that order in the NDEF message that the DTAG100 broadcasts. You must also start your tagdata.txt file with “!DTAG:” followed by a tag name, which acts as an identifier. On the second line, you must have “SmartPoster”. Lines following SmartPoster must include between one and three of the records described.

So, for example, a simple smart poster tag can be created as follows:

```
!DTAG:MyTagName
SmartPoster
URI=http://www.dotorigin.com
Text=This is Dot Origin
```

Launching apps directly on Android

You can use what are called Android Application Records to directly open any app of your choosing. If the app is not installed on the Android device, it will open its listing page in the Google Play Store so that the user can download it easily. You should put this record as the last record in your tagdata.txt file so that it does not interfere with tag reading on other operating systems. The example tagdata.txt below will open the YouTube app directly on Android devices and go to www.youtube.com on other devices:

```
!DTAG:MyTagName
SmartPoster
URI=http://www.youtube.com
Text=This is YouTube
Ext2Type=android.com:pkg
Ext2Payload=com.google.android.youtube
```

The type should always be android.com:pkg. The payload is the app’s package name. The easiest way to find the package name is to view the app’s listing on the Play Store in your browser. It is the last part of the URL. For example, this is YouTube’s app listing:

https://play.google.com/store/apps/details?id=***com.google.android.youtube***

Other types of NFC tag records

An NFC ‘SmartPoster’ tag does not need to contain a web address as the URI – many other types of URI are possible, although some are dependent on the mobile operating system supporting them. You can use the DTAG100 to experiment with the various different direct actions that are possible, such as triggering a phone call, email or SMS text message, as well as opening locations on a map or launching an app-specific URI – see **Appendix B** for more examples.

Our demonstration app for Chrome also has a handy tool for creating different types of NFC tag and sending these directly to the DTAG100 - please visit www.dtag100.com/demo for more information.

Raw tag data mode

The **tagdata.txt** file can hold more than the basic smart poster and external type records described above. If those record types do not fit your specific needs, you can have the DTAG100 broadcast any NFC message you want. To do this, you will need to use raw data mode. You will need to put the entire string of hex-encoded binary data that contains the required NDEF message into **tagdata.txt** (using Type 3/FeliCa tag formatting.) When you write your own hex-encoded data, always precede that data with “raw” on a line of its own. Here is an example file with such a message:

```
!DTAG:MyTagName
raw
94121a646f746f726967696e2e636f6d3a647461677061796d656e743b4d63446f6e616c64
733b33323134352e3030540f1a616e64726f69642e636f6d3a706b67636f6d2e646f746f72
6967696e2e6474616764656d6f72696731
```

Other advanced features

Please refer to later sections of this guide for information on other advanced NFC features of the DTAG100, including logging, two-way data transfer and various additional facilities and configuration settings available.

Essential BLE operation

When fitted with its Bluetooth Low Energy (BLE) interface module, the DTAG100 can interact with mobile phones and other devices using Bluetooth 4.0. These include most modern iPhones and Android phones, which allow installed apps to scan for beacons, display notifications and perform other relevant actions when a beacon is detected nearby.

The DTAG100 can behave as an Apple iBeacon, a Connectable beacon, plus Eddystone and other custom-format beacons. Crucially, in all cases, the transmitted beacon data is changeable via USB at any time, and the unit can even be configured to switch regularly between several different beacons and beacon types at once, without host involvement.

As with the DTAG100 NFC functionality, various advanced features are available, including the ability to transfer additional data to and from the phone.

Most device configuration and BLE-specific functionality is provided by reading and writing certain named text files on the DTAG100's mass storage – primarily '**config.txt**' and '**bledata.txt**'. These files can be changed manually using a text editor, or programmatically by software running on the host.

If you have just received a BLE-equipped DTAG100, try installing our free Bipzone app on your mobile phone. After initialising its database, you should see a notification that a beacon is nearby, because of the default beacon data stored on the DTAG100 when it is shipped.

Using the configuration files, you have full control over how the DTAG100 Bluetooth transmissions are formatted, including the type of beacon, its identity info, what data it shares, and the radio broadcast strength.

Beacon format options

The beacon advertising data that is broadcast at any one time can be in various different industry-standard formats that the DTAG100 provides, including iBeacon, Connectable and Eddystone beacons, plus a raw data mode that allows for custom applications. These are set and configured by modifying the file **bledata.txt** on the device.

A DTAG100 operating in iBeacon mode is fully compliant with the Apple specifications, and you can easily specify the necessary UUID, major and minor values to be recognized by any compatible iOS device and application, and change them dynamically via USB whenever needed. Other mobile devices and operating systems, including Android, can also scan for, and use, iBeacon format transmissions. Note that, in this mode, the DTAG100 does not allow incoming connections from mobile devices.

A DTAG100 Connectable beacon works in the same way as an iBeacon, and is compatible with the same apps and operating systems, except it also allows brief connections from mobile devices, and has an optional local name. This allows for further interactions that we cover later, such as sending and receiving data to and from individual mobile phones.

Eddystone beacons can either transmit a unique ID, for use in a similar way to iBeacons, or they can advertise a URL that mobile apps can see during scanning. You can also choose to advertise Eddystone telemetry data instead, or as well. On the DTAG100, Eddystone beacons are also Connectable, supporting advanced two-way interactions and applications.

Finally, the raw data mode allows your advertising data to be something other than one of the above formats, in which case the DTAG100 allows you to specify all 31 broadcast bytes in hexadecimal.

Please see **Appendix C** for further details on how the various different types of beacon can be identified and used.

Editing bledata.txt

The easiest way to change the BLE beacon data that the DTAG100 broadcasts is to plug the DTAG100 into your computer, browse to the DTAG100 mass storage drive, and open the **bledata.txt** file in a text editor (eg Windows Notepad).

For a standard iBeacon, or a Connectable beacon, you'll need to provide UUID, major and minor values according to the Apple specifications, plus a name that appears in logs for your reference. You can also include the advertised power and BLE local name that will override any default values that are set in **config.txt**

For example, to specify a standard iBeacon, advertising at the default transmission power, your **bledata.txt** file might look like this:

```
!DTAGble
beacon=i;MyiBeacon;000102030405060708090A0B0C0D0E01;1;1
```

To configure your DTAG100 as a Connectable beacon, and with parameters that will work immediately with the Bipzone app, your **bledata.txt** file should look like this:

```
!DTAGble
beacon=c;DotBeacon;a265660de11b5330510666978bd834fe;101;1;C6;_AUTO_
```

For an Eddystone beacon, you can set different parameters depending on the Eddystone frame type(s) you wish to use, plus a name that will appear in the logs for your reference and, optionally, advertised power and BLE local name values that will override those in **config.txt**.

To configure your DTAG100 to transmit an Eddystone URL beacon at default power levels, your **bledata.txt** file could look like this:

```
!DTAGble
beacon=u;MyEddyURL;http://www.dotorigin.com
```

Note that, in the case of Eddystone URL beacons, there is a maximum length of 18 characters for the transmitted URL, but this is after it has been shortened/encoded according to the Eddystone specifications, and so the actual URL specified in the file can usually be longer. The DTAG100 will automatically perform this encoding for you, for example replacing ‘http://’ and ‘.com’ with the specified single-character equivalents.

To transmit an Eddystone URI beacon you can use the following example:

```
!DTAGble
beacon=d;MyEddyUID;01020304050607080900;AABBCCDDEEFF
```

And to transmit (just) an Eddystone telemetry beacon, you would configure your DTAG100 as follows, although it is more likely that you would choose to transmit this beacon as part of a sequence of multiple beacon IDs as described later in this chapter:

```
!DTAGble
beacon=t;MyEddyTLM;0000;8000
```

Finally, you can create your own beacon format, or match another one used elsewhere, by using the DTAG100’s raw data mode, where you have full control over the BLE advertising data packet by specifying up to 31 bytes in hex:

```
!DTAGble
beacon=r;RawBeacon;0201060303D8FE0E16D8FE20000044544147313030007
```

Beacon parameter specifications

The detailed specifications for how to define the transmission data within the **bledata.txt** file for each available beacon format are given below:

iBeacon mode

```
beacon=i;<log_name>;<uuid>;<major>;<minor>;[advertised_power]
```

Connectable beacon mode

```
beacon=c;<log_name>;<uuid>;<major>;<minor>;[advertised_power];[local_name]
```

Eddystone URL mode

```
beacon=u;<log_name>;<uri>;[advertised_power];[local_name]
```

Eddystone UID mode

```
beacon=d;<log_name>;<namespace>;<instance>;[advertised_power];[local_name]
```

Eddystone Telemetry (TLM) mode

```
beacon=t;<log_name>;<voltage>;<temperature>;[local_name]
```

Raw data mode

```
beacon=r;<advertising_data>
```

The following table provides further information on how the various beacon parameters above need to be provided in the bledata.txt file:

Parameter name	Applicable beacon format(s)	Parameter data format
`log_name`	All apart from raw data	Alphanumeric string (up to 16 chars)
`uuid`	iBeacon & Connectable	32 hex digits
`major`	iBeacon & Connectable	Decimal number (0..65535)
`minor`	iBeacon & Connectable	Decimal number (0..65535)
`advertised_power`	All apart from raw data/TLM	2 hex digits (00..FF)
`local_name`	All apart from raw data	Alphanumeric string (up to 11/25 chars)
`uri`	Eddystone URL	Alphanumeric string (max 18 after encoding)
`namespace`	Eddystone UID	20 hex digits
`instance`	Eddystone UID	12 hex digits
`voltage`	Eddystone TLM	4 hex digits (0000=not supported)
`temperature`	Eddystone TLM	4 hex digits (8000 = not supported)
`advertising_data`	Raw data only	Up to 62 hex digits (must be LTV encoded)

Please see **Appendix C** for further details on the different beacon types and parameters they use.

Multiple beacon IDs

The DTAG100 is capable of broadcasting multiple beacon advertisements in quick succession - so that mobile devices may see them as separate beacons. This means one DTAG100 unit can optionally appear as several different beacons, to allow multiple brands or actions to be triggered, or to comply with multiple beacon transmission formats simultaneously (eg iBeacon and Eddystone).

You can control the number and format of the beacons, the interval of time between advertisement switching, and how long advertisements are broadcast for. We recommend using no more than 8 different successive beacons per DTAG100 to ensure a reliable experience, and in some cases less, otherwise performance of your apps may degrade as the number of beacons increases, due to the scanning frequency of the mobile OS and other timing interactions.

To set multiple beacons, simply include multiple ‘beacon=’ lines in the **bledata.txt** file and they will be advertised in the same sequence as found in the file. You must also specify the speed with which the DTAG100 will cycle between the beacon data you have provided with the interval value below, specified in decimal milliseconds:

```
!DTAGble
beacon=i;MyiBeacon1;000102030405060708090A0B0C0D0E03;3;3
beacon=i;MyiBeacon2;000102030405060708090A0B0C0D0E04;4;4
beaconInterval=500
```

Note that there are no restrictions on mixing different types of beacons when using this facility, although you must be careful if using this feature in conjunction with an app that is designed to connect to the DTAG100 and read or write some data (beacon adverts will currently be temporarily disabled while the connection is processed).

Bluetooth settings in config.txt

You can change various settings in **config.txt** to affect the way the Bluetooth beacon transmissions are performed, as follows.

To enable or disable Bluetooth beacon transmissions (1 enables, 0 disables):

```
BLEEnable=1
```

To set the actual transmission power from 0 to 15 (default is the maximum setting of 15):

```
BLETxPower=15
```

To set the advertised signal strength for this beacon (specified in 2's compliment hex, so C6 is -58 – see **Appendix D** for more information):

```
BLEAdvPower=C6
```

To set the local device name that is advertised when appropriate:

```
BLELocalName=DTAG100
```

Note that the length of the local name is limited to as few as 11 characters, in some cases, and 25 characters in others, depending on the configuration.

There is an additional option available that causes the local name to be automatically generated based on the beacon advertisement data, as well as any attribute data. To use this mode, simply set the local name to `_AUTO_`.

Getting the Bluetooth MAC address

The device MAC address can be obtained by reading at the **bleinfo.txt** file. This file also contains the version of Bluetooth firmware that is running and other relevant information.

Other advanced features

Please refer to later sections of this guide for information on more advanced features of the DTAG100, including logging, two-way data transfer and various additional features and configuration settings.

Sending data to mobile apps

You can configure the DTAG100 so that when a mobile app is launched, some additional data is available for the app to read and act upon.

The method of launching the app and reading the data depends on the proximity technology being used, as detailed below:

How to send data to your app using NFC

Android, Blackberry, Symbian

Create an external record with a custom type (ie, “myapp.com:open”) and a payload consisting of the data you want sent to your app. On these systems, there are mechanisms for your app to register for the custom type and receive the payload data. For Android, this process is handled through Intents and is outlined here:
http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#obtain-info

Windows 8 or Windows Phone 8

See the LaunchApp records in **Appendix E** for the best approach.

How to send data to your app via BLE

A mobile application can connect to the DTAG100, after which it can read certain beacon characteristics, as long as the advertised beacon is configured to be connectable.

These characteristics are in a Bluetooth GATT service on the DTAG100 whose service UUID is "46f582f2-b45c-44f2-9723-dbbdc9b07f70”.

There are 8 readable characteristics with a similar UUID but ending with “...71” through “...78”. The content of the readable characteristics can be set in the **bledata.txt**, so that you can change them in real time. For ease of use, you can set these with a single value in bledata.txt, and the DTAG100 will split this into 32 byte chunks in each of the 8 characteristics. Only ASCII characters should be used, so you have a maximum of 256 characters you can transmit at any one time:

```
readAtt=your data
```

On iOS you will most likely need to use the beacon’s local name in order to identify the correct device to connect to prior to reading the data.

An alternative data format is also available using the file **bleattr.txt** – please contact us for more details.

Sending data back from mobile apps

As well as reading data from the DTAG100, mobile apps can write data to it, effectively enabling a two-way channel for communications between the mobile device and the USB host system, via the DTAG100. Note that the tag and/or beacon data itself cannot be changed by a mobile app – this is under exclusive control of the connected USB host.

Writing data via NFC

To write data to the DTAG100, you should simply treat it just like a Type 3 NFC tag and write an NDEF message to it. You will need to format your NDEF message correctly in order for your write events to be logged, and will normally need to enable one of the advanced log modes in order to receive the data that has been written (see Event Logging below).

Writing data via BLE

A mobile application can connect to the DTAG100, after which it can read and write to certain additional beacon characteristics, as long as the advertised beacon is configured to be connectable. These characteristics are in a Bluetooth GATT service on the DTAG100 whose service UUID is "46f582f2-b45c-44f2-9723-dbbdc9b07f70”.You will normally need to enable one of the advanced log modes in order to receive the data that has been written (see Event Logging below).

There are 4 writeable characteristics with a similar UUID but ending with “...81” through “...84” that are part of block A. There are 4 more writeable characteristics with a similar UUID but ending with “...85” through “...88” that are part of block B. There are no behavioural differences between block A and B, other than you can control where they are logged independently using the log source settings.

Please see **Appendix F** for more info on writing data to the DTAG100.

Event logging

Viewing a record of activity

The file **Log.txt** is a rolling log of certain DTAG100 events. For example, by default on an NFC read event, the date and time of the read is recorded, followed by the name of the tag data that was read. On a valid NFC write event, the data between the write markers is recorded as hex encoded binary data, in blocks of 16 bytes per log line. This data is logged following the date, time, tag data name that was in tagdata.txt at the time of the write, and the block number. When a read and write are both done in the same NFC session, only the write will be logged.

The log.txt file only holds 500 lines of logs, after which point it overwrites the first log entry and continues in a circular fashion. It is not designed to be edited by the attached host, since the DTAG100 updates it itself.

Using the internal clock

The DTAG100 has a built in real time clock which is used in the log.txt and boot.txt files to record when events occur. You can set this clock by editing the **timeset.txt** file. As soon as you save the file, the DTAG100 will update its clock.

The clock will continue to operate as long as the DTAG100 has power, but as soon as power is cut, or it is rebooted, the time will be reset. This is why we have included a battery slot on the board. You can fit a CR1220 3V battery to the DTAG100 board so that time will be kept even if the USB power is disconnected.

Advanced log modes

The DTAG100 can also send event data to your computer via USB in the form of log events. The DTAG100 can do this by acting as a keyboard or as a virtual serial COM port. You'll need an application on the computer to receive and process these log events in an appropriate manner.

The log events can be reads from, or writes to the DTAG via either NFC or BLE. You can set which types of events get logged with the ComPortMode or KBLogMode settings in **config.txt**:

0 = Logging is disabled (default)

1 = Only read events are logged, with entries sent as they appear in log.txt

2 = Read and write events are logged. For NFC, write events are scanned for special markers ("<-" and "->") and data between them is sent in raw hex

3 = Only write events are logged. For NFC, write events are scanned for special markers ("<-" and "->") and data between them is sent as ASCII text

The virtual serial COM port is disabled by default on the DTAG100 to ensure maximum platform compatibility out of the box, while keyboard emulation is always enabled at the device level. To enable the COM port, change the following setting in the DTAG100's **config.txt** and reboot the device:

```
ComPortEnable=1
```

On Windows Vista and later, you should install the appropriate driver before enabling the DTAG100 COM port, by right clicking the DTAG100.inf file supplied on the device and choosing "Install". On Windows XP, you should enable the virtual COM port as described and then browse to the DTAG100.inf file provided when you are prompted to install the device driver.

Log sources

In addition to choosing where log events end up going with the log modes above, you can choose what types of events go to each destination independently, using log sources. There is a separate log source setting for the keyboard, the COM port, and the log.txt file.

```
KBSource=7

ComPortSource=7

LogFileSource=15
```

The possible values are bitwise combinations of the following:

BLE block A writes - 1
BLE block B writes - 2
NFC writes - 4
NFC reads - 8
BLE connections (keyboard and COM port only) – 10h
BLE scans (keyboard and COM port only) – 20h
NFC field detections (keyboard and COM port only) – 40h

Note that the value in each source option is expected to be a hexadecimal number (without the trailing 'h' indicator), with a value of 7 will only log NFC and BLE writes.

Note - previous versions of the DTAG100 firmware, the BLE scan and NFC field detection log sources were enabled through BLEScanLog and NFCFieldLog settings, which have since been replaced by the above scheme.

These values do not change the individual log mode options, which still determine the format and content of the data, but keep in mind some combinations of the log mode and log source settings will have no effect.

Other advanced features

Controlling the LED behaviour

The DTAG100 has 4 blue LEDs on the board to indicate activity and attract users. You can set how these LEDs behave by making a change to the **config.txt** file on the DTAG100.

LEDMode=0 is the default setting. The value is a bit mask containing 3 settings:

Bit 0	Idle – LEDs cycling or off (0=cycling, 1=off)
Bit 1	Reaction to device in NFC field – all LEDs on or no reaction (0=all on, 1=no reaction)
Bit 2	Reaction to BLE connection – all LEDs on or no reaction (0=all on, 1=no reaction)

Note: Prior to firmware V2.3 the LEDMode setting used slightly different values. If you are updating from an earlier version and have used this feature please therefore check and adjust your code accordingly.

Disabling NFC and/or Bluetooth

If you need to configure a DTAG100 device so that the NFC or Bluetooth functions remain turned off even if the hardware interface module is present, you can do so with these two separate values in **config.txt**:

```
NFCEnable=0

BLEEnable=0
```

The default value for these is 1, which enables the interfaces. If NFC is enabled you will need a properly formatted tagdata.txt, and if BLE is enabled you will need a properly formatted bledata.txt as well as the Bluetooth module. If you have enabled the interfaces and their data files are not present and valid, three of the DTAG100 LEDs will remain lit to indicate there is an error, and boot.txt will show status=2 for NFC error, or status=40 for BLE error.

Issuing commands

The DTAG100 supports a few direct commands that are sent via a dedicated file **command.txt**. They are “reboot”, “remount”, and “refresh”. Reboot will power cycle the DTAG100, as if the USB cable was disconnected momentarily. Remount will remove the drive from the operating system briefly and then attach it again. This will attempt to force the operating system to re-read any changes that were made to the file system by the DTAG100.

This is useful if you want to read the up to date log information for example. Windows will not recognize the changes made to log.txt by the DTAG100, but if you use the remount command, Windows will re-read all of the files. Refresh will force the DTAG100 to re-read tagdata.txt and config.txt. This is only useful if you have renamed a file to be tagdata.txt or config.txt because file renaming is not recognized by the DTAG100. The **command.txt** format is below:

```
!DTAGcommand
reboot
```

Optional delays

There are two additional configuration options in **config.txt** that change the default behaviour of the DTAG100. First, you can change the time between the DTAG100 seeing "!DTAG" written to its file system and the time it re-reads the text files. This is called the update delay, which by default is one second. If you find that one second is too soon for your situation (some systems, depending on how the write is done from a low level, may delay writing the entire file for more than a second), you can increase this to a value that works for you. The other additional option is to change the time between dismounting and remounting the DTAG100 when issuing the remount command. This is called the remount delay, which by default is 5 seconds.

Flash Storage

The DTAG100 contains embedded flash memory that stores the text files for use during default operation. There is also an option to use a micro SD card instead of the embedded flash memory. The DTAG100 does not function differently and both have the same features. The difference being that the embedded flash is limited in size to 460 KB while the micro SD can be up to 2 GB. If a larger SD card is used, you may need to format it first. You may find the extra space useful for sending larger tag data over NFC with the DTAG100, like larger NDEF records or more than one NDEF record.

Be aware that the DTAG100 may show two different removable disks in Windows, but only one will ever have its contents viewable. If there is no SD card inserted, you can view and edit the files on the embedded flash memory, but the SD card drive will appear empty. If there is an SD card inserted, you can view and edit the files on it, but the embedded flash drive will appear empty.

Updating the firmware

To update the main firmware of the DTAG100, you should copy the firmware image file to the root of the DTAG100's mass storage. Make sure the file is named **firmware.dat** then reboot the device. There will be a delay of a couple seconds when the DTAG100 boots up again and performs the update, but then it will continue to operate as normal. You can check the **boot.txt** file to see if the firmware version is what you expect.

The BLE firmware can also be updated using a similar mechanism. Please follow the instructions provided with the BLE firmware update file.

Appendix A: Reference configuration files

The following example files can be copied and installed on your DTAG100 and/or used for reference purposes:

Tagdata.txt

```
!DTAG:MyTagName
SmartPoster
URI=http://www.dotorigin.com
Text=This is Dot Origin

;
; Edit this file to change the DTAG100 NFC tag contents.
;
; For a standard NFC smart poster, edit the URI= line and (optional) Text= lines above, change the
; tag name after the colon on the first line above, and save the file.
;
; For more complex NFC messages, you can include one or more external type NFC records. The external
; type is used by your receiving application to filter on, and the external payload is the data you
; want to send to your application. You can use external types to launch any Android application, by
; using the type specified in Ext2Type below, and replacing Ext2Payload with the app's "package name"
;
; The new tag data will be active immediately after the file has been written. If the lights on the
; front of the DTAG100 stop their usual cycle, then a file format error has probably been detected.
;
; Note that the tag name is important, and must consist of at least one character. It is used to
; identify the specific tag data in the cyclic log that is written to each time the DTAG is read.
;
; Note that only the first 255 characters of the URI and Text fields are used - any extra characters
; will be ignored. Any lines starting with a semi-colon (like these) will be ignored as comments.
;
; Example extended tag:
;
;SmartPoster
;Ext1Type=appdomain.com:theapp
;Ext1Payload=apppayload
;URI=http://www.dotorigin.com
;Text=This is Dot Origin
;Ext2Type=android.com:pkg
;Ext2Payload=com.yourdomain.yourapp
;
```

Bledata.txt

```
!DTAGble
beacon=i;MyBeacon;000102030405060708090A0B0C0D0E01;1;1

;
; Edit this file to change the DTAG100 BLE beacon broadcast data.
;
; Note that you must have the appropriate DTAG100i BLE hardware module to enable the Bluetooth
; functionality.
;
; For a standard Apple-compliant iBeacon, edit the beacon= line above to change the UUID, major
; and minor values in the following format (the advertised power value is optional):
;
;beacon=i;<name for logging>;<uuid>;<major>;<minor>;[advertised_power]
;
; Here is how you can specify multiple beacons, which will be transmitted quickly in sequence:
;
;beacon=i;MyBeacon1;000102030405060708090A0B0C0D0E03;3;3
;beacon=i;MyBeacon2;000102030405060708090A0B0C0D0E04;4;4
;beaconInterval=500
;
; You can also change the beacon into a connectable beacon, in which case you can specify additional
; attribute data that can be read by a mobile device, and enable other advanced features including
; writing data back to the host. Please refer to the DTAG100 documentation for more information.
;
; Here is an example of how to define a connectable beacon along with a local name and some
; associated readable data in the form of a URL:
;
;beacon=c;MyBeacon;000102030405060708090A0B0C0D0E09;9;9;C6;DTAGTest
;readAtt=http://www.dotorigin.com/
```

Config.txt

```
!DTAGconfig
LEDMode=0
ComPortMode=0
ComPortEnable=0
KBLogMode=3
;WritePwd=MyPassword
UpdateDelay=1
RemountDelay=5
TagCommandTimeout=1000
BLEEnable=1
NFCEnable=1
BLETxPower=15
BLEAdvInterval=160
BLEAdvPower=C6
BLELocalName=DTAG100
BLEScanLog=0
BLEConnectTimeout=5000
NFCFieldLog=0
KBSource=7
ComPortSource=7
LogFileSource=15
;
;
; This file contains various configuration settings for controlling how the DTAG100 behaves.
; Some settings require the device to be rebooted before they take effect. The BLE-related
; settings depend on the optional DTAG100i Bluetooth module being present at startup.
;
;
; LEDMode - controls how the LEDs react to devices in the field. The value is a bit mask containing
; 3 settings. The default value is 0.
;
;  Bit 0    Idle – LEDs cycling or off (0=cycling, 1=off)
;  Bit 1    Reaction to device in NFC field – all LEDs on or no reaction (0=all on, 1=no reaction)
;  Bit 2    Reaction to BLE connection – all LEDs on or no reaction (0=all on, 1=no reaction)
;
; ComPortMode - controls how the DTAG100 logs events over its virtual COM port interface
;  0 = Logging to the COM port is disabled (default)
;  1 = Only read events are logged to the COM port with entries sent as they appear in log.txt
;  2 = Read and write events are logged to the COM port. For NFC, write events are scanned for
;      special markers ("<-" and "->") and data between them is sent to the COM port in raw hex.
;  3 = Only write events are logged to the COM port. For NFC, write events are scanned for special
;      markers ("<-" and "->") and data between them is sent as ASCII text to the COM port.
;
; ComPortEnable - set this to 1 and then reboot to enable the COM port interface.
;      A value of 0 disables it. You should install the COM port driver first, according to
;      the documentation provided.
;
; KBLogMode - controls how the DTAG100 logs events over its keyboard emulation interface
;  0 = Logging to the keyboard interface is disabled (default)
;  1 = Only read events are logged to the keyboard with entries sent as they appear in log.txt
;  2 = Read and write events are logged to the keyboard. For NFC, write events are scanned for
;      special markers ("<-" and "->") and data between them is sent to the keyboard in raw hex.
;  3 = Only write events are logged to the keyboard. For NFC, write events are scanned for special
;      markers ("<-" and "->") and data between them is sent as ASCII text to the keyboard, using
;      US keyboard layout.
;
; WritePwd - set a write password that must be included in data written to the DTAG100 in order
; for the write to be recognized and logged. If set, you must preceed your written data with:
; [-your_write_password-]
;
; UpdateDelay - this value is the number of seconds between the DTAG100 seeing "!DTAG" written
; to its file system and the time it re-reads the text files. Used in reading tagdata.txt,
; config.txt, command.txt, and timeset.txt
;
; RemountDelay - this value is the number of seconds between dismounting and remounting the
; DTAG100 when the issuing the "remount" command through command.txt
;
; TagCommandTimeout - this value is the number of milliseconds that the DTAG100 waits after
; the last read/write command it receives via the NFC interface before it assumes the data
; exchange is complete (and therefore writes/sends its log data). The minimum value is 250
; and the setting is only accurate to the nearest 250.
```

```
;
; BLEEnable – controls whether BLE is active (default on, if BLE module present)
;  0 = off
;  1 = on
;
; NFCEnable – controls whether NFC is active (default on)
;  0 = off
;  1 = on
;
; BLETxPower - sets the actual Bluetooth transmission power, from 0..15
;
; BLEAdvInterval - sets the Bluetooth advertising broadcast repeat interval, in multiples of
; 625uS. Should not normally be changed from default setting of 160.
;
; BLEAdvPower - sets the default Bluetooth advertised signal strength. Specified in 2's
; complement, so C6 is -58. Can be overridden using individual beacon data in bledata.txt
;
; BLELocalName - sets the default local name that is advertised via Bluetooth (25 ASCII
; characters max, depending on mode). Can be overridden using individual beacon data in bledata.txt
;
; BLEConnectTimeout - the amount of time (in mS) that a Bluetooth connection is allowed to
; be kept open by a mobile device, before the DTAG100 forcefully closes it. Default 5 sec
;
; KBSource, ComPortSource, and LogFileSource - respectively controls what log events go to
; the keyboard, COM port, and log file log modes. The values are determined from a bitwise
; combination of the following hexadecimal values
;  1 = BLE block A writes
;  2 = BLE block B writes
;  4 = NFC writes
;  8 = NFC reads
;  10 = BLE connections (keyboard and COM port only)
;  20 = BLE scans (keyboard and COM port only)
;  40 = NFC field detections (keyboard and COM port only)
;
```

Timeset.txt

```
!DTAGtime
2014,01,01,12:34:56
;
; Edit this file to set the real-time clock within the DTAG100
;
; The time format is YYYY,MM,DD,HH:MM:SS
;
; The clock is only set when this file changes. Unless a battery is fitted
; in the DTAG100, the clock time will be reset whenever power is removed.
;
```

Command.txt

```
!DTAGcommand
;reboot
;
; Use this file to send various commands to the DTAG100.
;
; Type the command you want above (without the semi-colon in front) and when the file
; is saved, the command will be sent to the DTAG100 and actioned immediately.
;
; Valid commands are:
;
; reboot - this will power cycle the DTAG100, as if the USB cable was disconnected
;
; remount - this will briefly remove the drive and then attach it again
;
; refresh - this will force the DTAG100 to re-read all text files on its drive
;
```

Appendix B: Example NFC tag formats

The information below includes a description of tag type followed by the format of the URI data needed to perform the example action. The angle brackets (<>) should be replaced with your specific values, eg “geo: 51.511,-0.119”

Opens the location on a map
geo:<latitude>,<longitude>

Opens the location on a map at a specific zoom level
geo:<latitude>,<longitude>?z=<zoom #>

Opens the map and does a search around the location
geo:0,0?q=<my+street+address>

Opens location in Google StreetView
google.streetview:cbll=<lat>,<lng>&cbp=1,<yaw>,,<pitch>,<zoom>&mz=<mapZoom>

Calls a number directly
tel: <phone_number>

Opens the link in a browser
<http://web_address>

Opens the link in a browser
<https://web_address>

Composes a text message
sms:<phone_number>?body=<message_text>

Composes an email
mailto:<email_address>?subject=<subject_text>&body=<body_text>

Opens the IMDB title activity
imdb:///title/<titleID>

Opens the IMDB name activity
imdb:///name/<nameID>

Opens the IMDB and searches
imdb:///find?q=<search_query>

Opens the Google Play page for the app by package name
market://details?id=<Package_name>

Opens Google play and searches
market://search?q=<Search_Query>

Opens the Google Play page for the publisher
market://search?q=pub:<Publisher_Name>

Opens Skype and does specified action
skype:<username|phonenumber>[?[add|call|chat|sendfile|userinfo]]

Opens Spotify to specifics
spotify:<artist|album|track>:<id>

Opens Spotify and searches
spotify:search:<text>

Opens Spotify to playlist
spotify:user:<username>:playlist:<id>

Opens a compatible Telnet app and makes a connection
telnet://<user>:<password>@<host>[:<port:/]

Opens a compatible FTP app and makes a connection
ftp://<user>:<pass>@<host>/

Appendix C: More on beacons

We allow for the DTAG100 to be fully Apple iBeacon compliant, so that you can set its configuration in a way that most iOS devices will immediately understand. Other platforms including Android can also understand this transmission format, so it is not limited to any one platform and serves to as a useful and uniform format.

In addition, the DTAG100 supports multiple alternative beacon formats, including its own Connectable beacon format and Eddystone beacons (all three frame types). If none of these are suitable for your application then a general-purpose custom-format beacon mode is also available.

More on iBeacons and Connectable beacons

Apple iBeacons use three values to identify themselves: UUID, major, and minor. The UUID is a 32 digit hexadecimal value that should normally be unique to your brand and collection of beacons. The major value is a 5 digit decimal value (0 to 65535) that is a refinement in scope within your collection. You could use it to classify broad locations of your deployment or organize the DTAG100 devices into different purposes. The minor value is a second 5 digit decimal value (0 to 65535) that is to further define scope, be it a specific location or data set that it is meant share. These three values combined will tell your application how to interact with a specific DTAG100, so must be chosen carefully so that your deployment can react to all scenarios that you require.

The last bit of information that is included with the advertisement is an indication of signal strength. To clarify, this is the signal strength that the beacon is telling other devices that it is broadcasting at, so that they can make a judgement as to how far away from the DTAG100 they are by comparing it to the actual received signal strength.

Connectable beacons can have a local name, which is a text-based name that is transmitted in addition to the advertisement of the UUID, major, and minor values. Mobile apps can connect to this type of beacon in order to read additional data and to and write data via the DTAG100 to the attached host system.

More on Eddystone beacons

An Eddystone beacon is a beacon format created by Google that is compatible with Android and iOS applications. Most notably it is capable of advertising URLs to mobile apps without requiring a direct connection between the beacon and phone. In addition to advertising URLs, Eddystone beacons can also advertise UID values similar to iBeacon UUIDs, and/or telemetry data including battery voltage and ambient temperature.

Eddystone UIDs are separated into a 10 byte namespace value (20 digit hexadecimal value) and a 6 byte instance value (12 digit hexadecimal value). Though not enforced, it is suggested that the namespace identifies a group of beacons, and the instance identifies an individual beacon.
Eddystone telemetry data contains battery voltage, beacon temperature, advertising PDU count, and the time since the beacon was powered on. The DTAG100 does not currently provide any actual

voltage or temperature measurements, but you can test your apps by setting those values on the DTAG100 to be whatever you wish, and/or feed useful values from the connected host system. The voltage and temperature are each 2 byte values (4 digit hexadecimal values). As per the Eddystone spec, the values of 0000 and 8000 respectively, indicate to scanning apps that voltage and temperature are not supported. It will correctly advertise the advertising PDU count and time since it was powered on.

Eddystone beacons are meant to be used in conjunction with Google's Proximity Beacon and Nearby Messages APIs for Android and iOS. You can additionally make use of the two-way data transfer features provided by the DTAG100, in the same way as Connectable beacons.

Other beacon formats

The DTAG100 also provides a raw data mode where the full 31 bytes of BLE advertising data can be specified and changed at any time.

Appendix D: BLE advertised power calculations

When the DTAG100 is configured as a BLE beacon using either the Apple-compliant iBeacon data transmission format, or the alternative Connectable beacon format, the regularly-transmitted data packet includes a value that represents the strength of the Bluetooth radio transmission when measured 1m away from the beacon. This is used by receiving devices to estimate their own distance from the beacon, which they can do by comparing the advertised power value with the actual received signal strength that they see.

The strength of the actual transmission will depend on a number of factors, not least the choice of Bluetooth antenna, it's location, and the amount of metal and other materials nearby. If your application uses distance estimation, it will therefore be important to calibrate each DTAG100 installation to ensure that the advertised power value (BLEAdvPower) is set correctly, and according to the actual transmission power being used at any one time.

The value for the advertised power can be set in **config.txt**, to match the transmitted power which is also set there, but can also be overridden as part of the individual beacon configuration data in **bledata.txt** if required. Because radio signal strength values are typically measured as negative values (eg -58dBm) the hex value that should be used for advertised power must be calculated using 2's complement binary arithmetic.

The following table provides suggested values to use for BLEAdvPower when using the standard DTAG100 internal antenna at the various transmission power settings available:

BLETxPower	BLEAdvPower
15	C5
14	C1
13	C0
12	BE
11	BE
10	BD
9	BB
8	BA
7	B8
6	B6
5	B4
4	B1
3	B0
2	B0
1	AB
0	AA

Appendix E: More on launching apps

How to launch an app automatically using NFC

Android

Use Android Application Records (AAR), which is an external record with type “android.com:pkg”. The payload is the app’s package name (ie, com.google.android.youtube). If the app is not present on the phone, it will navigate to it in the Play Store app. This record should be the last record in your message as to not disturb tag reading on other operating system. More information is available here:

http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#aar

Windows 8 or Windows Phone 8

Use a LaunchApp record, which is an absolute URI record with type “windows.com/LaunchApp”. The payload starts with the data you wish to pass to your app, followed by platform and app name pairs (meant to be extensible). If the app is not present on the device, it will ask to install it from the Microsoft Marketplace. Creating this type of record is not currently possible through the DTAG Producer. You would need to create a tagdata.txt file with the raw hex data manually. This record must be the first record in your message. More information is available here:

http://social.technet.microsoft.com/wiki/contents/articles/13539.how-to-launch-apps-via-proximity-apis-nfc.aspx#LaunchApp_Tags

There is a good visualization of creating cross-platform app launching tags here:
https://social.technet.microsoft.com/wiki/contents/articles/13955.how-to-create-cross-platform-launchapp-nfc-tags.aspx#Cross-Platform_LaunchApp_Tag

How to launch apps using BLE

Currently, latest iOS devices are capable of launching apps for a limited time, in response to detecting one or more specified iBeacon UUIDs. The app must be installed and must be ‘beacon aware’ before this functionality is available.

Please refer to the Apple developer documentation for more information.

On Android you will need to develop an app that runs continuously in the background in order to scan for beacons and then notify the user as appropriate. Later version of Android include facilities to help you do this – please see the Android developer documentation for more information.

Appendix F: More on writing data

How to write to the DTAG100 from Android via NFC

The DTAG100 will recognize any data that is written to it over NFC that is between ASCII encoded "<-" and "->" markers. It will write this data over the keyboard or virtual COM port (depending on the contents of your config.txt), as long as it is preceded by a valid license code and optional password (see previous section). The easiest way to write an ASCII encoded string of data from Android, is to create an NDEF Record with Record Type Definition (RTD) of text.

You'll have to build the text payload byte array yourself according to the NFCForum-TS-RTD_Text_1.0. You can then use the NdefRecord constructor to add the additional headers for a text RTD, and the NdefMessage constructor to add that record to a message. Then write the message to your tag obtained from the intent, which should be set up in your AndroidManifest.xml file.

Below is a createTextRecord function as a starting point.

```
	private NdefRecord createTextRecord(String text) throws UnsupportedEncodingException {
		// Convert the text to hex (with the default UTF-8) and get the length
		byte[] textBytes  = text.getBytes();
		int textLength = textBytes.length;

		// Convert to the text language to hex (ensuring US-ASCII) and get the length
		String lang       = "en";
		byte[] langBytes  = lang.getBytes("US-ASCII");
		int langLength = langBytes.length;

		// Make an array of the proper size to hold the status byte, language, and text
		byte[] payload    = new byte[1 + langLength + textLength];

		// Set the status byte (which is only the length in our case of using UTF-8)
		payload[0] = (byte) langLength;

		// Copy langbytes and textbytes into the payload
		System.arraycopy(langBytes, 0, payload, 1, langLength);
		System.arraycopy(textBytes, 0, payload, 1 + langLength, textLength);

		// Create an NdefRecord with TNF=well known and RTD=text with the payload we created
		NdefRecord recordNFC = new NdefRecord(NdefRecord.TNF_WELL_KNOWN, NdefRecord.RTD_TEXT,
new byte[0], payload);

		return recordNFC;
	}
```

Ignoring the necessary error and exception handling, and that connecting and writing to NFC tags should be outside the main application thread, the flow should look something like:

```
NdefRecord[] records = { createTextRecord("[-" + yourLicenseCode + ";" + yourOptionalWritePassword +
"-]<-" + yourTextString + "->") };
NdefMessage message = new NdefMessage(records);
Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
Ndef ndef = Ndef.get(tag);
ndef.connect();
ndef.writeNdefMessage(message);
ndef.close();
```

How to write to the DTAG100 from Windows 8 or Windows Phone 8 via NFC

You can treat the DTAG100 just like any other NFC tag and write to it. You'll still need to have the data you write back to be encapsulated in our writeback markers, "<-" and "->", and preceded by your license and optional password string, "[-your_license_code;your_write_password-]".

You can use the Proximity and Proximity Device APIs to interface with the DTAG100. The Proximity Device API exists for Windows and Windows Phone 8. Here is the method to write to NFC tags:

http://msdn.microsoft.com/en-us/library/windows/apps/br241226.aspx

How to write to the DTAG100 via Bluetooth

When fitted with a Bluetooth module, the DTAG100 enables two-way communication via BLE, using the same log modes for keyboard and COM that the NFC interface uses. To achieve this, you must have configured the DTAG100 to a Connectable, Eddystone or custom beacon, since the iBeacon mode does not support this feature.

In order to write data you first need to find the DTAG100 device using your platform's Bluetooth APIs. Once found you would normally discover the device's services and then discover the characteristics of the service with UUID 46f582f2-b45c-44f2-9723-dbbdc9b07f70.

The write-back characteristic has UUID 46f582f2-b45c-44f2-9723-dbbdc9b07f81, and any data that you write to it is fed into the USB logging mechanisms you have enabled (keyboard, COM, or both) as well as recorded in log.txt. The data is written as ASCII text and has a maximum length of 32 bytes. This feature does not currently require or allow licensing or password protection like the NFC interface does, and so you do not need the "<-" and "->" markers.

To confirm a successful write, you can read back the value of the characteristic above, which will contain a count that increments whenever a write occurs to it. It is in the format "wXXXXX" where XXXXX is the zero-padded count value. After a write is detected, the count is incremented and the new count value is set for this characteristic. Note that it is possible for a fraction of a second that the value read back will be the last value that you wrote, before the DTAG100 is able to update it with the new count.