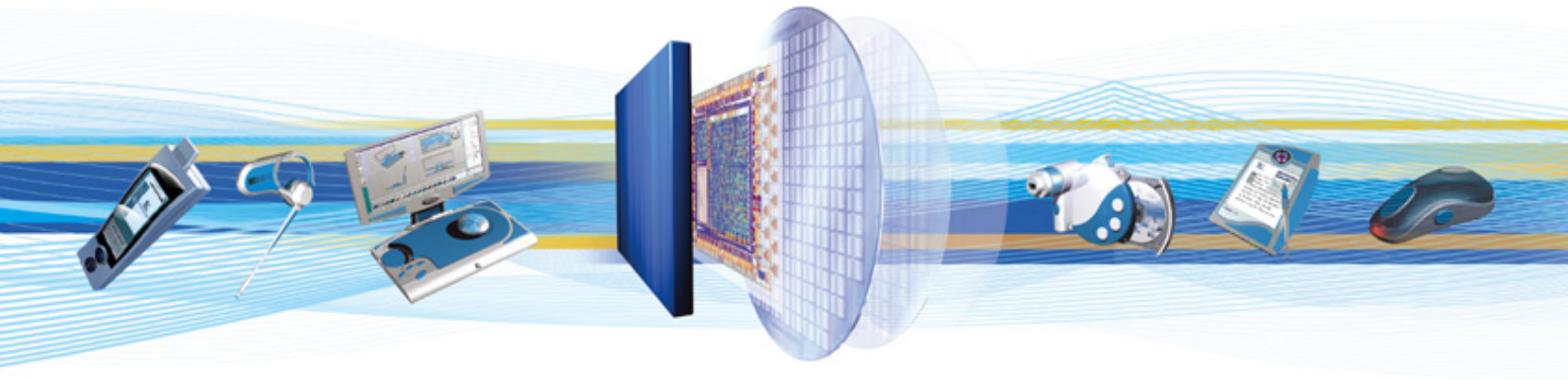
BlueCore™

# BCCMD Commands

September 2005



**CSR**

Churchill House
Cambridge Business Park
Cowley Road
Cambridge CB4 0WZ
United Kingdom

Registered in England 3665875

Tel: +44 (0)1223 692000
Fax: +44 (0)1223 692001
www.csr.com

# Contents

**List of Figures**

**List of Tables**

# 1 Introduction

Document [BCCMD] describes a protocol used to access a command interpreter on CSR's **BlueCore™** Bluetooth® wireless technology chips. The command interpreter presents commands that allow monitoring and control of the chip. The command interpreter protocol is not part of the Bluetooth standard.

This document describes the commands presented by the command interpreter.

# 2 Context

The BlueCore Command (BCCMD) protocol, described in [BCCMD], runs between a command/monitoring entity on a host processor and a command interpreter in a BlueCore chip. Figure 2.1 shows the command/monitoring entity as `bcmgr` and the chip's command interpreter as `bccmd`.

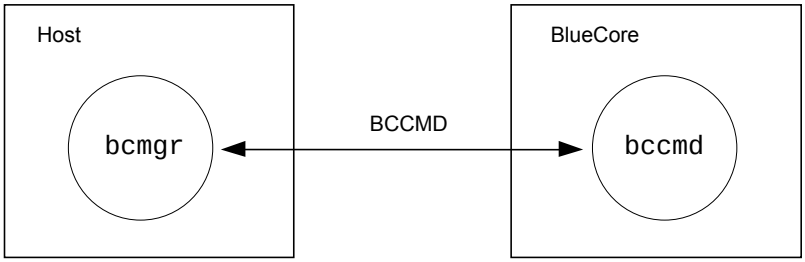



**Figure 2.1: BCCMD Protocol**

BCCMD commands flow from `bcmgr` to `bccmd`. `bccmd` handles the commands and returns responses to `bcmgr`. The general flow of processing follows the pattern of a remote procedure call (RPC) interface.

There are alternative paths over which BCCMD can flow, but these are irrelevant to this document.

This document defines the commands handled by `bccmd`.

# 3 System Control Commands

The following are commands to control the chip.

## 3.1 Cold_Reset

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4001 | Valueless | WO | WO |

This command forces a hardware reset of the chip, deliberately discarding all of its current state. The command emulates removing power from the chip, then restoring it.

It is highly probable that the command's GETRESP will not reach the caller.

## 3.2 Warm_Reset

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4002 | Valueless | WO | WO |

This command forces a hardware reset of the chip, arranging that some elements of the chip's current state may be available when the chip is restarted, assuming the chip remains powered through the reset.

For example, the host may set the "baud rate" PS Key in a BlueCore-ROM part's (psram) persistent store. The value survives a Warm_Reset, and is used on rebooting.

It is highly probable that the command's GETRESP will not reach the caller.

## 3.3 Cold_Halt

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4003 | Valueless | WO | WO |

This command halts the chip, deliberately arranging that all current state will not be available when the chip is restarted. This is like the first half of the Cold_Reset command. The chip must be power cycled or hardware reset to restore operation.

It is highly probable that the GETRESP for this command will not reach the caller.

## 3.4 Warm_Halt

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4004 | Valueless | WO | WO |

This command halts the chip, arranging that some elements of the chip's current state may be available when the chip is restarted, assuming the chip remains powered. This is like the first half of the Warm_Reset command. The chip must be power cycled or hardware reset to restore operation.

It is highly probable that the GETRESP for this command will not reach the caller.

## 3.5 Enable_TX

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4007 | Valueless | WO | WO |

This command allows the Bluetooth radio to transmit.

The Bluetooth module contains a large amount of complex code that decides which Bluetooth packets to transmit and when. This may be unacceptable for RF emission control agencies. The Enable_TX and Disable_TX commands invoke relatively simple code that talks directly to a chip hardware control register, so the chance of these commands misbehaving should be much lower than using the more complex (HCI) controls.

## 3.6 Disable_TX

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4008 | Valueless | WO | WO |

This command prevents the Bluetooth radio transmitting. See the description of the Enable_TX command.

## 3.7 Config_UART

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6802 | uint16 | RW | RW |

The chip's UART is set up at boot time from the chip's Persistent Store; this is the preferred configuration method.

Under rare circumstances it may be necessary to change the UART's configuration after booting. This command immediately configures the UART's baud rate and settings. The GETRESP for this command will probably return to the host at the new UART settings.

The low 13 bits of the `uint16` set the UART's baud rate. The value is used as a clock divider, so unusual values can be set:

```
Value = (baud rate / 244.140625)
```

Table 3.1 provides the values for some conventional baud rates.

| Baud Rate | Value |
|---|---|
| 9k6 | 0x0027 |
| 19k2 | 0x004f |
| 38k4 | 0x009d |
| 57k6 | 0x00ec |
| 115k2 | 0x01d8 |
| 230k4 | 0x03b0 |
| 460k8 | 0x075f |
| 921k6 | 0x0ebf |

**Table 3.1: Baud Rate Values (uint16 Low Bits)**

The top three bits of the uint16 control some of the UART's settings.

| Setting | Value |
|---|---|
| 1 Stop Bit | 0x0000 |
| 2 Stop Bits | 0x2000 |
| No Parity | 0x0000 |
| Odd Parity | 0x4000 |
| Even Parity | 0xc000 |

**Table 3.2: Baud Rate Values (uint16 Top Bits)**

The value of the varid is formed by ORing the values above. For example, to set one stop bit, 38.4k baud and even parity, the varid's value is as follows:

```
( 0x009d | 0x0000 | 0xc000 )
```

The Persistent Store's configuration controls give finer control over the UART hardware.

## 3.8 Bypass_UART

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4014 | Valueless | WO | WO |

The BlueCore-ROM hardware provides a "UART bypass mode" in which the four normal UART pins are routed directly to PIO pins 4 to 7. This enables another peripheral, e.g. an IR (IrDA) transceiver, to use the same host UART as BlueCore.

The command immediately forces BlueCore-ROM to enter a Deep Sleep state from which it can be woken only by a hardware reset.

## 3.9 Max_Tx_Power

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6827 | int8 | RW | RW |

When the chip boots it copies the value of the PS Key PSKEY_LC_MAX_TX_POWER to a location in RAM. This is used to control the radio's maximum transmit power, expressed in dBm.

This varid accesses this RAM value, allowing the maximum transmit power to be altered after booting. The varid's value is a signed number.

The limit is only referenced when the firmware considers increasing the radio's transmit power; if the transmit power on a link is currently above this level the new value will not take effect until an attempt is made to increase the power.

## 3.10 Default_Tx_Power

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x682b | int8 | RW | RW |

When the chip boots it copies the value of the PS Key PSKEY_LC_DEFAULT_TX_POWER to a location in RAM. This is used to control the radio's default transmit power, expressed in dBm.

This varid accesses this RAM value, allowing the maximum transmit power to be altered after booting. The varid's value is a signed number.

The default transmit power is used for paging, responding to pages, inquiring and responding to inquiries. It is also used as the initial power on freshly created ACL connections (before LMP power control takes over).

The value is normally constrained by the Bluetooth power class, though under most circumstances it should be between 0 and 4dBm.

This command can be used to set the *desired* default transmit power; it can be over-ruled by the chip's power table (see PS Key PSKEY_LC_POWER_TABLE) and by maximum transmit power constraints.

The value read via this varid obtains the actual default transmit power; this can be different from the value written, which is the *desired* default transmit power. It is quite possible that writing to the varid then reading the value back will produce a different value.

If necessary, the default TX power is reduced so that it is no larger than the maximum TX power.

# 4 System Status Commands

Commands to monitor the status of the chip.

## 4.1 BuildID

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2819 | uint16 | RO | RO |

This command returns a value that identifies the Bluetooth module's firmware version.

The value is incremented each time CSR produces a new labelled build of the firmware.

At the time of writing, the value is also available via the HCI command Read_Local_Version_Information; the same value is returned in *both* the HCI_Revision and LMP_Subversion fields. A fresh version of the HCI specification may clarify the fields' meanings, so this mechanism may change.

## 4.2 BuildID_Loader

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2838 | uint16 | RO | RO |

This command returns a value that identifies the firmware version of the module's "DFU loader".

An HCI firmware build is composed of two major components: the main Bluetooth "stack" and the (DFU) "loader". On a freshly manufactured Bluetooth module, both of these components are loaded into flash memory. If the module's firmware is subsequently upgraded using DFU (e.g. using "DFU Wizard"), then the "stack" component is replaced in flash, using functionality in the "loader" element. Thus, the "loader" element is not replaced during DFU.

The "stack" and "loader" each contain a build identifier. The BuildID command allows the host to read the "stack's" build identifier. The BuildID_Loader command allows the host to read the "loader's" build identifier.

A loader's identifier is normally identical to one of the "stack" identifiers built from the common code base. For example, four builds were made from the HCIStack1.1v16.4 code base (BlueCore01b/BlueCore2-External and 56-bit/128-bit), and each of these four "stack" builds was given a distinct identifier. The corresponding DFU "loader's" build identifier(s) for these builds is the same as one of the four "stack" identifiers.

## 4.3 ChipVer

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x281a | uint16 | RO | RO |

This command returns a value that identifies the major version number of the BlueCore chip (digital) hardware.

| Value | BlueCore Version |
|---|---|
| 0x00 | BlueCore01a (obsolete) |
| 0x01 | BlueCore01b |
| 0x02 | BlueCore2-External, BlueCore2-ROM (ST and TSMC variants) |
| 0x03 | BlueCore3-Multimedia, BlueCore3-ROM, BlueCore3-Flash, BlueCore4-External, BlueCore4-ROM |
| 0x04 | Reserved for future chip versions |

**Table 4.1: ChipVer**

## 4.4 ChipRev

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x281b | uint16 | RO | RO |

This command returns a value that identifies the minor version number of the BlueCore chip (digital) hardware:

| Value | BlueCore Version |
|---|---|
| 0x64 | BlueCore01b, engineering sample (ES) version |
| 0x65 | BlueCore01b, production version. (previously referred to as “metal-mod” version) |
| 0x89 | BlueCore2-External, version a. (aka engineering sample 2 {ES2}) |
| 0x8a | BlueCore2-External, version b |
| 0x28 | BlueCore2-ROM |
| 0x43 | BlueCore3-Multimedia |
| 0x15 | BlueCore3-ROM |
| 0xe2 | BlueCore3-Flash |
| 0x26 | BlueCore4-External |
| 0x30 | BlueCore4-ROM |

**Table 4.2: ChipRev**

## 4.5 ChipAnaVer

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2836 | uint16 | RO | RO |

This command returns a version number for the chip’s analogue components.

## 4.6 Max_Crypt_Key_Length

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x282c | uint16 | RO | RO |

The maximum effective encryption key length that can be applied to a connection is 16 bytes, as defined in [BT2.0]. A number of factors can cause weaker encryption to be used:

- When encryption is applied to a link the devices negotiate via LMP to determine the encryption strength. The local device may allow 16-byte encryption, but the remote device may insist on weaker encryption.
- The local device's LMP encryption strength negotiation is also constrained by the values stored under PS Keys PSKEY_ENC_KEY_LMIN and PS Key PSKEY_ENC_KEY_LMAX.
- The firmware build may itself impose an upper limit to the encryption strength, to be applied to all links.

This varid reveals the last of these limits, measured in bytes.

Typically firmware builds support a maximum of 56- or 128-bit encryption, so this command normally returns 7 or 16.

## 4.7 Interface_Version

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2825 | uint16 | RO | RO |

This command returns the version of the bccmd interface. This allows host code to adapt itself to different versions of the bccmd interface.

The value of this varid is the version number of the source file that defines the bccmd interface in the firmware; this is incremented each time a new version of the file is generated.

This defines the version number of the interface, not the version of the implementation of commands behind the interface.

**Note:**

This command has been removed from firmware builds after HCIStack1.2v18.2.

## 4.8 BT_Clock

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2c00 | uint32 | RO | RO |

This command returns the local Bluetooth module's Bluetooth clock.

This is represented within the firmware as an incrementing 28-bit value held in a 32-bit unsigned integer, so the value will wrap after 0x0fffffff.

## 4.9 Get_Next_Builddef

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x300b | Complex | RO | RO |

This command extracts a set of identifiers that describe the major characteristics of the firmware (its "build definitions"). These definitions say such things as "this build is for bc01b" and "this is a conventional HCI build", etc.

The set of all current build definitions is described in section 15; the list is expected to grow over time.

Figure 4.1 shows the command's structure.

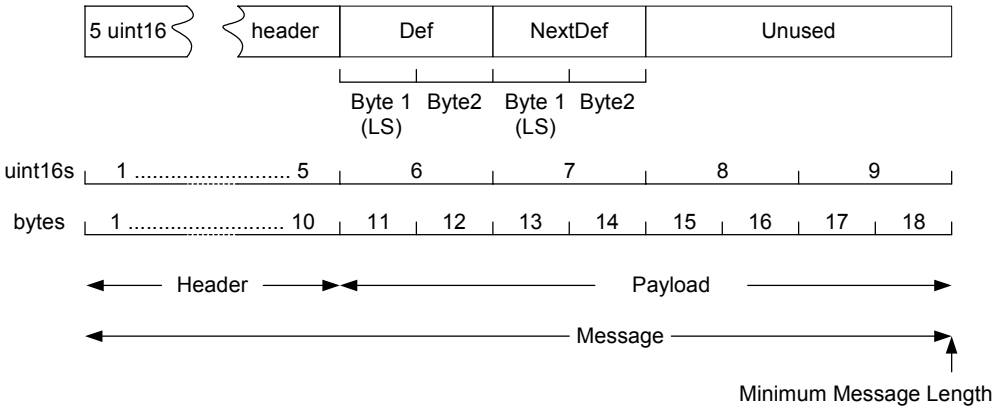



**Figure 4.1: Get_Next_Buildef Data Structure**

The command sets `NextDef` to the firmware's first "build definition" value that is numerically greater than Def, or zero if there is no such value.

A firmware build's set of build definitions can be extracted by the standard "get-next" mechanism:

- When the command is first called `Def` and `NextDef` fields are set to zero. When the call returns `NextDef` contains the first "build definition" value.
- The host then copies `NextDef` to `Def` and makes a second call to the command. On this occasion, `NextDef` contains the second "build definition" value.
- This is repeated until all of the build definition values have been extracted; this is signified by a returned `NextDef` of zero.
- The result is the host obtains the (numerically-sorted) list of "build definitions" for the firmware.

This command allows host tools to adapt to the firmware build. In particular, `pstool.exe` can adjust the set of PS Keys it presents.

## 4.10 Get_Clr_Evt

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x300a | Complex | RO | RO |

The chip firmware contains a set of counters that are used to monitor certain events; each time the event occurs a counter is incremented.

This varid obtains the value of a counter and then clears the counter to zero.

The VM also has access to these same event counters, so care must be taken if the VM and host access a single counter.

The monitoring mechanisms are not precise - it is possible for each counter to overcount or undercount. The counters must be treated as indications of the amount of a particular activity, not as an accurate count.

Each counter is held in an unsigned 16-bit integer, so the values can wrap.

Figure 4.2 shows the data structure used by this varid. The `evtcntid` identifies the counter to be read and cleared. The GETREQ must hold `cnt` set to zero. The corresponding GETRESP will carry the same `evtcntid` and `cnt` will hold the counter's value.

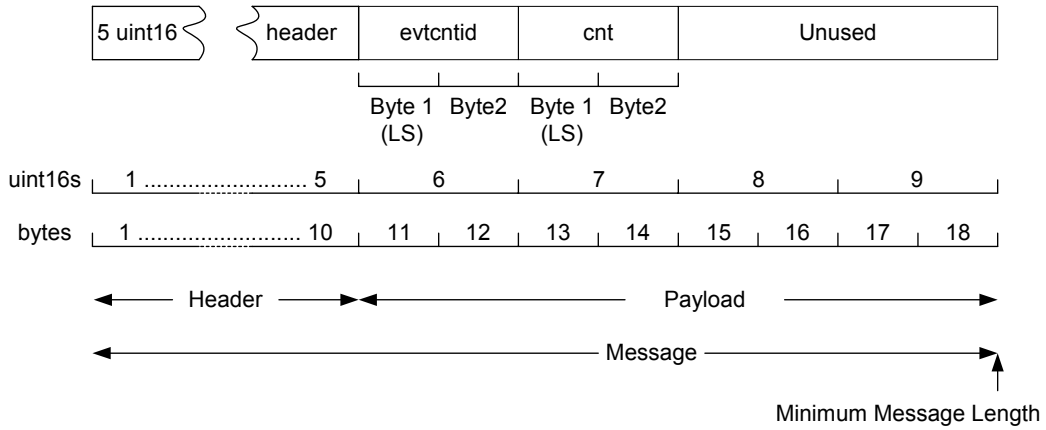



**Figure 4.2: Get_Clr_Evt Data Structure**

Table 4.3 shows the acceptable values for `evtcntid`.

| Counter Name | evtcntid | Comment |
|---|---|---|
| NONE | 0 | No counter; always returns zero |
| TXACL | 1 | Transmit radio ACL packets |
| RXACL | 2 | Receive radio ACL packets |
| TXSCO | 3 | Transmit radio SCO packets |
| RXSCO | 4 | Receive radio SCO packets |
| TXHOST | 5 | Send packets to the host |
| RXHOST | 6 | Receive packets from the host |
| RADIOACTIVE | 7 | Any radio event |

**Table 4.3: evtcntid**

Reading the NONE counter has no effect; it accesses no counter and the `count` value returned is always zero.

The firmware contains a handful of counters with `evtcntid` values greater than RADIOACTIVE. These counters are deliberately not documented here; these must not be accessed from the host as internal chip operations depend on them.

# 5 Connection Control Commands

The following are commands to control Bluetooth connections.

## 5.1 Map_SCO_PCM

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x481c | uint16 | WO | WO |

The standard HCI command “Add_SCO_Connection” normally creates a SCO data channel that flows through the HCI, using HCI SCO Data packets.

BlueCore01b provides a PCM hardware port that can be used as an alternative route for a SCO channel: SCO data flows from air, through BlueCore01b and out over the PCM port, and vice versa.

The hardware of BlueCore2-External and later chips can multiplex up to three SCO streams over its single PCM port. However, no use is made of this in the HCIStack1.1v13.x and HCIStack1.1v14.x builds.

If this varid is set to value 1 then the next HCI Add_SCO_Connection command will route the SCO channel over the chip’s first SCO interface. In HCIStack1.1v15.x builds on BlueCore2-External and later chips, values 2, 3 and 4 select other PCM channels through the PCM port.

The Add_SCO_Connection effectively clears this PCM channel mapping request, regardless of whether the call succeeds or fails, i.e. the varid prepares the chip for the next Add_SCO_Connection call only.

Only one mapping request can be pending at a time. To route the first two PCM channels the host would set the varid to 1, then call Add_SCO_Connection, then set the varid to 2, then Add_SCO_Connection for the second connection.

If this varid is set to zero it clears any pending PCM mapping request.

The chip will reject an attempt to set the varid to a non-zero value if the request exceeds the chip's resources (e.g. trying to open PCM channel 5), or if the channel is in use (the PCM channel is already carrying a SCO connection).

This varid can also be used before a SCO HCI Accept_Connection_Request command: the host receives the SCO Connection_Request event, the host sets the varid, then the host sends the HCI Accept_Connection_Request command. This should route the new SCO connection over the PCM interface.

It is not recommended that this command is used with auto-accepted HCI SCO connections as the host will normally have no control over which SCO connection will route to which PCM channel. In practice the PCM channels are likely to be hard-wired to peripheral hardware.

See [BCHCI] for a fuller description of the commands used.

**Note:**

The value of PS Key PSKEY_MAP_SCO_PCM overrides this command.

## 5.2 Scatternet_Override_SCO

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x683b | uint16 | WO | WO |

If a device is maintaining a single HV3 SCO link and is slave on another piconet then it is difficult to pass traffic on the second link.

From HCIStack1.2v18.1, the LC’s packet scheduler has been adjusted to improve the support for passing LMP traffic on the second link.

The same basic Bluetooth baseband signalling problem affects HCI/ACL traffic; this can be blocked for extended periods in this situation.

The Scatternet_Override_SCO command boosts the priority of HCI/ACL traffic, allowing it to flow. However, the prioritisation is at the expense of SCO traffic: HV3 packets are dropped to allow the ACL traffic to pass. Consequently, the SCO link's audio quality will suffer badly while ACL traffic is transferred.

This command is intended only to support brief signalling on the ACL connection, for example, an application message requesting destruction of the SCO link.

The command's `uint16` is Boolean: `0x0001` enables the support, `0x0000` disables it. The default for all builds up to and including HCIStack1.2v18.2 is `0`. For builds after HCIStack1.2v18.2 the default setting will be `1`.

## 5.3 PCM_Attenuation

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6832 | uint16 | RW | RW |

Some (Motorola) CODECs allow their gain to be controlled by the three top bits received at the end of a 13-bit PCM sample in a 16-bit PCM frame. The value of these three bits in all such samples sent from BlueCore over the PCM port is initialised from PSKEY_PCM0_ATTENUATION. The 3-bit value subsequently can be read and written using this command.

## 5.4 Limit_EDR_Power

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x7020 | uint16 | RW | RW |

Limit_EDR_Power is used in conjunction with the Enhanced Radio Power Table (configured via PSKEY_LC_ENHANCED_POWER_TABLE) to define behaviour on a per-link basis for high power EDR transmissions. If Limit_EDR_Power is disabled (default status), then EDR transmissions using power table settings with the corresponding EDR control bit set will be renegotiated to Basic Rate. If Limit_EDR_Power is enabled, then the firmware will refuse to increase power to entries with the EDR control bit set, effectively capping the maximum allowable transmit power in EDR links.

Figure 5.1 shows the data structure used by this command. Handle gives the HCI ACL handle of interest; Status is a Boolean defining whether the command is enabled or disabled. If the command is unsuccessful, the returned GETRESP will return ERROR with both the handle and status set to zero.

This command is supported on BlueCore4-External and later chips and appears in firmware supporting version 2.0 + EDR of the Bluetooth specification.

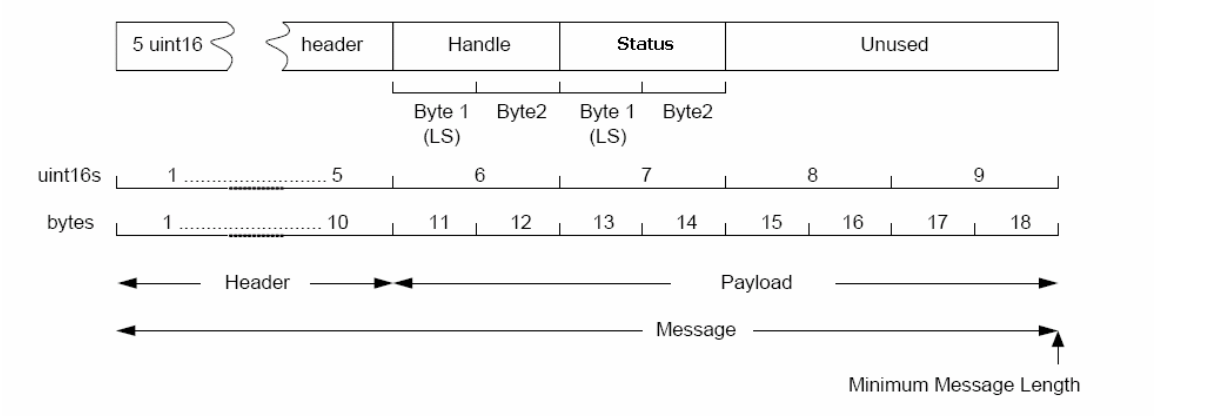



**Figure 5.1: Limit_EDR_Power Data Structure**

# 6 Connection Status Commands

The following commands are used to obtain status information on Bluetooth connections.

## 6.1 Piconet_Instant

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x3009 | Complex | RO | RO |

This command gives the Bluetooth clock used on the link identified by an HCI handle.

(The BT_Clock command gives the local device's clock. However, if the local device is an ACL slave then the link's timing is based on the link master's Bluetooth clock.)

For ease of comparison with the LMP negotiation data, the value returned by this command is *half* the Bluetooth clock value, so it wraps after `0x07ffffff`.

Figure 6.1 shows the data structure used by this command. In the BCCMD GETREQ, the "Handle" gives the HCI handle of interest; the `Clock` and `Unused` fields must be zero. If the command is successful, the returned GETRESP will hold the same `Handle`, and the `Clock` will carry (half) the link's Bluetooth clock.

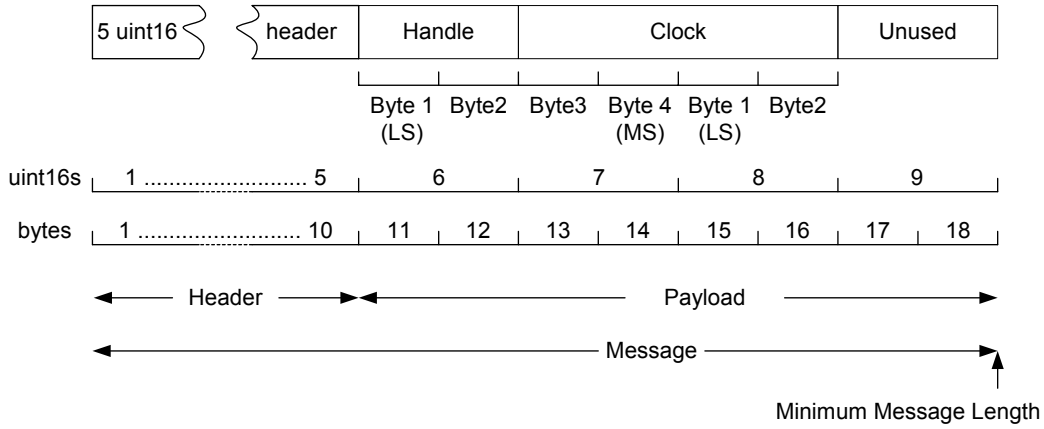



**Figure 6.1: Piconet_Instant Command Structure**

## 6.2 Crypt_Key_Length

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x3008 | Complex | RO | RO |

When encryption is applied to an ACL connection the Link Managers on the local and remote Bluetooth devices negotiate the (effective) encryption key length. This length is not visible over HCI.

The Crypt_Key_Length command allows the host to discover the effective encryption key length negotiated for point-to-point traffic on an established ACL link.

**Note:**

There is no guarantee that encryption of the reported strength is in use at the time of the query. It could be that the effective key length has been negotiated for a link, encryption turned on, and then encryption turned off again. The value obtained via this varid will report the outcome of the negotiation, not the strength of encryption currently being used on the link.

Figure 6.2 shows the data structure used by this command. In the BCCMD GETREQ the "Handle" gives the HCI ACL handle of interest; the `Keylen` and `Unused` fields must be zero. If the command is successful, the returned GETRESP will hold the same `Handle`, and the `Keylen` will carry the link's effective encryption key length, measured in bytes.

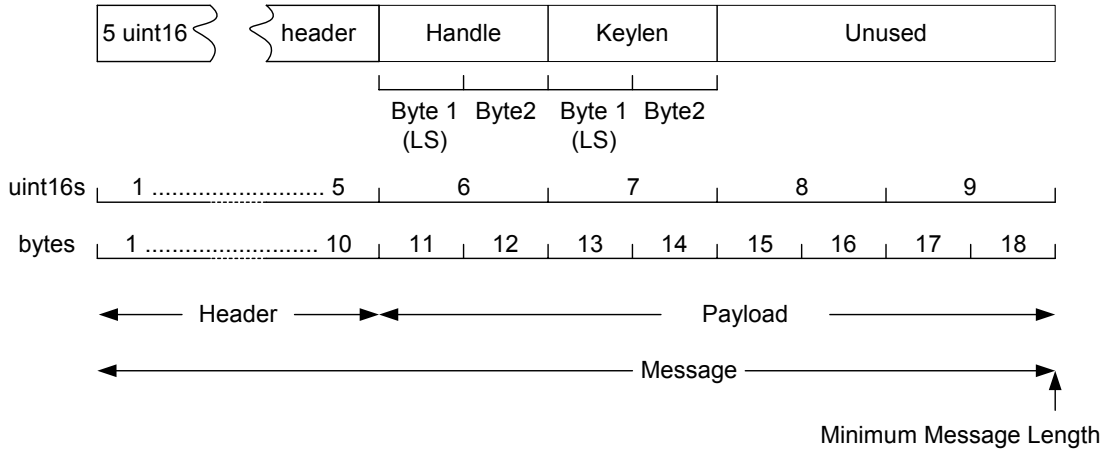



**Figure 6.2: Crypt_Key_Length Command Structure**

## 6.3 BER_Threshold

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x7011 | Complex | RW | RW |

This command arms a monitor on a connection's measured receive BER value. If the measured value for a single received packet exceeds the `Threshold` parameter for the HCI connection identified by `Handle`, then a BER_Trigger HQ message is sent to the host, as described in [HQCMDS]. Once the monitor has fired, it returns to its default un-armed state, i.e. this is a one-shot monitor.

The `Threshold` value is in arbitrary units from `1` to `255`, where `1` is the most sensitive and `255` is the least sensitive. Writing a `Threshold` value of `0` disables a currently armed monitor. (When a monitor fires, it internally sets the link's `Threshold` to zero.)

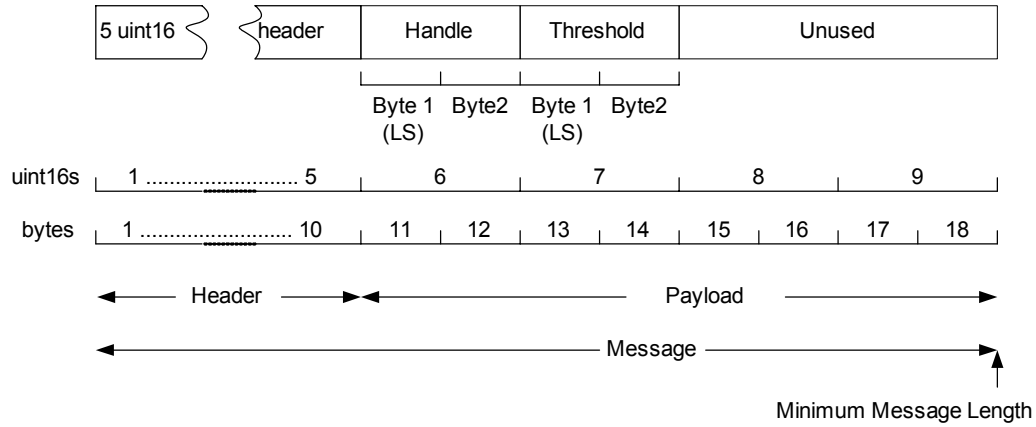



**Figure 6.3: BER_Threshold Command Structure**

The firmware's Link Controller maintains a measure of each connection's received bit error rate (BER); this is used by the HCI Get_Link_Quality command, as described in [BCHCI]. The HCI command uses a *running average* of the bit error rate, whereas the BER_Threshold command uses the bit error rate in each received packet.

## 6.4 RSSI_ACL

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x301d | Complex | RO | RO |

Reading RSSI_ACL retrieves the Received Signal Strength Indication (RSSI) for a given HCI ACL handle. The RSSI value should be interpreted as a signed integer with units of dBm. The RSSI value will always be negative, as the reading saturates at about −20dBm. The value is a rolling average and so will respond more slowly when data is received less frequently, and will freeze at the last value when the link is in Hold mode.

Figure 6.4 shows the data structure used by this command. In the BCCMD GETREQ the Handle gives the HCI ACL handle of interest; the RSSI and Unused fields must be zero. If the command is successful, the returned GETRESP will carry a status of OK, the Handle will hold the supplied handle and the RSSI will hold the link's RSSI as described above. If the command is unsuccessful, the returned GETRESP will carry a status of ERROR, the Handle will hold the supplied handle and the returned RSSI will be set to zero.

This command is supported on BlueCore2 and later chips and appears in 19.x and later builds.

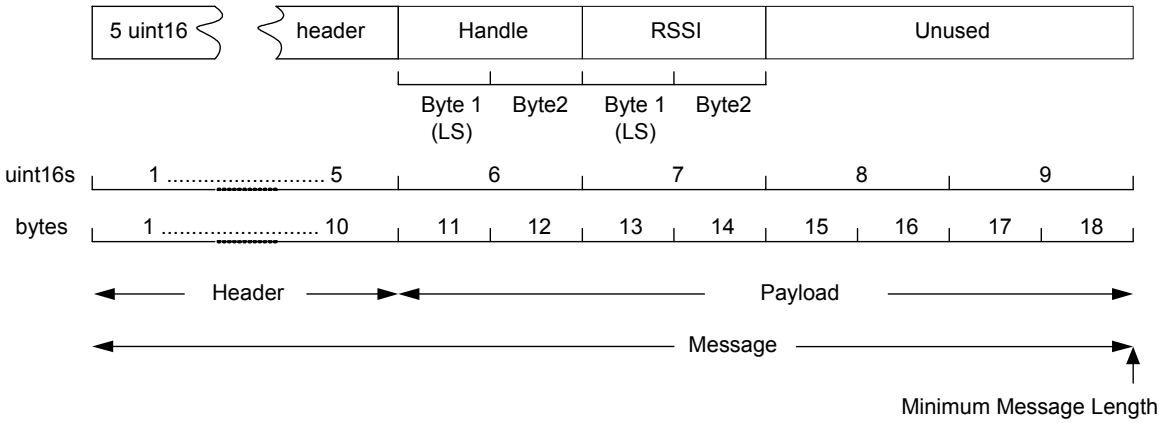



**Figure 6.4: RSSI_ACL Command Structure**

In HCIStackv19.2 and later builds, the firmware measures the RSSI value on a particular ACL link and provides an absolute RSSI indication. In builds before HCIStack1.2v19.2, a much cruder RSSI estimate is available. For more information, refer to [RSSI].

# 7 PIO Commands

BlueCore01b has an 8-bit wide, general-purpose PIO port. Each bit of the port can be programmed to be an input or an output. This section describes commands that deal with the chip's PIO port.

Later chips can provide up to 16 PIO lines, though the chip's PIO pins also provide programmable weak pull-up or pull-down. Some chips can provide stronger pull-up and pull-down on PIOs; the actual number available depends on the chip and its packaging.

## 7.1 PIO

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x681f | uint16 | RW | RW |

This varid writes data out to the port and reads data in from the port. If the chip's PIO port is 8-bits wide, only the low byte of the `uint16` is used. (The varid PIO_Direction_Mask determines whether each of the PIO's pins is used as an input or an output.)

When the chip boots all of the PIO port's pins are set to be inputs, and all pins are weakly pulled down.

If the PIO_Direction_Mask has set a given PIO port bit to be an output then writing to the bit using this varid sets the bit's value, and reading from the bit reads the value last written to the port.

If the PIO_Direction_Mask has set a given PIO port bit to be an input then reading from the bit reads the external value being presented to the pin. On BlueCore01b, writing to an input bit has no effect. On BlueCore2-External and later chips, writing to an input bit sets the direction of weak pull-up (`1`) or pull-down (`0`).

A `1` in the varid's value maps to a high voltage on the corresponding PIO pin.

Some of the PIO port's bits may not be available. See the description of the PIO_Protect_Mask command.

## 7.2 PIO_Direction_Mask

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x681e | uint16 | RW | RW |

This varid accesses a direction register that determines the PIO port's pins' directions.

If the chip's PIO port is 8 bits wide, only the low byte of the `uint16` is used.

Each valid bit of the `uint16` is set to `1` if the corresponding PIO port's bit is to be an output; it is set to `0` if the bit is an input.

See the descriptions of the PIO and PIO_Protect_Mask commands.

## 7.3 PIO_Protect_Mask

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2823 | uint16 | RO | RO |

Although the PIO port is supposed to be general purpose, many of its pins can be dedicated to specific roles. This command reveals the bits allocated to these specific roles. Each 1 in the `uint16` indicates that the bit is not available for use by the PIO and PIO_Direction_Mask commands.

Attempting to write to a protected bit will have no effect. Similarly, trying to set a protected bit's direction will have no effect.

In some cases the chip's internal hardware takes over the pins. For example, if the chip uses an external radio (transmit) power amplifier and (receive) LNA then bits 0 and 1 of the PIO control them.

In some cases the chip's firmware takes over the pins. For example, pins 2 to 5 can be used to support the chip's USB interface.

The varid's value is derived from the PS Key PSKEY_PIO_PROTECT_MASK; this will normally be set up as part of a module's configuration.

In HCIStack1.1v13.x and later builds some bits of the protect mask are derived from other PS Keys. See the description of PSKEY_PIO_PROTECT_MASK.

## 7.4 PIO_Strong_Bias

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6839 | uint16 | RW | RW |

As noted above, the PIO command can be used to apply a weak pull up/down on an input PIO pin.

On BlueCore2-ROM and later chips, with HCIStack1.1v17.x and later builds, PIO pins can be set to have weak or strong pull up/down, controlled by this command. When the chip is reset, PIO pins have weak pull up/down. Writing a `1` to a bit in the command's `uint16` value causes the corresponding PIO pin to apply strong pull up/down.

# 8 Analogue to Digital Converter Commands

BlueCore includes an analogue to digital converter (ADC) that can read voltages from points within the chip. It is also able to read the voltages on BlueCore01b's Test_A and Test_B pins. In front of the ADC is a multiplexor (a rotary switch) that selects the point to be monitored.

In BlueCore2-External and later chips, a voltage can be read from the auxiliary I/O pins AIO0 and AIO1 in exactly the same way as from pins Test_A and Test_B on BlueCore01b, respectively. It is not currently possible to use AIO2 for this purpose.

The ADC is used in real time by the chip's radio hardware; this is the basis of the chip's RSSI/AGC measurement. Consequently, any other access to the ADC has to be co-ordinated with the radio hardware's access. This constraint (partly) explains the rather clumsy ADC interface provided via BCCMD.

To make an ADC reading a value is first written to the ADC varid; this selects the voltage source to be measured (it sets the multiplexor) and it queues a request to the converter to make the reading. A subsequent read of the ADC_Result varid may pick up the result.

The most common use for the ADC is to monitor a battery's voltage, e.g. in a headset. The ADC does not provide readings directly in mV; however, one ADC channel accesses the chip's internal 1V25 reference, against which other ADC channels can be scaled.

The ADC readings are scaled relative to VDD.

## 8.1 ADC

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4829 | uint16 | WO | WO |

Writing to this varid requests the chip's ADC to make a reading on the channel described by the varid's `uint16`. If writing to the varid is successful the result of the conversion should be available after a short delay, typically 10ms, via varid ADC_Result.

Acceptable values for this varid are 0 to 37. The majority of these values select points within the radio, and are not documented here. This is CSR-proprietary information and there is no current intention to publish all of the values.

The only values expected to be used are:

| Value | ADC Channel |
|---|---|
| 1 | Internal 1V25 reference |
| 16 | BlueCore01b pin Test_A; BlueCore2-External pin AIO0 |
| 17 | BlueCore01b pin Test_B; BlueCore2-External pin AIO1 |
| 36 | Chip's internal temperature (change) sensor. BlueCore2-ROM and later only. |

**Table 8.1: ADC Values**

**Note:**

Channels 16 and 17 momentarily change, and then restore, the settings of both of the chip's two AMUXs. See the descriptions of TestA_AMUX and TestB_AMUX commands in section 11.

Writing to this varid will fail if the `uint16` carries an unacceptable value or if the chip's resources are extremely heavily laden.

Only one ADC reading can be made at a time.

## 8.2 ADC_Result

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x3007 | Complex | RO | RO |

Writing to the ADC varid requests the chip to make an ADC reading. The ADC_Result varid may provide the result of the conversion.

The result of reading the ADC is held in the bottom 8 bits of `Result` field. The host should accept this value only if `Valid` is `1` and if `Channel` matches the value written to the previous varid.

Under heavy load conditions the ADC can take several tens of milliseconds to make the reading, hence the presence of the `Valid` flag.

Only one ADC read attempt should be made at a time; the `Channel` field provides a check for this; it should match the value written to the ADC varid.

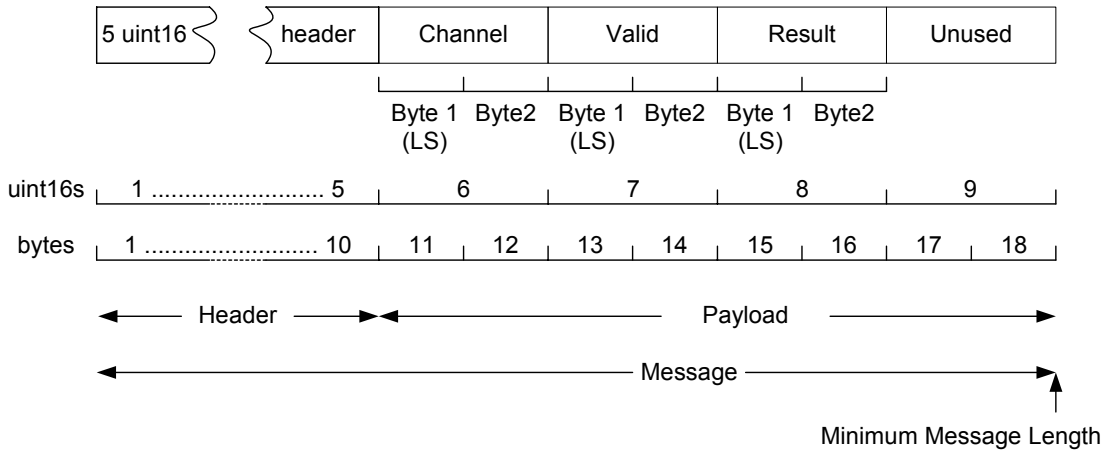



**Figure 8.1: ADC_Result Command Structure**

# 9 Miscellaneous Commands

The following are commands that do not comfortably fit under other headings.

## 9.1 No_Variable

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6000 | Valueless | RW | RW |

This is a dummy varid. Reading and writing this variable always succeeds, but no action is performed and no value is returned.

## 9.2 Rand

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x282a | uint16 | RO | RO |

The chip uses a random number generator to support pairing, authentication and encryption.

This varid obtains a single `uint16` of random data from the generator.

## 9.3 Get_External_Clock_Period

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2843 | uint16 | RO | RO |

From BlueCore3 onwards, an external 32KHz clock can be used to provide a sleep clock. During testing, it is useful to have a way of measuring this clock to make sure it has been properly connected to BlueCore.

The 32KHz clock gets divided down to 1KHz for use by BlueCore and also for this value.

This varid returns the number of 0.25us periods required for 1 (1KHz) clock cycle to occur, so a value of 4000 (0xFA0) is expected.

## 9.4 HostIO_Enable_Debug

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4845 | uint16 | RO | RO |

This command returns extended information when the H4 protocol detects a communication error and resynchronisation is required.

The command is enabled by setting bit 0 to 1, and is disabled by setting bit 0 to 0.

Several extended HCI error events are returned. The first contains the following debug information:

| Format | Error Event |
|---|---|
| uint16 | `hardware_code` (from Bluetooth spec, always `0xFE`) |
| uint16 | `port_offset` |
| uint16 | `index` |
| uint16 | `outdex` |
| uint16 | `tail` |
| uint16 | `rounded_tail` |
| uint16 | `rounded_index` |
| uint16 | `data_length` |

This is followed by more extended HCI error events (with `hardware_code` = `0xFE`) containing `data_length` bytes of data (limited to 128 bytes per event) from `rounded_tail` to `rounded_index` from the buffer used to receive data from the host.

# 10 Persistent Store Commands

This section describes BCCMD commands that access the chip's Persistent Store.

## 10.1 Persistent Store Context

The BlueCore Command (BCCMD) protocol, described in [BCCMD], runs between a command/monitoring entity on a host processor and a command interpreter in a BlueCore chip. In Figure 10.1, the command/monitoring entity is shown as `bcmgr` and the chip's command interpreter as `bccmd`.

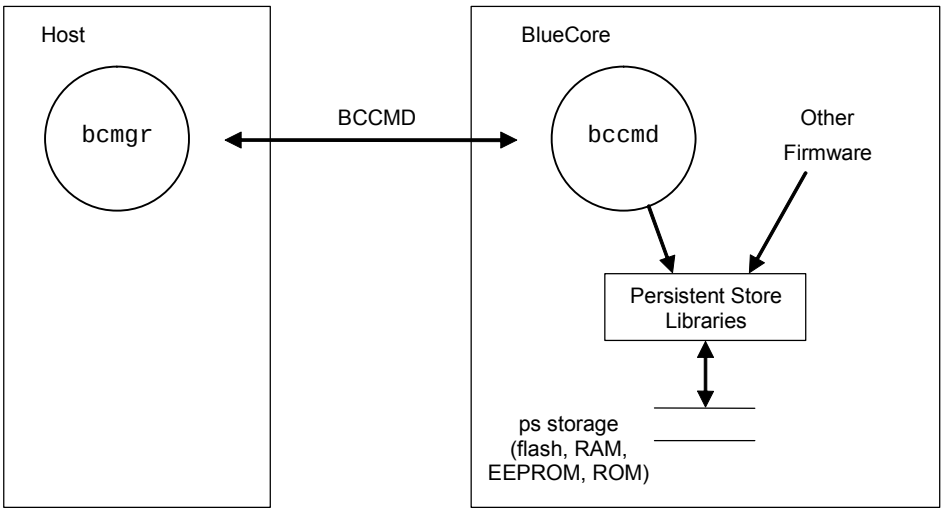



**Figure 10.1: Command/Monitoring Entity and Command Interpreter**

BCCMD commands flow from `bcmgr` to `bccmd`. `bccmd` handles the commands and returns responses to `bcmgr`. The general flow of processing follows the pattern of a remote procedure call (RPC) interface.

A handful of BCCMD commands access the Persistent Store libraries on the chip. These libraries access a simple name/value database that typically uses an area of flash memory as its main data store.

The database contents are primarily used to configure many elements of the firmware, so many subsystems access the Persistent Store libraries.

In most hardware configurations the BlueCore firmware uses a block of flash memory to hold values "persistently", i.e. the values are retained when the power is removed. This holds information that configures the local Bluetooth module for its application. For example, the Persistent Store holds the Bluetooth module's Bluetooth address, its country code, its maximum transmit power, etc.

The store is structured as a name/value database in the manner of Unix's `dbm`. The "name" is a `uint16` (a 16-bit unsigned integer) that identifies a particular piece of information, and the information is held in a sequence of zero or more `uint16`s. The "name" is called a "PS Key" (a Persistent Store key).

For example, if the chip is configured to use its USB interface then the USB "Vendor ID" is held in the single `uint16` value stored under PS Key PSKEY_USB_VENDOR_ID.

There are over 400 PS Keys defined. The majority are of interest only to system testers and integrators.

A recent addition is support to hold values in RAM; this retains information until the power is removed or a cold reset is performed, and is primarily intended for systems that use ROM instead of flash.

Another recent addition, only included in firmware builds intended to be placed in ROM, is support to hold values in an $I^2C$ EEPROM device attached to PIO pins. This can also support storage of a VM application.

### 10.1.1 Persistent Store Libraries: Simple Model

The simplest view of the Persistent Store is a name/value database. The firmware internally calls the following `psset()` function to set the chip's Bluetooth country code to `0x01`:

```
psset( PSKEY_COUNTRYCODE, 0x01 );
```

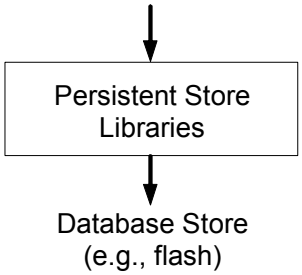



**Figure 10.2: psset() Function**

The country code subsequently can be read by a call in the form:

```
psget(PSKEY_COUNTRYCODE, &cc, sizeof(cc));
```

### 10.1.2 Persistent Store Libraries: Layered Model

A more realistic view of the Persistent Store is as a stack of four layered libraries, as shown in Figure 10.3.

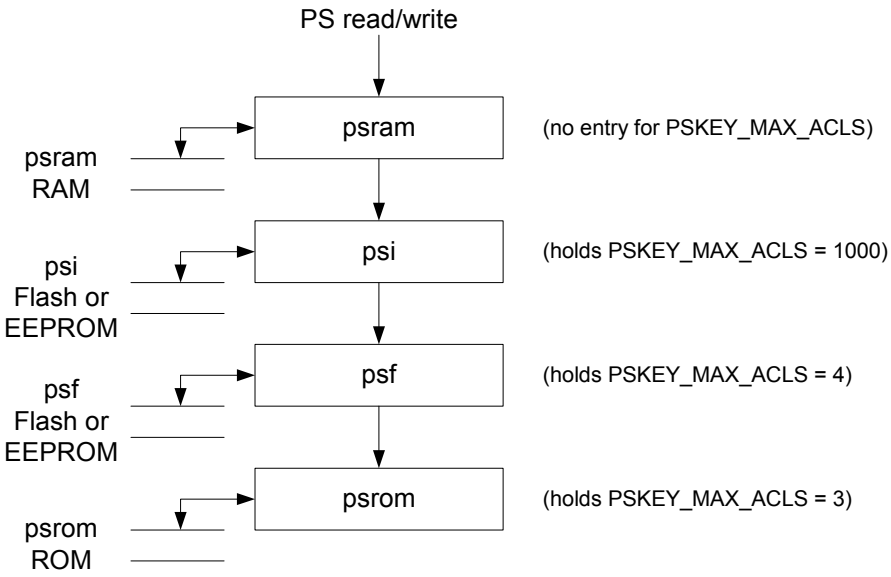



**Figure 10.3: Persistent Store Layered Libraries**

The top library holds values in RAM. The two middle libraries each use a separate area of flash memory or EEPROM to hold values. The bottom library holds values in ROM.

For example, a call to read the value stored under PS Key PSKEY_MAX_ACLS (the maximum number of ACL connections supported) may be given to the top library (`psram`). This searches its RAM store and if it finds it holds data under that PS Key, it returns the corresponding value. In the example shown, `psram` does not hold such a value, so the call continues down to the second library (`psi`) which does hold a value, and so it returns the (absurd) value of `1000`.

The call would have propagated down to the psf library followed by the psrom library in a similar manner, if no value had been found at each stage. The four layers act as a set of masks for each PS Key (values in higher layers hide values in lower layers).

The psrom library is a read-only database; it holds values in ROM. This acts as a backstop for the call. The psrom library is the natural home for default configuration values, e.g. the firmware's default country code lies here. There are default values for approximately 250 of the PS Keys in psrom. In practice a module has very few values in the upper layers to mask the defaults.

### 10.1.3 Implementation Configuration: psi

BlueCore modules can be fitted with different types of non-volatile memory that may be used for the Persistent Store implementation. A particular firmware build may include support for one or more of the following memory types: flash memory, $I^2C$ EEPROMs and ROM. The psi library selects the most appropriate implementation for a particular hardware platform. These are selected in the following order of preference:

- Persistent Store in flash memory. This is selected if the firmware includes flash writing support and a supported flash device can be detected.
- Persistent Store in EEPROM. This is selected if the firmware includes EEPROM support, a supported EEPROM device can be detected, and the device has been prepared using the BCCMD E2_Device command.
- Persistent Store in ROM (read only). A region of ROM can be used to supplement the values provided by psrom. This enables multiple mask ROM images to be generated from a single firmware build, each with its own (different) default PS Key values. The same library that handles Persistent Store in flash memory provides this support.

### 10.1.4 Factory Configuration: psf

When a BlueCore-based Bluetooth module is freshly manufactured there are no values in the psi library. Similarly, there are no values in the psram and psf libraries, so the Persistent Store presents all of its default (psrom) values.

During design and manufacture the module is characterised, e.g. the radio settings. Also, a set of unique values is allocated to each module, e.g. Bluetooth address and crystal trim. These are written to the Persistent Store.

The values initially lie in the psi library, but a BCCMD command ("factory-set") moves these values to the psf store. This now holds the default values for the individual module.

Some PS Keys' values may be changed after manufacture, for example, the module's default Host Controller Interface (HCI) "user friendly name" is held in the Persistent Store. These values are held in the top psi layer, so they override values beneath.

It may be that an end-user sets a crucial PS Key to a value that stops the whole Bluetooth module working. For example, PSKEY_MAX_ACLS could be set to zero. A second BCCMD command ("factory-restore") removes all values from the psi layer, so re-exposing the psf and psrom values. This command could be invoked from an end-user support program.

### 10.1.5 Temporary Configuration: psram

The BlueCore firmware supports operation without writable non-volatile memory by temporarily storing PS Keys in RAM in the psram library. These values are preserved across warm resets, as triggered by the BCCMD Warm_Reset command, but they are lost when the BlueCore is reset by any other mechanism (such as loss of electrical power, a watchdog reset or the BCCMD Cold_Reset command).

The psram store is normally used with BlueCore-ROM parts, particularly where the device has no EEPROM, as described in [ROMBOOT]

The host must download any necessary module-specific settings each time power is applied or a non-warm reset is performed.

The firmware uses many PS Keys' values during initialisation, e.g. to select and configure the host transport, so this occurs before the host can write values to psram. Consequently, a dual-boot approach is required:

- The chip first boots using its default PS Key values. The host then writes the module's configuration to psram, after which it provokes a warm reboot.
- The chip boots a second time, using the configured values.

Values stored in psram consume RAM that would otherwise be available to support Bluetooth links or VM applications, so designers must exercise prudence.

The psram library can be useful in systems that have flash memory or EEPROM, as it holds values across a warm reboot. For example, a system could boot with the UART controlled entirely by a VM application. The VM application could first use a private protocol with a host, then the VM writes to the psram to cause the chip to use H4 after a warm reboot.

## 10.2 PS Access Controls

The Persistent Store has its own set of access controls on top of those applied by the chip's BCCMD command interpreter: each PS Key has a defined set of restrictions. Attempts to reach Persistent Store values for which access is denied are rejected in the same manner as attempts to run inaccessible BCCMD commands.

For example, if a PS Key is READONLY, then an attempt to write to the key is rejected with the Status field set to NO_ACCESS.

## 10.3 PS

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x7003 | Complex | RW | RW |

This command provides the two basic Persistent Store operations: reading and writing values to/from the database.

The command has a relatively complex data structure. As with normal BCCMD commands, the SETREQ, GETREQ and GETRESP all have the same structure:

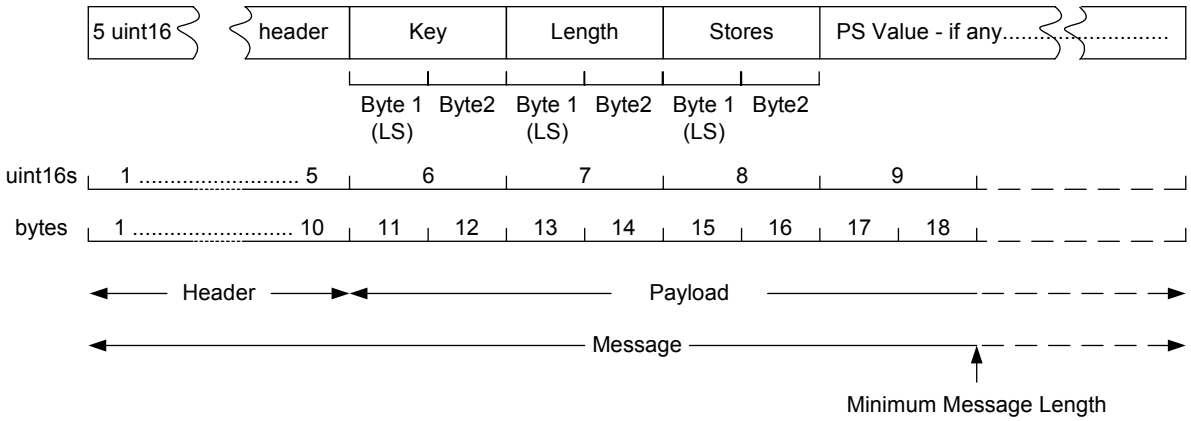



**Figure 10.4: Structure for SETREQ, GETREQ and GETRESP**

As with normal BCCMD packets, the PS packets are a minimum of 18 bytes long and all unused locations must be set to zero.

The Key field carries the PS Key of the database element being accessed.

The Length field describes the length of the PS Value field, counted in uint16s.

- If a value is being written to the PS (SETREQ) then the Length field gives the length of the data being written.
- If a value is being read from the PS then the Length field in the GETREQ gives the maximum length of the data that can be taken from PS in this operation.
- In the corresponding GETRESP the Length field describes the actual length of data retrieved from the store.
- If the size of data in the PS exceeds Length the Status field is set to TOO_BIG, in which case the value returned in the PS Value field is indeterminate.

If Length is zero, then the single uint16 of PS Value must also be zero.

The Stores field controls the searching of the four component stores that make up the full Persistent Store.

If a value is being written to the Persistent Store (SETREQ), then the Stores field controls which store will be modified:

| Persistent Stores Searched | Stores Field Value |
|---|---|
| Default (1) | 0x0000 |
| psram | 0x0008 |
| psi | 0x0001 |
| psf | 0x0002 |

**Table 10.1: SETREQ Persistent Stores Search**

**Note:**

(1) The value is written to the psi store if this is writable, else it is written to the psram store. (The psi store can hold its values in ROM.)

If a value is being read (GETREQ) from the Persistent Store, then the Stores field controls which stores will be examined:

| Persistent Stores Searched | Stores Field Value |
|---|---|
| Default (1) | 0x0000 |
| psram | 0x0008 |
| psi | 0x0001 |
| psf | 0x0002 |
| psrom | 0x0004 |
| psi then psf | 0x0003 |
| psi, psf then psrom | 0x0007 |
| psram then psi | 0x0009 |
| psram, psi then psf | 0x000b |
| psram, psi, psf then psrom | 0x000f |

**Table 10.2: GETREQ Persistent Stores Search**

**Note:**

(1) Equivalent to 0x000f (psram, psi, psf and psrom).

Writing to the chip's (flash or EEPROM) Persistent Store freezes the chip's processor for a significant period. This should be avoided while the chip is supporting a Bluetooth link to avoid losing a block of radio packets.

## 10.4 PS_Next

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x3005 | Complex | RO | RO |

This command allows the host to discover the complete set of store identifiers: the PS Keys themselves (in the Persistent Store).

The command notionally treats the store as an ordered list, sorted in PS Key numerical order. The command takes a single PS Key as an input, and reports the next (numerically higher) PS Key in the store. If the input PS Key is zero then the command reports the first (lowest numbered) PS Key in the store. If the input PS Key is the highest numbered in the store the command reports zero.

Typically the host first uses the zero input to obtain the first key, then it uses that key to find the next key, and so on until the complete list of keys has been reported to the host.

This function deals only with the values of PS Keys themselves; it ignores the data stored with keys.

The command ignores PS Keys' individual access permissions. The host can discover all of the PS Keys in the store, but there's no guarantee it will be able to use the "PS" command to read all keys' values.

The command's structure is shown below. In the GETREQ the `Pskey` field holds the input PS Key and the `Nextkey` field must be zero. In the corresponding GETRESP the `Nextkey` holds the next key in the store.

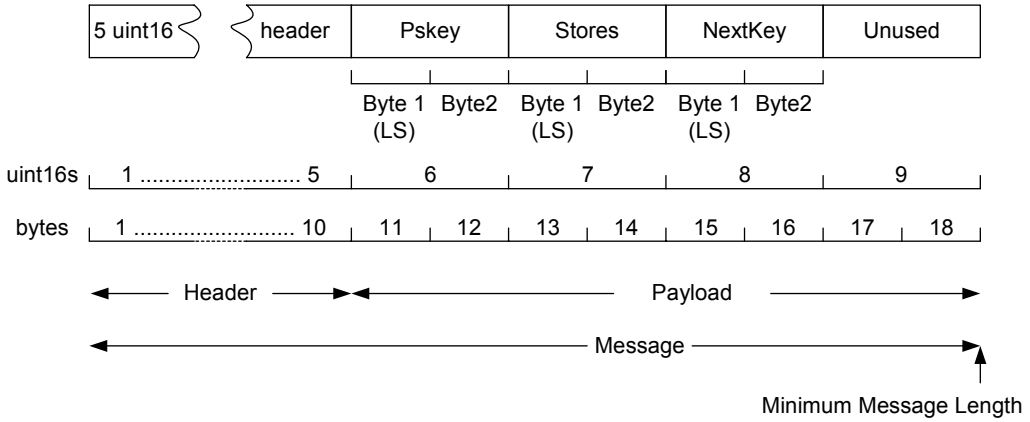



**Figure 10.5: PS_Next Command Structure**

The command takes a `Stores` parameter that allows searching of the four component stores that make up the full Persistent Store. See the description of the PS (GETREQ) command for a list of the acceptable values.

## 10.5 PS_Next_All

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x3013 | Complex | RO | RO |

This command allows the host to use a get-next mechanism to discover the list of PS Keys (the identifiers themselves, not their contents) that either are used directly by the firmware (e.g. PSKEY_BDADDR) or that can reasonably be set for use with/by the firmware (e.g. PSKEY_USR0). This information might be used by a host tool to display or report only pskeys known to the current firmware build.

The command notionally treats the store as an ordered list, sorted in PS Key numerical order. The command takes a single PS Key as an input, and reports the next (numerically higher) PS Key known to the firmware. If the input PS Key is zero then the command reports the first (lowest numbered) PS Key in the store. If the input PS Key is the highest numbered in the store the command reports zero.

Typically the host first uses the zero input to obtain the first key, then it uses that key to find the next key, and so on until the complete list of keys has been reported to the host.

This function deals only with the values of PS Keys themselves; it ignores the data stored with keys.

The command's structure is shown below. In the GETREQ the `Pskey` field holds the input PS Key and the `Nextkey` field must be zero. In the corresponding GETRESP the `Nextkey` holds the next key in the store.

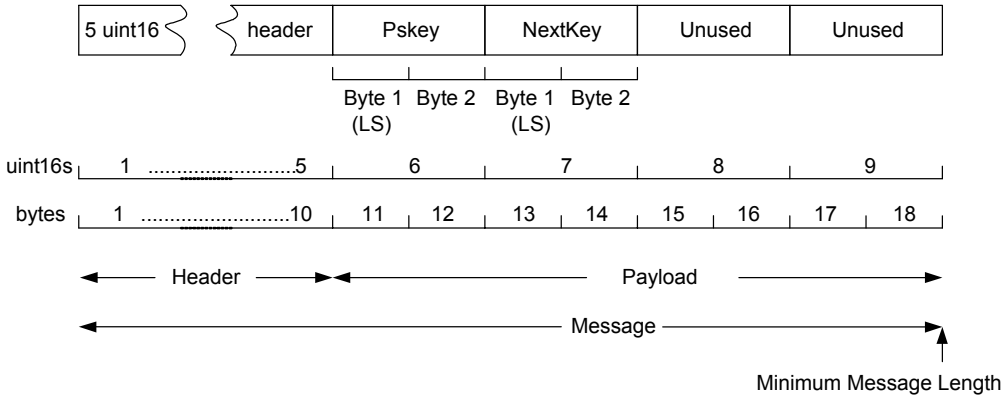



**Figure 10.6: PS_Next_All Command Structure**

## 10.6 PS_Size

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x3006 | Complex | RO | RO |

This command determines how much data is stored under a PS Key.

In the GETREQ, the `Pskey` field holds a PS Key, and the `Length` field must be zero. The `Stores` field can be set to any of the values listed for the PS (GETREQ) command.

The corresponding GETRESP holds the same `Pskey` value, and the `Length` field holds the number of `uint16`s stored under that key.

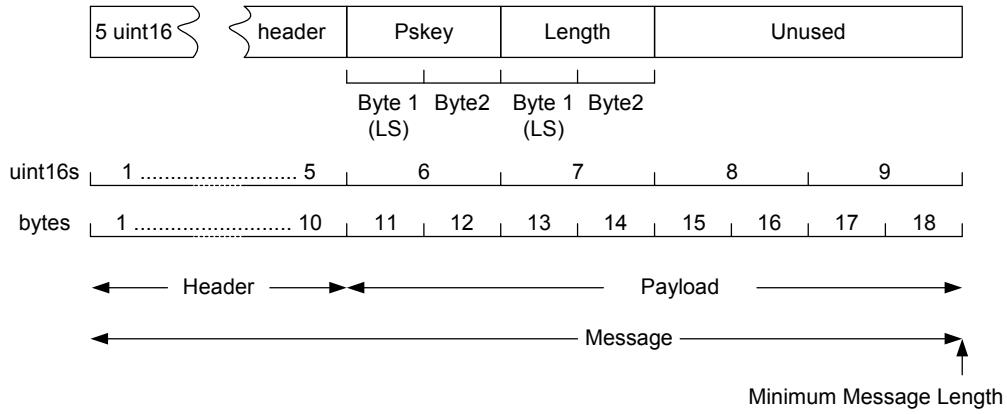



**Figure 10.7: PS_Size Command Structure**

## 10.7 PS_Clr

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4818 | uint16 | RW | WO |

This command is deprecated; it is functionally equivalent to PS_Clr_Stores with a Stores parameter of 0x0000.

## 10.8 PS_Clr_Stores

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x500c | Complex | RW | WO |

This command removes the value stored under the PS Key specified by the Pskey field.

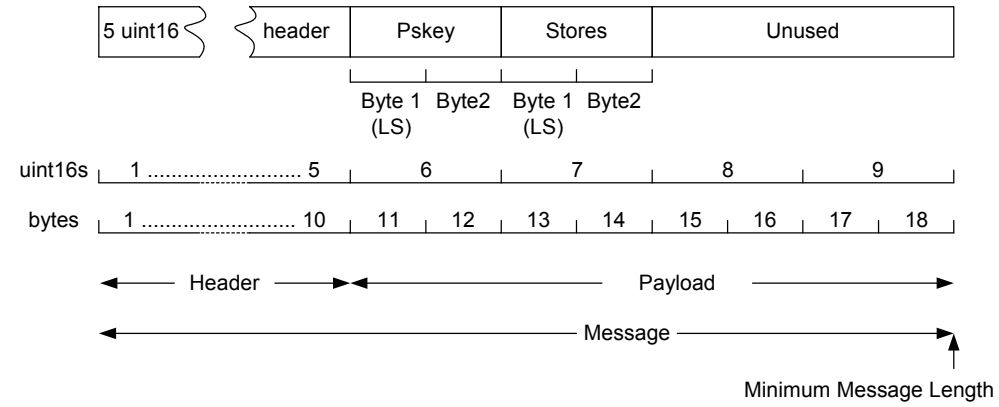



**Figure 10.8: PS_Clr_Stores Command Structure**

The Stores parameter controls from which (of the writable) stores the PS Key values will be removed:

| Persistent Stores Searched | Stores Field Value |
|---|---|
| Default (1) | 0x0000 |
| psram | 0x0008 |
| psi | 0x0001 |
| psf | 0x0002 |
| psi and psf | 0x0003 |
| psram and psi | 0x0009 |
| psram, psi and psf | 0x000b |

**Table 10.3: Clearable Stores Selection**

**Note:**

(1) Equivalent to 0x0009 (psram and psi).

Clearing values from the PS can have subtle effects. The description of the layered model of the PS stores above shows how clearing a value from the psi layer can (re)expose a value in the psf or psrom layer.

Clearing a Persistent Store value from flash or EEPROM implies writing to the flash or EEPROM memory, so this should be avoided while the chip is supporting a radio operation.

## 10.9 PS_Clr_All

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x000b | Valueless | No Access | WO |

This command is deprecated; it is functionally equivalent to PS_Clr_All_Stores with a parameter of 0x0000.

## 10.10 PS_Clr_All_Stores

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x082d | uint16 | No Access | WO |

This command removes all values from the PS stores specified in the uint16.

See the description of PS_Clr_Stores for a list of the acceptable values. However, the default value in the stores parameter affects psram, psi and psf (equivalent to 0x000b).

Clearing a Persistent Store value from flash or EEPROM implies writing to the flash or EEPROM memory, so this should be avoided while the chip is supporting a radio operation.

This command should be used with caution.

## 10.11 PS_Factory_Set

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x000c | Valueless | No Access | WO |

This command moves all values in the psi layer to the psf layer, and so lays the basis for the two factory-restore commands.

## 10.12 PS_Factory_Restore_All

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x400e | Valueless | WO | WO |

This command removes all values from the psi store, so restoring the chip's Persistent Store to the state it was in when the PS_Factory_Set command was invoked.

## 10.13 PS_Factory_Restore

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x400d | Valueless | WO | WO |

This command is identical to PS_Factory_Restore_All except that values in the psi layer that do not obscure values in the psf and/or psrom layers are left in place.

## 10.14 PS_Defrag_Reset

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x400f | Valueless | WO | WO |

This command fills the flash-based PS stores (psi and psf layers), then forces a chip reboot. This means the firmware will recognise the PS flash is full when booting and perform a "defrag", compacting the Persistent Store flash storage.

## 10.15 PS_Memory_Type

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x3012 | Complex | RO | RO |

This command reports the memory (technology) type used to implement any of the persistent stores.

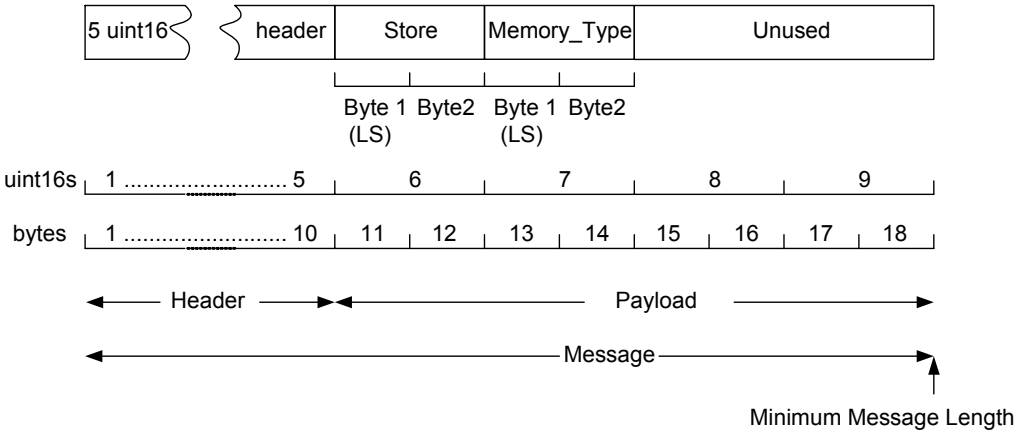



**Figure 10.9: PS_Memory_Type Command Structure**

The `Store` parameter in the GETREQ selects which stores are to be examined. The `Memory_Type` field must be zero in the GETREQ.

| Store Field Value | Persistent Store Examined |
|---|---|
| 0x0000 | Default (1) |
| 0x0008 | psram |
| 0x0001 | psi (2) |
| 0x0002 | psf |

**Table 10.4: Store Field Values**

**Notes:**

(1) `psram` or `psi`, chosen by the firmware according to the stores available.

(2) EEPROM or flash.

The corresponding GETRESP holds the same value in the `Store` field, and the `Memory_Type` field reveals the memory (technology) type used to implement `Store`.

| Memory_Type Value | Meaning |
|---|---|
| 0x0000 | Flash memory |
| 0x0001 | EEPROM |
| 0x0002 | RAM (transient) |
| 0x0003 | ROM (or “read-only” flash memory) (1) |

**Table 10.5: Memory_Type Values**

**Note:**

(1) May be flash memory to which the firmware cannot write.

The following commands are for EEPROM support.

## 10.16 E2_Device

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x300e | Complex | RO | RW |

This command allows the characteristics of an attached I$^2$C EEPROM device to be read or written. It must be used to prepare a new device before it can be used for storage of Persistent Store values or a VM application.

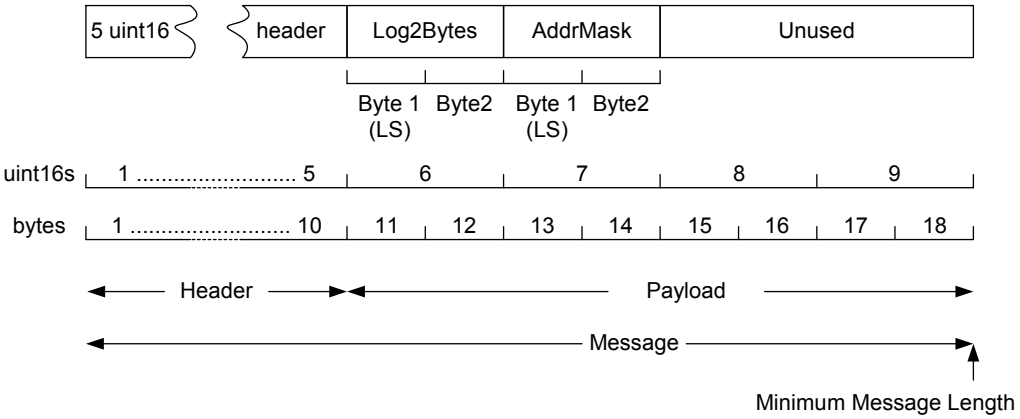



**Figure 10.10: E2_Device Command Structure**

The Log2Bytes parameter specifies the size of the EEPROM ($\log_2$ bytes), i.e. the number of bits required to address any byte of the memory array.

The AddrMask parameter is a bitmap specifying which, if any, of the slave address bits are used to specify high-order bits of the array address.

Table 10.6 provides values that are appropriate for most common I$^2$C EEPROM devices:

| Device Size | Log2Bytes | AddrMask |
|---|---|---|
| 1kbit | 0x0007 | 0x0000 |
| 2kbit | 0x0008 | 0x0000 |
| 4kbit | 0x0009 | 0x0001 |
| 8kbit | 0x000a | 0x0003 |
| 16kbit | 0x000b | 0x0007 |
| 32kbit | 0x000c | 0x0000 |
| 64kbit | 0x000d | 0x0000 |
| 128kbit | 0x000e | 0x0000 |
| 256kbit | 0x000f | 0x0000 |
| 512kbit | 0x0010 | 0x0000 |
| 1Mbit | 0x0011 | 0x0001 |

**Table 10.6: Values for I$^2$C EEPROM Devices**

**Note:**

A few unusual devices (especially some 16-kbit and 512-kbit devices) require different settings for the `AddrMask` field. The correct values can be determined from the device's data sheet.

The BlueCore must be reset after the device characteristics have been written but before any attempt is made to write to Persistent Store.

## 10.17 E2_App_Size

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2833 | uint16 | RO | RW |

This command allows the amount of space reserved for a VM application to be read or changed. It is only applicable if an attached $I^2C$ EEPROM is being used for storage of Persistent Store values or a VM application.

- If the size is being written then the `uint16` of the SETREQ specifies the number of words to reserve. Any running VM application is stopped. If the resize is successful, and the target size is non-zero, then the first word of the application is set to `0xffff`.
- If the size is being read, then the `uint16` of the GETREQ should be zero. In the corresponding GETRESP the parameter indicates the number of words currently reserved. Note that a non-zero value does not imply the presence of a valid VM application.

Resizing the space reserved for a VM application may block the chip's normal operation for a significant period. This should be avoided while the chip is supporting a Bluetooth link to avoid dropping connections. The host interface will not respond until the operation has completed.

## 10.18 E2_App_Data

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x300f | Complex | RO | RW |

This command allows part of a VM application to be read or written. It is only applicable if an attached $I^2C$ EEPROM is being used for storage of Persistent Store values or a VM application.

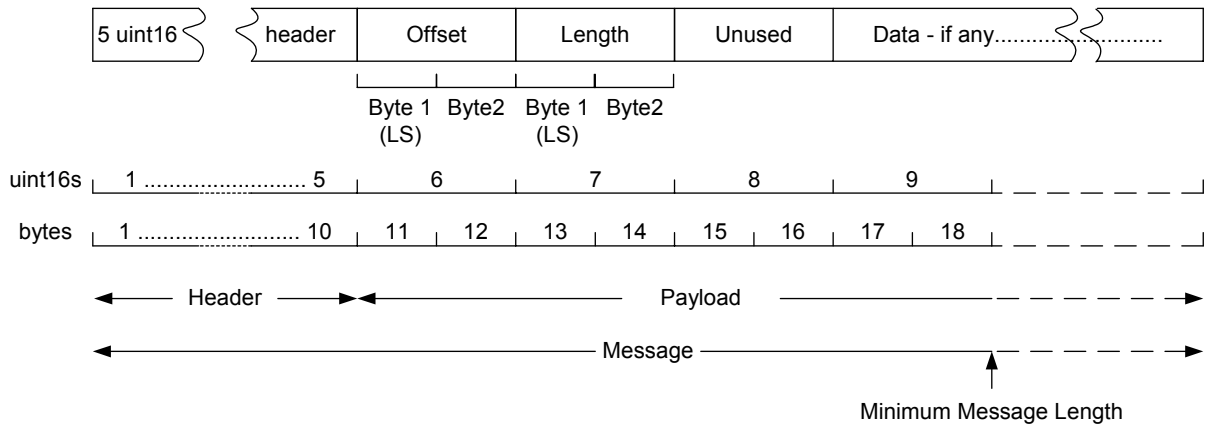



**Figure 10.11: E2_App_Data Command Structure**

The `Offset` parameter specifies the offset from the start of the VM application image at which to start reading or writing.

The `Length` field describes the length of the `Data` field, counted in `uint16`s. In the GETREQ and SETREQ it specifies the maximum number of words to read or write. In the GETRESP it is set to the number of words successfully read or written, which may be less than the requested value if the space reserved for a VM application has been exceeded.

## 10.19 Bootmode

From HCIStack1.2v18.1 the firmware can be booted into one of several “bootmodes”. In each bootmode the firmware sees a different set of PS Keys; these set the firmware’s behaviour. For example, the firmware may first run in a bootmode that makes the firmware behave as a USB keyboard, then the firmware reboots into a second bootmode where it behaves as a conventional USB Bluetooth dongle.

Each bootmode’s view of the Persistent Store is modified by a set of PS Key values that are effectively layered over the normal stores, masking some underlying PS Key values. The set of bootmodes’ masks is, itself, stored in PS Keys. This is more fully explained in [BOOTMODES].

The use of bootmodes is supported by a single BCCMD command that can get or set the current bootmode. This functionality is normally commanded from a VM application, so this command is likely to be used only for testing.

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x683a | uint16 | RW | RW |

Read or write the firmware’s current bootmode.

Setting the firmware into a new bootmode provokes a warm reboot, so it is probable that the host will not receive the command’s GETRESP.

# 11 Test Commands

These commands can cause the chip to enter unusual operating modes, and consequently the normal operation of the firmware may be severely affected. The command set is less well documented than others and is intended for use only by engineers who are aware of the commands' detailed consequences. These commands should not be used by any end-user applications.

## 11.1 TestA_AMUX

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6803 | uint16 | RW | RW |

The BlueCore01b chip holds a pair of analogue multiplexors (rotary switches) that route the voltage from a point within the chip to one of a pair of external test pins. This varid sets the first switch's position. Acceptable values are between 0 and 15.

In BlueCore2 chips, this command controls the first of the chip's four analogue multiplexors, AMUX 0. Acceptable values in this case are between 0 and 31.

The meanings of varid's values are CSR-proprietary information; there is no current intention to publish the values.

See the section on the chip's ADC in section 8.

## 11.2 TestB_AMUX

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6804 | uint16 | RW | RW |

See the description for TestA_AMUX.

On the BlueCore01b chip this sets the position of the second switch's position. Acceptable values are between 0 and 15.

On BlueCore2 chips, this command controls the second of the chip's four analogue multiplexors, AMUX 1. Acceptable values in this case are between 0 and 31. There is currently no way of controlling AMUX 2 or AMUX 3 using the bccmd command set.

## 11.3 Panic_Arg

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6805 | uint16 | RW | RW |

The firmware has a library function, panic(), that is used to force an immediate halt (and probable reboot) of the chip. The function takes a uint16 argument that can indicate why the panic() call was made. (For example, this may indicate that the chip ran out of pool memory.)

The argument to the panic() call is placed in an area of RAM that is not changed as the chip boots, so this may be available after the reboot. This varid gives access to the value.

A panic() call forces a "cold" halt or reboot of the chip, so the value accessed via the Preserve_Valid varid should be zero.

See the description for Preserve_Valid and the section on "Panic and Fault Codes" in [HQCMDS].

## 11.4 Fault_Arg

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x6806 | uint16 | RW | RW |

The firmware has a library function `fault()` that is used to notify that a problem has been detected. The function takes a `uint16` argument that can indicate why the `fault()` call was made. For example, this may indicate that an attempt to generate the chip's synthesiser lookup table failed. Each `fault()` call normally causes an HCI hardware error event to be sent to the host. It may also send an HQ_FAULT message to the host.

The argument to the `fault()` call is placed in a `uint16` in RAM that is not changed as the chip boots, so this may be available after the reboot. This varid gives access to the value.

See the description for Preserve_Valid and the section on "Panic and Fault Codes" in [HQCMDS].

## 11.5 Preserve_Valid

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2807 | uint16 | RO | RO |

When the firmware boots it sets most of its RAM to zero. A small area, the "preserve" space, is not cleared, so data in this area may survive if power is maintained through a reboot.

If the system is shut down cleanly (a "warm" reboot; see the varid in section 3), then the firmware calculates a checksum over this area of RAM as it shuts down, and writes this within the preserve space.

This varid examines the value of the checksum. If the varid is `1`, it indicates that the contents of the "preserve" space apparently survived a reboot.

If the varid is `0`, this does not imply the "preserve" contents are invalid, merely that the checksum was invalid. For example, if a `panic()` call reboots the chip the Preserve_Valid varid will be `0`, but the value of the Panic varid may still indicate why the panic occurred.

## 11.6 Provoke_Panic

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4820 | uint16 | WO | WO |

This command provokes a `panic()` call within the chip. This normally causes an immediate shutdown and reboot of the chip.

The value written to this varid is the value passed to the `panic()` call.

This command is provided for testing only and should not be used as a normal mechanism for restarting the module.

## 11.7 Provoke_Delayed_Panic

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4821 | uint16 | WO | WO |

This command provokes a `delayed_panic()` call within the chip. This normally causes a shutdown and reboot of the chip after a delay.

The value written to this varid is the value passed to the `delayed_panic()` call.

This command is provided for testing only and should not be used as a normal mechanism for restarting the module.

## 11.8 Provoke_Fault

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4822 | uint16 | WO | WO |

This command provokes a `fault()` call within the chip. This will normally cause the chip to attempt to send an HQ_FAULT report back to the host. (It is possible to suppress all fault reports via PS Key settings.)

The value written to this varid is the value passed to the `fault()` call. This is provided for testing only.

## 11.9 Single_Chan

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x482e | uint16 | WO | WO |

This command configures the radio to use only the single hop channel set by the value of the `uint16` and disables the radio's use of whitening. Acceptable values are between `0` and `78` inclusive.

This command has no useful effect when the chip is in radio test mode.

See the Hopping_On command.

## 11.10 Hopping_On

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4011 | Valueless | WO | WO |

This command restores normal radio operation after running the Single_Chan command.

## 11.11 Kill_VM_Application

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4011 | Valueless | WO | WO |

This command terminates any application running in the VM.

The VM application can only be restarted by rebooting the firmware.

See also PS Key PSKEY_VM_DISABLE (0x025d).

## 11.12 Sync_Clock

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x4015 | Valueless | WO | WO |

Most BlueCore designs use a crystal clock for the device's accurate timing. This determines the accuracy of the device's radio tuning and of its Bluetooth clock. The crystal is (commonly) nominally 16 MHz, but manufacturing tolerances mean that there are small frequency differences between individual crystals. The value in PSKEY_ANA_FTRIM (0x01f6) is usually set during module manufacture to "pull" the local device's crystal so that the device's time base, and thus the radio's tuning, is more accurate.

One technique to derive the value for the PS Key is to make an ACL link to a device with a known good clock, to make the local device the connection's slave, and then to measure the rate at which the local (slave) device's Bluetooth clock drifts to maintain the ACL. A good estimate for the crystal trim value can then be derived from the drift rate.

The Sync_Clock and Ana_Ftrim commands partially automate this technique. If the local device is the slave, the Sync_Clock command starts the firmware adjusting the crystal's trim value to minimise the drift rate between the local device's Bluetooth clock and the master's clock.

The Ana_Ftrim command can then retrieve the firmware's best estimate of the preferred crystal trim value. The value is then normally written into PSKEY_ANA_FTRIM.

The derivation of the trim value started by Sync_Clock can take up to five seconds to complete, and there is no direct indication sent to the host to say that the process has finished. However, the host can repeatedly call Ana_Ftrim until the value returned is stable.

## 11.13 Ana_Ftrim

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2837 | uint16 | RO | RO |

See the description for Sync_Clock.

## 11.14 Ana_Ftrim_Readwrite

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2837 | uint16 | RW | RW |

This command can be used to read or write the crystal trim value that is usually set once at boot using PSKEY_ANA_FTRIM. The value read is identical to the value read by Ana_Ftrim. Initially this will be the value configured by PSKEY_ANA_FTRIM.

Writing the value has the same effect as a call to the radio test command Config_XTAL_Ftrim. However, the Radiotest module is not available in all firmware while Ana_Ftrim_Readwrite is.

Setting the value has an immediate effect on the crystal trim and this may be used together with a suitably configured spectrum analyser to identify the correct value to be stored in PSKEY_ANA_FTRIM.

## 11.15 Memory

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x1001 | Complex | No Access | RW |

This command reads or writes a block of RAM within the chip.

This varid uses the following data structure, in which “Base” is the base address of the memory block to be read/written, “Length” is the length of the block counted in `uint16`s and “Data” is the data.

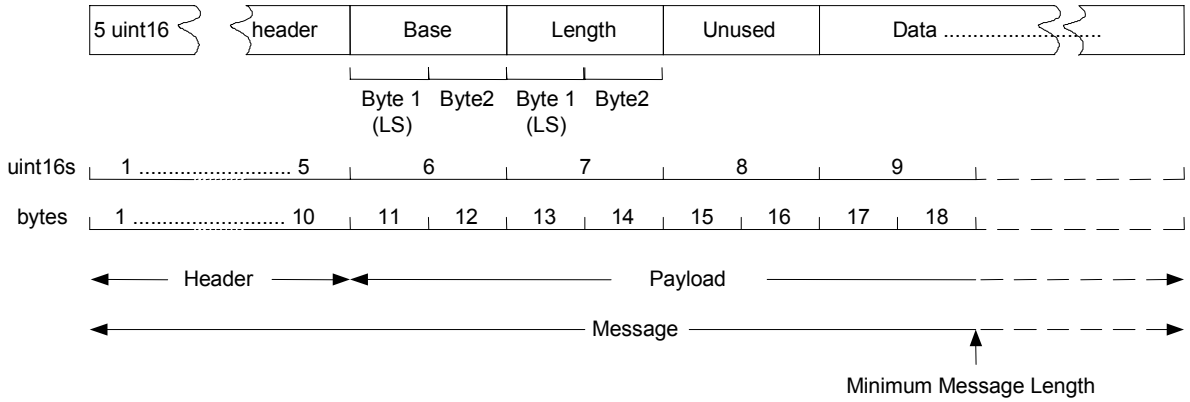



**Figure 11.1: Memory Command Structure**

This command provides lots of opportunities to abuse the chip’s firmware, so it should be used only with great care.

The command is unlikely to be of value without a corresponding memory address map.

## 11.16 Buffer

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x1002 | Complex | No Access | RW |

This command reads or writes a block of data from one of the chip’s MMU byte data buffers.

This varid uses the following data structure, in which “Handle” is the firmware’s internal identifier of the buffer, “Start” is the index into the buffer of the first byte to be transferred, “Length” is the length of the block counted in bytes and “Data” is the data. Unusually for a BlueCore chip, the data is a block of bytes: each is held in the lower half of the `uint16`s of “Data”.

This command must only be used with great care because MMU buffers use a virtual memory system that allocates pages of real RAM on demand (access).

This command is unlikely to be of value without detailed knowledge of the hardware and the firmware’s configuration.

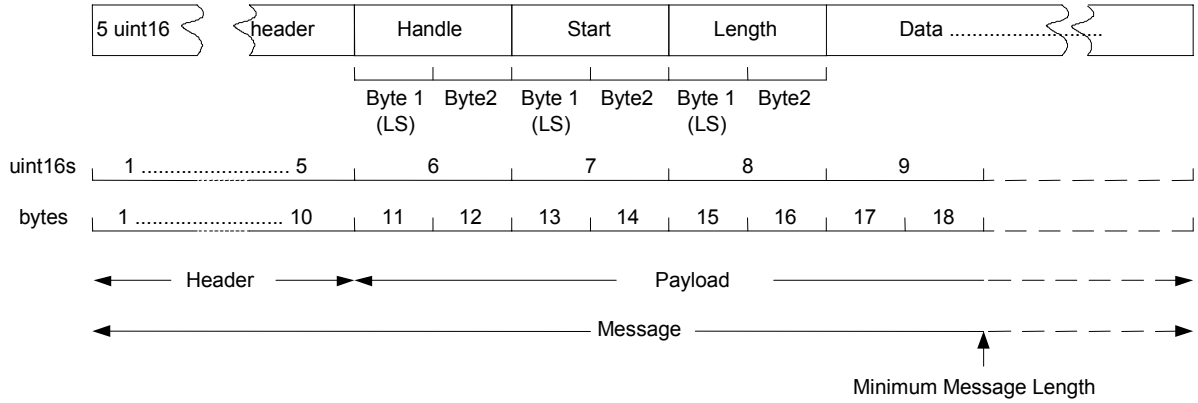



**Figure 11.2: Buffer Command Structure**

## 11.17 Firmware_Check

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2834 | uint16 | RO | RO |

The Firmware_Check and Firmware_Check_Mask commands provide a firmware integrity test.

The firmware is composed of three, largely independent components:

- The Bluetooth “stack”: This provides the core Bluetooth functionality.
- The “loader”: This allows the “stack” to be replaced via a DFU (Device Firmware Upgrade) operation.
- A VM application: This can also be replaced via DFU. This is not always present.

Each of these components comprises code and constant data. In all cases the code and constant data should not change after installation.

From HCIStack1.1v16.4, when one of these components is built, a checksum is calculated over the component’s code and constant data. The checksum value is placed in a special location within the component, and so the checksum is present when the component is loaded into memory.

The Firmware_Check_Mask command returns a bitfield, as indicated in Table 11.1.

| Bit Position | Firmware Component |
|---|---|
| 0 | Bluetooth stack |
| 1 | DFU loader |
| 2 | VM application |

**Table 11.1: Firmware_Check_Mask Command Bit Positions**

A ‘1’ returned in one of these bit positions indicates that the software component is present and that it contains a pre-calculated checksum.

The Firmware_Check command calculates a checksum for each component present, and then compares it with the corresponding pre-calculated checksum. Using the same bit coding, the command returns a ‘1’ for each test passed. All unused bits are zero.

Thus, if the values returned by the two commands are identical, the host can be reasonably confident that the firmware is intact.

Installation of the pre-calculated checksum requires version 1.50 or later of DFU Tools.

The VM application must be produced with version 2.6 or later of BlueLab.

## 11.18 Firmware_Check_Mask

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2835 | uint16 | RO | RO |

See the description for Firmware_Check.

## 11.19 I2C_Transfer

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x100d | Complex | No Access | RW |

This command performs a composite $I^2C$ transfer, as described in [I2C], consisting of a sequence of writes followed by a sequence of reads, optionally separated by a repeated start condition (Sr).

Not all firmware builds contain support for communication with $I^2C$ devices; the support is intended mainly for BlueCore devices that hold their programs in ROM.

This varid uses the following data structure, in which “Address” is the destination $I^2C$ slave address, “Tx” specifies the number of bytes to write, “Rx” specifies the number of bytes to read, “Restart” is non-zero to insert a repeated start condition (Sr) between the writes and reads, and “Data” is the data.

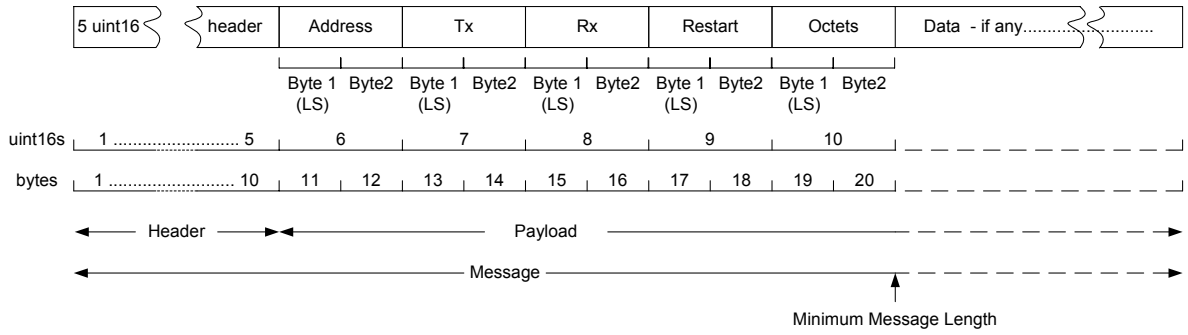



**Figure 11.3: I2C_Transfer Command Structure**

Unusually for a BlueCore chip, the data is a block of bytes: each is held in the lower half of the `uint16`s of “Data”. This field holds the “Tx” bytes to be written followed by space for the “Rx” bytes to be read.

The sequence of operations performed on the $I^2C$ bus is:

1. Start condition (S).
2. Write slave address and direction byte (“Address”).
3. Write “Tx” bytes from “Data”.
4. Repeated start condition (Sr) if “Restart” is non-zero.
5. Write slave address and direction byte (“Address” with bit 0 set) if “Restart” is non-zero.
6. Read “Rx” bytes into “Data”, starting after the “Tx” bytes that were written, acknowledging all but the final byte.
7. Stop condition (P).

If “Tx” is non-zero and “Rx” is zero then the sequence reduces to:

1. Start condition (S).
2. Write slave address and direction byte (“Address”).
3. Write “Tx” bytes from “Data”.
4. Stop condition (P).

Alternatively, if “Tx” is zero and “Rx” is non-zero then the sequence reduces to:

1. Start condition (S).
2. Write slave address and direction byte (“Address” with bit 0 set).
3. Read “Rx” bytes into “Data”, acknowledging all but the final byte.
4. Stop condition (P).

Finally, if both “Tx” and “Rx” are zero then the following minimal sequence is used:

1. Start condition (S).
2. Write slave address and direction byte (“Address” with bit 0 set if “Restart” is non-zero).
3. Stop condition (P).

In the SETREQ or GETREQ the “Octets” field must be zero. This is set in the GETRESP to indicate the total number of bytes transferred and acknowledged, including the slave address byte(s). Note that the final byte read is included in the count, even though it is followed by a NACK. The transfer will be aborted if either the slave address or byte being written is not acknowledged. The stop condition (P) will still be driven to the bus.

**Notes:**

The resulting transfer will not conform to the $I^2C$-bus specification if “Restart” is zero and both “Rx” and “Tx” are non-zero.

“Address” is not shifted before use, i.e. the 7 bit slave address should occupy bits 7 to 1. This allows the R/W bit to be forced to 1 for composite write/read transfers without a repeated start condition (Sr).

This command is primarily intended to exercise $I^2C$ peripherals during hardware and application development.

This command does not control any WP (write protect) line, so may not be sufficient to allow writes to an $I^2C$ EEPROM if hardware write protection is used.

# 12 Radio Test Commands

## 12.1 Radio Test Command Structure

The firmware's set of radio test functions support testing of the chip's radio itself and integration of the chip into Bluetooth modules. Some commands exist mainly to support conformance tests; some are useful to test the performance of product designs.

For historical reasons, the radio test commands are structured differently to other BCCMD functions:

- All radio test functions are accessed through a single BCCMD varid (`0x5004`), then a further discriminator in the command payload selects the radio test command (shown as "TestID" in the command's description).
- All of the functions are intrinsically write-only (despite the RADIOTEST varid being RW). Many commands cause values to be returned to the host via HQ channel messages; described in [HQCMDS].
- All commands take over the chip's radio, terminating normal Bluetooth operation. All of the main radio test commands place the firmware and hardware in special states.
- Once any radio test command is started the only way to restore normal Bluetooth operation is to reboot the chip. The standard HCI Reset command cannot be used for this.
- The host can send a sequence of radio test commands before rebooting the chip.
- All radio test commands return a BCCMD status of BCCMDPDU_STAT_OK if the command was accepted, or BCCMDPDU_STAT_ERROR if rejected.

Figure 12.1 shows the structure of all BCCMD radio test messages.

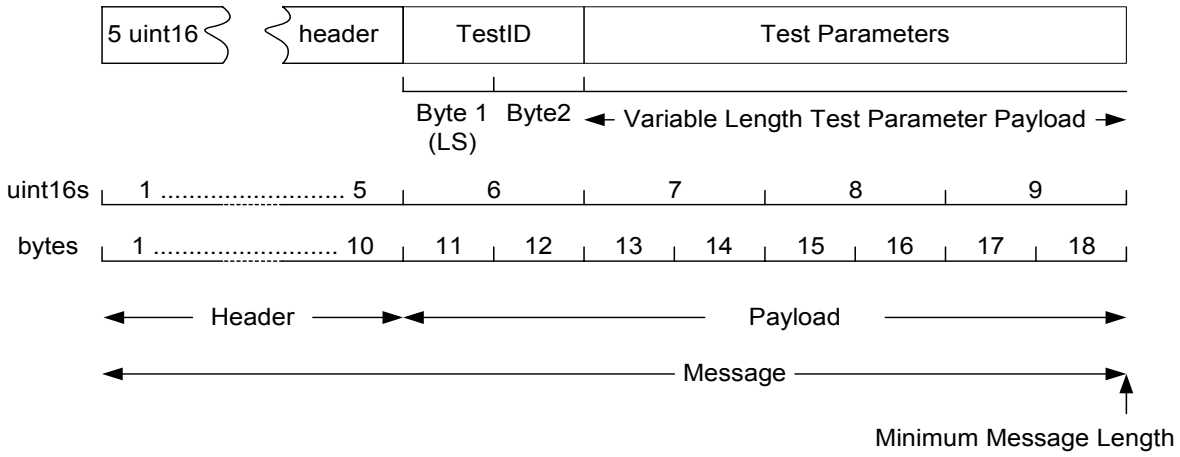



**Figure 12.1: BCCMD Radio Test Message Structure**

Unless noted in its description, each command takes 0, 1, 2 or 3 `uint16` parameters. These are held (in order) in the `Test Parameters` field. Any unused part of the `Test Parameters` field must be set to zero.

### 12.1.1 Command Types

There are two major groups of radio test commands:

- Commands that start a radio test. These are described in most of this document's major sections that follow.
- Commands that configure a (possibly running) radio test. These commands are described in section 25.

### 12.1.2 Command Reports

Many of these commands pass reports back to the host over the Host Query (HQ) channel, described in [HQCMDS]. In most cases, the firmware emits a report at regular intervals. In a few cases a set of messages is sent at each report interval.

The radio tests' HQ data rate is low, so this should normally run quite smoothly. However, the firmware may discard HQ messages to prevent filling the chip's internal buffers.

Almost all of the commands described in this section cause the chip to terminate normal Bluetooth operation and enter special test operating modes. Normal Bluetooth operation can only be restored by rebooting the chip.

These radio test commands are less well documented than others and are intended for use only by engineers who are aware of the commands' detailed consequences.

The firmware contains a set of functions that are used to exercise the radio. These are used to test the chip's radio itself and to help integrate the chip into Bluetooth modules. Some commands exist mainly to support conformance tests and some are useful to test the performance of customers' product designs.

The radio test commands are structured a little differently to other BCCMD functions: all of the radio test functions are accessed through a single varid, then a further discriminator in the varid's payload selects the radio test command.

## 12.2 Radiotest_Interface_Version

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x2826 | uint16 | RO | RO |

This varid obtains the version of the radio test interface. This allows host code to adapt itself to different versions of the radio test interface.

The value of this varid is the revision control system (RCS) version number of the source file that defines the radio test interface in the firmware; this is incremented each time a new version of the file is generated. Value 0 means the information is unavailable.

This command gives the version number of the interface, not of the implementation of commands behind the interface.

This command is not present in builds after HCIStack1.2v18.2.

## 12.3 Radiotest

| Varid | Type | Permissions | Intrinsic Permissions |
|---|---|---|---|
| 0x5004 | Complex | RW | RW |

The Radiotest command takes a complex data type that holds one of a set of commands used to exercise the chip's radio.

Most of these commands cannot be used at the same time as supporting normal Bluetooth operation; most can only be exited by rebooting the chip. (The standard Host Controller Interface (HCI) Reset command does not reboot the chip.)

## 12.4 Simple RF Tests

### 12.4.1 TXStart

(TestID = 1)

This command sets the radio to transmit continuously at frequency `lo_freq`, power `lvl` and modulation frequency `mod_freq`.

The upper byte of `lvl` sets the external amplifier's gain; this only has an effect if an external power amplifier is present. The lower byte of `lvl` sets the internal amplifier's gain.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `lo_freq` | uint16 | `2402` | `2495` | Transmitter frequency in MHz |
| `lvl` | uint16 | `0` | `0xff3f` | Transmitter power amplifier setting |
| `mod_freq` | int16 | `0` | `4096` | 2048 = 500kHz modulation, 4096 = 1MHz, etc. |

Typical parameter values: {`lo_freq` = `2441`, `lvl` = `0xff32`, `freq` = `0`}.

### 12.4.2 RXStart1

(TestID = 2)

This command sets the radio to receive continuously at frequency `lo_freq`, using either high side or low side mix as set by `highside`, and with attenuation `attn`. The `highside` parameter determines whether a lower or upper sideband signal is generated.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `lo_freq` | uint16 | `2402` | `2495` | Receive frequency in MHz |
| `highside` | bool | `0(FALSE)` | `1(TRUE)` | Selects low or high side carrier with reference to the local oscillator |
| `attn` | uint16 | `0` | `15` | Receive attenuation value |

Typical parameter values: {`lo_freq` = `2432`, `highside` = `1`, `attn` = `0`}.

### 12.4.3 RXStart2

(TestID = 3)

This command sets the radio to receive continuously at frequency `lo_freq`, using either high side or low side mix as defined by `highside`, and with attenuation `attn`.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `lo_freq` | uint16 | `0` | `3000` | Receive frequency in MHz |
| `highside` | bool | `FALSE(0)` | `TRUE(1)` | Selects low or high side carrier with reference to the local oscillator |
| `attn` | uint16 | `0` | `15` | Receive attenuation value |

Typical parameter values: {`lo_freq` = `2432`, `highside` = `1`, `attn` = `0`}.

Received Signal Strength Indication (RSSI) readings are sent to the host via the RSSI command described in [HQCMDS], at about 10 readings per second.

## 12.5 Quantitative Transmitter Tests

The TXData and RXData tests use Bluetooth address 0x95fb6bc6967e by default. (The address can be changed using Config_UAP_LAP.) If one device is running an RXData test and another is running a TXData test, then the two devices can interoperate.

### 12.5.1 TXData1

(TestID = 4)

This command sets the radio to transmit a continuous stream of packets at frequency lo_freq and at transmit level lvl. The packets' payload derives from a PRBS9 data sequence.

The upper byte of lvl sets the power control DAC (digital to analogue converter). The lower byte of lvl sets the internal amplifier's gain.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| lo_freq | uint16 | 2402 | 2495 | Transmit frequency in MHz |
| lvl | uint16 | 0 | 0xff3f (basic rate)<br>0xff7f (EDR) | Transmitter power amplifier setting |

Typical parameter values when a basic rate packet type is selected: {lo_freq = 2441, lvl = 0xff32}.

Typical parameter values when an EDR packet type is selected: {lo_freq = 2441, lvl = 0xff69}.

The packet type defaults to DH1, but this can be adjusted by the Config_Packet command.

The packet rate defaults to 800Hz, but this can be adjusted by the radio test Config_Freq command.

An RF (radio frequency) analyser can be used to check the carrier output.

Related tests in [BTTEST]:

- TRM/CA/01/C (Power control)
- TRM/CA/04/C (TX output spectrum – frequency range)
- TRM/CA/05/C (TX output spectrum – 20dB bandwidth)
- TRM/CA/06/C (Adjacent channel power)
- TRM/CA/08/C (Initial carrier frequency tolerance)
- TRC/CA/01/C (Out-of-band spurious emissions)

### 12.5.2 TXData2

(TestID = 5)

This command sets the radio to transmit a continuous stream of packets using the hopping sequence required by Bluetooth country code cc and at transmit level lvl. The packets' payload derives from a PRBS9 data sequence.

The upper byte of lvl sets the power control DAC. The lower byte of lvl sets the internal amplifier's gain.

Setting cc to any other value than 0 is deprecated; it must be set to 0.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| cc | uint16 | 0 | 0 | Country code value |
| lvl | uint16 | 0 | 0xff3f (basic rate) 0xff7f (EDR) | Transmitter power amplifier setting |

Typical parameter values when a basic rate packet type is selected: {cc = 0, lvl = 0xff32}.

Typical parameter values when an EDR packet type is selected: {cc = 0, lvl = 0xff69}.

The packet type defaults to DH1, but this can be adjusted by the Config_Packet command.

The packet rate defaults to 800Hz, but this can be adjusted by the Config_Freq command.

Related tests in [BTTEST]:

- TRM/CA/01/C (output power)
- RM/CA/02/C (power density)

### 12.5.3 TXData3

(TestID = 6)

Enable the transmitter using an output frequency lo_freq with a transmit power level lvl. The payload is "1010101010…".

The upper byte of lvl sets the power control DAC. The lower byte of lvl sets the internal amplifier's gain.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| lo_freq | uint16 | 2402 | 2495 | Transmit frequency in MHz |
| lvl | uint16 | 0 | 0xff3f (basic rate) 0xff7f (EDR) | Transmitter power amplifier setting |

Typical parameter values when a basic rate packet type is selected: {lo_freq = 2441, lvl = 0xff32}.

Typical parameter values when an EDR packet type is selected: {lo_freq = 2441, lvl = 0xff69}.

The packet type defaults to DH1, but this can be adjusted by the Config_Packet command.

The packet rate defaults to 800Hz, but it can be adjusted by the Config_Freq command.

Related tests in [BTTEST]:

- TRM/CA/07/C (modulation characteristic)
- TRM/CA/09/C (carrier frequency drift)

### 12.5.4 TXData4

(TestID = 7)

Enable the transmitter using an output frequency lo_freq with a transmit power level lvl. The payload of "1111000011110000…." is sent using DH1 packets.

The upper byte of lvl sets the power control DAC. The lower byte of lvl sets the internal amplifier's gain.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| lo_freq | uint16 | 2402 | 2495 | Transmit frequency in MHz |
| lvl | uint16 | 0 | 0xff3f | Transmitter power amplifier setting |

Typical parameter values: {lo_freq = 2441, lvl = 0xff32}.

The packet type defaults to DH1, but this can be adjusted by the Config_Packet command.

The packet rate defaults to 800Hz, but this can be adjusted by the Config_Freq command.

Related test in [BTTEST]:

- TRM/CA/07/C (modulation characteristic)

## 12.6 Quantitative Receiver Tests

The TXData and RXData tests use Bluetooth address 0x95fb6bc6967e by default. If one device is running an RXData test and another is running a TXData test, then the two devices can interoperate.

### 12.6.1 RXData1

(TestID = 8)

This command enables the receiver at frequency lo_freq with receive attenuation attn. If highside is set to FALSE, then low side modulation is used, otherwise high side modulation is used.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| lo_freq | uint16 | 0 | 3000 | Receive frequency in MHz |
| highside | bool | FALSE(0) | TRUE(1) | Selects low or high side carrier with regard to the local oscillator |
| attn | uint16 | 0 | 15 | Receive attenuation |

Typical parameter values: {lo_freq = 2441, highside = 1, attn = 0}.

The radio is configured to receive any ACL packet type. The received data-medium rate (DM) packets, which use 2/3 FEC (forward error correction), are used to derive data:

- Number of received packets
- Number of payloads with correctable errors

Analysis of these two numbers allows calculation of the bit error rate (BER) provided the probability of more than one error per packet is low.

An analysis report is sent back to the host via an RX_Pkt_Stats command, described in [HQCMDS], at about one report per second.

### 12.6.2 RXData2

(TestID = 9)

This command enables the receiver using the hopping sequence implied by country code `cc` and receive attenuation `attn`. If `highside` is FALSE, then low side modulation is used, otherwise high side modulation is used.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| cc | uint16 | 0 | 0 | Setting cc to any other value than 0 is deprecated; it must be set to 0. |
| highside | bool | FALSE(0) | TRUE(1) | Selects low or high side carrier with regard to the local oscillator |
| attn | int16 | 0 | 15 | Variable receive attenuation |

Typical parameter values: {cc = 0, highside = 1, attn = 0}.

The radio is configured to receive any ACL packet type. The received DM packets, which use 2/3 FEC, are used to derive data:

- Number of received packets
- Number of payloads with correctable errors

Analysis of these two numbers allows calculation of the BER provided the probability of more than one error per packet is low.

An analysis report is sent back to the host via an RX_Pkt_Stats command, described in [HQCMDS], at about one report per second.

### 12.6.3 BER1

(TestID = 19)

This command enables the receiver at frequency `lo_freq` with receive attenuation `attn`. If `highside` is set to FALSE, then low side modulation is used, otherwise high side modulation is used.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| lo_freq | uint16 | 2402 | 2495 | Receive frequency in MHz |
| highside | bool | FALSE(0) | TRUE(1) | Selects low or high side carrier with regard to the local oscillator |
| attn | uint16 | 0 | 15 | Receive attenuation |

Typical parameter values: {lo_freq = 2441, highside = 1, attn = 0}.

The radio is configured to receive any ACL packet type. Each second a set of nine BIT_ERR reports is passed back to the host. (The BIT_ERR report format is described in [HQCMDS].)

The report period can be changed from its one second default with the Config_Freq command.

Related tests in [BTTEST]:

- RCV/CA/01/C (Sensitivity – single slot packets)
- RCV/CA/02/C (Sensitivity – multi slot packets)
- RCV/CA/03/C (C/I performance)
- RCV/CA/04/C (Blocking performance)

- RCV/CA/05/C (Intermodulation performance)
- RCV/CA/06/C (Maximum input level). (i.e. all RX tests)

### 12.6.4 BER2

(TestID = 20)

This command enables the receiver using the hopping sequence implied by country code `cc` and receive attenuation `attn`. If `highside` is set to FALSE, then low side modulation is used, otherwise high side modulation is used.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `cc` | uint16 | `0` | `0` | Setting `cc` to any other value than `0` is deprecated; it must be set to `0`. |
| `highside` | bool | `FALSE(0)` | `TRUE(1)` | Selects low or high side carrier with regard to the local oscillator |
| `attn` | uint16 | `0` | `15` | Variable receive attenuation |

Typical parameter values: {`cc` = `0`, `highside` = `1`, `attn` = `0`}.

The radio is configured to receive any ACL packet type. The received DM packets, which use 2/3 FEC, are used to derive data returned to the host. Each second a set of 9 BIT_ERR reports is passed back to the host. (The BIT_ERR report format is described in [HQCMDS].)

The report period can be changed from its one second default with the Config_Freq command.

## 12.7 Loopback Tests

This section outlines the available loopback tests. However, the use of the radio tests described in this section has been superseded by the introduction of commercially available Bluetooth test equipment capable of exercising the firmware's Bluetooth Test Mode (defined in [BT2.0]). CSR considers the use of such external test equipment to be a better approach to loopback testing. Thus, the use of radio tests described in this section is deprecated.

### 12.7.1 Loop_Back

(TestID = 21)

This command receives data at frequency `lo_freq`, then retransmits it at frequency `lo_freq` at output power `lvl` some time later.

The upper byte of `lvl` sets the power control DAC. The lower byte of `lvl` sets the internal amplifier's gain.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `lo_freq` | uint16 | `2402` | `2495` | TX/RX Frequency in MHz |
| `lvl` | uint16 | `0` | `0xff3f` | Transmitter power amplifier setting |

Typical parameter values: {`lo_freq` = `2441`, `lvl` = `0xff32`}.

The Config_Freq command should normally be used to set the packet repetition rate and delay to values of the order of 12.5ms and 1.875ms respectively; the default values will normally be too small for this test to run.

The radio's highside and attenuation control parameters are set to zero for this test.

See the description below of the RX_Loop_Back command.

### 12.7.2 RX_Loop_Back

(TestID = 25)

This command transmits PRBS9 packets on frequency `lo_freq` at level `lvl` and listens for packets in the next slot but one. Highside reception is off and attenuation is zero. The command defaults to using single-slot packets.

The upper byte of `lvl` sets the power control DAC. The lower byte of `lvl` sets the internal amplifier's gain.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `lo_freq` | uint16 | `2402` | `2495` | TX/RX Frequency in MHz |
| `lvl` | uint16 | `0` | `0xff3f` | Transmitter power amplifier setting |

Typical parameter values include: {`lo_freq` = `2441`, `lvl` = `0xff32`}.

An analysis report is sent back to the host via an RX_Pkt_Stats command, described in [HQCMDS], at about one report per second.

This is intended to be used with a second radio running the Loop_Back test command.

### 12.7.3 BER_Loop_Back

(TestID = 26)

This command transmits PRBS9 packets on frequency `lo_freq` at level `lvl` and listens for packets in the next slot but one. Highside reception is off and attenuation is zero. The command defaults to using single-slot packets.

The upper byte of `lvl` sets the external amplifier's gain; this only has an effect if an external power amplifier is present. The lower byte of `lvl` sets the internal amplifier's gain.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `lo_freq` | uint16 | `0` | `3000` | TX/RX Frequency in MHz |
| `lvl` | uint16 | `0` | `0xff3f` | Transmitter power amplifier setting |

Typical parameter values: {`lo_freq` = `2432`, `lvl` = `0xff32`}.

Each second a set of nine BIT_ERR reports is passed back to the host. (The BIT_ERR report format is described in [HQCMDS].)

The report period can be changed from its one second default with the Config_Freq command.

This is intended to be used with a second radio running the Loop_Back test command.

### 12.7.4 PCM_LB

(TestID = 11)

This command enables PCM Loopback test mode in which data read from the PCM input port is written back out again via the pulse code modulation (PCM) output port after a delay.

The `pcm_mode` parameter can take the following values:

- `0` Slave in normal 4-wire configuration
- `1` Master in normal 4-wire configuration
- `2` Master in Manchester encoded, 2-wire configuration

For a 'normal' loopback configuration `pcm_mode` should be `1`.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| pcm_mode | uint16 | 0 | 2 | Selects PCM Mode |

Typical parameter value: {pcm_mode = 1}.

### 12.7.5 PCM_Ext_LB

(TestID = 28)

This command enables PCM external loopback test mode in which a block of random data is written to the PCM output port and is read back again on the PCM input port. A check is made that the data matches. External wiring must be provided between the corresponding pins.

The pcm_mode parameter can take the following values:

- 0 Slave in normal four-wire configuration
- 1 Master in normal four-wire configuration
- 2 Master in Manchester encoded, two-wire configuration

For a 'normal' loopback configuration pcm_mode should be 1.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| pcm_mode | uint16 | 0 | 2 | Selects PCM Mode |

Typical parameter value: {pcm_mode = 1}.

The BCCMD Radiotest command returns BCCMDPDU_STAT_OK if the data was looped back successfully, otherwise it returns BCCMDPDU_STAT_ERROR.

### 12.7.6 PCM_Timing_In

(TestID = 43)

This command is similar to PCM_Ext_LB: this makes a series of writes to a PIO pin and tests that the result is readable on one of the PCM pins used for timing in slave mode, PCM_SYNC and PCM_CLK. A check is made that the data matches. External wiring must be provided between the corresponding pins.

The pio_out parameter can take the values 0 to 7, corresponding to the PIO pin to be used for output.

The pcm_in parameter can take the following values:

- 0 Test input on PCM_SYNC
- 1 Test input on PCM_CLK

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| pio_out | uint16 | 0 | 7 | PIO pin used for writing |
| pcm_in | uint16 | 0 | 1 | PCM pin used for reading |

The BCCMD Radiotest command returns BCCMDPDU_STAT_OK if the data was looped back successfully, otherwise it returns BCCMDPDU_STAT_ERROR.

### 12.7.7 CTS_RTS_LB

(TestID = 42)

This command performs a series of writes to the UART CTS (clear to send) line and attempts to read back the value written on the UART RTS (ready to send) line. External wiring must be provided between the corresponding pins.

The BCCMD Radiotest command returns BCCMDPDU_STAT_OK if the data was looped back successfully, otherwise it returns BCCMDPDU_STAT_ERROR.

## 12.8 Miscellaneous Radio Tests

### 12.8.1 Pause

(TestID = 0)

This command halts the current test, stopping any radio activity. A new Radiotest command may be submitted subsequently.

### 12.8.2 Deep_Sleep

(TestID = 10)

The chip's Deep Sleep mode halts its fast (crystal) clock, halts its CPU and freezes much of its support circuitry. The chip runs only a slow, low accuracy clock. In this state its power consumption is much lower than when it runs normally. While in this state the processor will occasionally wake to kick the chip's watchdog reset circuit.

This command places the chip in its Deep Sleep state half a second after sending the BCCMD message acknowledging reception of the command to the host.

The command is normally used to measure the chip's power consumption.

The low power state can be exited by sending data to the chip.

### 12.8.3 Synth_Settle

(TestID = 12)

This command monitors the settling time of the chip's synthesiser. The synthesiser is set to the Bluetooth channel `chan1` and allowed to settle. The synthesiser is then switched to channel `chan2` and a sequence of 10 reports of the synthesiser (LO_TUNE) error voltage over the next 200 microseconds is sent to the host via SYNTH_SETTLE messages. (Synth_Settle is described in [HQCMDS].)

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `chan1` | uint16 | `0` | 78 | Bluetooth channel |
| `chan2` | uint16 | `0` | 78 | Preferably a different channel than `chan1` |

Typical parameter values: {`chan1` = `0`, `chan2` = 78}.

### 12.8.4 Settle_Repeat

(TestID = 38)

This command continuously retunes the synthesiser between two channels. RF output at the antenna is enabled and can be used to monitor the settling behaviour. The cycle sequence is:

1. The external RF output is enabled.
2. The synthesiser is tuned to `chan1`.
3. After 900µs, the external RF output is disabled.
4. Wait for 100µs.
5. The external RF output is enabled.
6. The synthesiser is tuned to `chan2`.
7. After 900µs, the external RF output is disabled.
8. Wait for 100µs.

The cycle continues indefinitely. The external RF signal on the PIO line (enable external PA) can be used for triggering a spectrum analyser.

After the SETRESP (success) message has been returned over BCCMD, a hardware reset of the chip is required before any other command can be executed.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `chan1` | uint16 | `0` | 78 | Bluetooth channel |
| `chan2` | uint16 | `0` | 78 | Preferably a different channel than `chan1` |

Typical parameter values: {`chan1` = `0`, `chan2` = 78}.

### 12.8.5 VCOTrim

(TestID = 14)

This command sweeps the voltage controller oscillator (VCO) across all of the channels used to support country code `cc` and returns an array of the VCO trim value for each channel.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `cc` | uint16 | `0` | 3 | Setting `cc` to any other value than `0` is deprecated; it must be set to `0`. |

Typical parameter value: {`cc` = `0`}.

The command returns 79 values if `cc` is zero; otherwise 23 values are returned.

The array of trim values is returned by a set of LUT_ENTRY messages. (The LUT_ENTRY message is described in [HQCMDS].).

### 12.8.6 RF_IQ_Match

(TestID = 15)

This command measures the RF IQ (in-phase and quadrature) match by injecting a test signal, sweeping the IQ trim and measuring RSSI for on-channel and image signals.

The command generates an array of IQ measurements against IQ trim, returned as a set of IQ_MATCH messages, described in [HQCMDS].

**Note:**

This command has been removed from HCIStack1.2v18.1 and later builds.

### 12.8.7 IF_IQ_Match

(TestID = 16)

This command measures the IF (Intermediate Frequency) IQ match by injecting a test signal, sweeping the IQ trim and measuring RSSI for on-channel and image signals.

The command generates an array of IQ measurements against IQ trim, returned as a set of IQ_MATCH messages, described in [HQCMDS].

**Note:**

This command has been removed from HCIStack1.2v18.1 and later builds.

### 12.8.8 Build_LUT

(TestID = 17)

This command builds the radio's channel LO_TRIM frequency lookup table the returns it to the host with a sequence of LUT_ENTRY commands, described in [HQCMDS].

The upper byte of each trim table location holds the transmitter trim and the lower byte holds the receiver trim.

### 12.8.9 Read_LUT

(TestID = 18)

This command reports the radio's channel LO_TRIM frequency lookup table to the host with a sequence of LUT_ENTRY commands, described in [HQCMDS].

The upper byte of each trim table location holds the transmitter trim and the lower byte holds the receiver trim.

### 12.8.10 Radio_Status

(TestID = 34)

This command returns the values from a set of radio control registers via an HQ RADIO_STATUS message, described in [HQCMDS].

This command is now obsolete. The Radio_Status_Array command should be used instead.

### 12.8.11 Radio_Status_Array

(TestID = 39)

This command returns the values from a set of radio control registers via an HQ RADIO_STATUS_ARRAY message, described in [HQCMDS]. This is useful for diagnosing RF related problems with a module.

### 12.8.12 PCM_Tone

(TestID = 41)

This command outputs a sine wave to the PCM port. On BlueCore2-External, if the Persistent Store value PSKEY_HOSTIO_MAP_SCO_CODEC is set to TRUE, this uses the CODEC instead of the PCM port.

The parameter freq controls the frequency of the sine wave. The value 0 corresponds to 250Hz; each increment of 1 doubles the frequency. Hence, 1 corresponds to 500Hz, 2 to 1kHz and so on. No bounds checking is performed on the parameter, but large values will not be useful owing to hardware limitations.

The parameter ampl controls the amplitude of the sine wave. 8 is the full volume; each decrement of 1 reduces the amplitude by a factor of 2. 0 is valid and causes the hardware to be activated with constant audio data.

The parameter dc is a constant offset to be added to the audio data.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| freq | uint16 | 0 | 65535 | |
| ampl | uint16 | 0 | 8 | |
| dc | uint16 | 0 | 65535 | |

Typical parameter values: {freq = 2, ampl = 6, dc = 0}.

## 12.9 Radio Test Configuration Commands

All of these commands configure a running radio test, started by one of the commands in the previous major sections.

### 12.9.1 Config_Freq

(TestID = 22)

This command sets configuration data used by radio receive and transmit tests.

txrx_freq is the interval in microseconds between transmissions and expected receptions, i.e. the time between transmit packets. This defaults to 1.25ms (two slots).

report_freq is the interval in seconds between reports sent to the host for the sets of RXData and BER tests. It does not apply to the RXSTART2 RSSI reports. This defaults to one second.

If any parameter is zero the corresponding current value is not altered.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| txrx_freq | uint16 | 625 | 65535 | |
| report_freq | uint16 | 1 | 65535 | |

Typical parameter values: {txrx_freq = 1250, report_freq = 1}.

### 12.9.2 Config_Packet

(TestID = 23)

This command sets the packet type and payload size for the packets transmitted during the set of TXData tests. This has no effect on RXData or Loopback tests.

pkt_type is the standard Bluetooth packet type, although only packets for voice or data should be used. Any other value restores the default. The default is DH1.

pkt_size is the length of the data excluding any FEC or CRC. If it is zero, this sets the default: 27 bytes of data in a DH1 packet.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| pkt_type | uint16 | 0 | 31 | |
| pkt_size | uint16 | 0 | 1023 | Maximum limited by packet type |

Typical parameter values: {pkt_type = 4, pkt_size = 27}.

The Bluetooth packet types and the values entered to set them are shown in Table 12.1.

| Basic Data Rate | | Enhanced Data Rate | | | |
|---|---|---|---|---|---|
| **pkt_type** | **Value** | **pkt_type** | **Value** | **pkt_type** | **Value** |
| DH1 | 4 | 2-DH1 | 20 | 3-DH1 | 24 |
| DH3 | 11 | 2-DH3 | 26 | 3-DH3 | 27 |
| DH5 | 15 | 2-DH5 | 30 | 3-DH5 | 31 |
| DM1 | 3 | | | | |
| DM3 | 10 | | | | |
| DM5 | 14 | | | | |
| EV3 | 7 | 2-EV3 | 22 | 3-EV3 | 23 |
| EV4 | 12 | | | | |
| EV5 | 13 | 2-EV5 | 28 | 3-EV5 | 29 |

**Table 12.1: Config_Packet Command Packet Types**

### 12.9.3 Config_Bit_Err

(TestID = 24)

This command sets the count target for the Bit Error Rate tests. bits_count gives the total number of bits to receive before the counter is reset to zero. If reset is non-zero, the counters are immediately reset.

This command is unusual in that it has a parameter of type uint32. This can be thought of as two uint16s, with the more significant word of the uint32 in the first uint16.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| bits_count | uint32 | 0 | 4294967295 | Target of total counters |
| reset | bool | FALSE(0) | TRUE(1) | Resets the counters immediately |

Typical parameter values: {bits_count = 1600000, reset = 0}.

### 12.9.4 Config_TX_IF

(TestID = 27)

This command configures the IF used in the transmitter tests. `Offset_half_mhz` sets the intermediate frequency transmitter offset in units of 0.5MHz.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `offset_half_mhz` | int16 | `-5` | 5 | -2.5 MHz to 2.5 MHz |

Typical parameter value: {`offset_half_mhz` = `0`}.

### 12.9.5 Config_XTAL_Ftrim

(TestID = 29)

A crystal oscillator provides the timing for all elements of the chip. The chip provides a means to “pull” the crystal’s value so that the oscillator runs at a desired frequency. Under normal operation this pull value is read from the Persistent Store at boot. This command provides immediate control over the pull value.

The value of `xtal_ftrim` takes effect immediately and will remain until the next device reset. The command does not adjust the crystal trim value held in the Persistent Store.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `xtal_ftrim` | uint16 | `0` | 63 | 1 bit maps to approximately 110 fF (femtoFarad) |

Typical parameter value: {`xtal_ftrim` = `27`}.

### 12.9.6 Config_UAP_LAP

(TestID = 30)

This command configures the upper address part (UAP) and lower address part (LAP) that are used in test transmissions.

This command is unusual in that it has a parameter of type `uint32`. This can be thought of as two `uint16`s, with the more significant word of the `uint32` in the first `uint16`.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `uap` | uint16 | `0` | `0xffff` | |
| `lap` | uint32 | `0` | `0x00ffffff` | |

Typical parameter values: {`uap` = `0x005b`, `lap` = `0x00123456`}.

### 12.9.7 Config_Access_Errs

(TestID = 31)

The receiver uses a sliding correlator to determine that it has matched the start of a packet. The receiver allows up to `n_errs` errors before the match is rejected.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `n_errs` | uint16 | `0` | `15` | |

Typical parameter value: {`n_errs` = `10`}.

### 12.9.8 Config_IQ_Trim

(TestID = 32)

Sets the IQ trim value to trim, overriding the value calculated by the internal calibration algorithm.

Use of this command is unnecessary for all normal users.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `trim` | uint16 | `0` | TBD | |

Typical parameter value: {`trim = 12`}.

### 12.9.9 Config_Lo_Lvl

(TestID = 35)

This command sets the value of the Analogue Local Oscillator output level to `lvl`, overriding the value calculated by the internal calibration algorithm.

**Note:**

Use of this command is unnecessary for all normal users.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| `lvl` | uint16 | `0` | `15` | |

Typical parameter value: {`lvl` = 15}.

### 12.9.10 Config_Hopping_Sequence

(TestID = 45)

Configures the set of hop frequencies to be used with RxData2, TxData2 and BER2. The functionality is analogous to the HCI Set_AFH_Host_Channel_Classification command.

The command is needed because Japanese regulatory approval sometimes requires a power density test, and AFH (Adaptive Frequency Hopping) can alter the density. The adjusted hopping sequence allows use of RxData2, TxData2 and BER2 while simulating a reduced hop set. This can be performed using 20 channels, the minimum number of active channels permitted by the AFH specification.

The command payload is a set of five `uint16`s whose bits represent the 79 channels: channel 0 is the lowest bit of the first word, channel 78 is bit 14 of the final word. If the bit is 1, the channel is used, else it is not.

The hopping algorithm is the same as that used when all frequencies are in use, but only on the selected frequencies. This guarantees that the radio spends equal amounts of time on each selected frequency.

| Parameter | Type | Min | Max | Comments |
|---|---|---|---|---|
| chan00_15 | uint16 | 0x0000 | 0xffff | Chan 0 is bit 0x0001 |
| chan16_31 | uint16 | 0x0000 | 0xffff | |
| chan32_47 | uint16 | 0x0000 | 0xffff | |
| chan48_63 | uint16 | 0x0000 | 0xffff | |
| chan64_78 | uint16 | 0x0000 | 0x7fff | Chan 79 is bit 0x4000 |

Example parameters, to use the lowest 20 channels only: {0xffff, 0x000f, 0x0000, 0x0000, 0x0000}.

# 13 Test Commands for BlueCore3-Flash

The following commands have been added for the BlueCore3-Flash chip to configure the power supply unit (PSU), the battery charger and associated functionality.

## 13.1 PSU_Trim

| Varid | Type | Data |
|---|---|---|
| 0x701f | PSU_Trim | Calibration data. |

Sets the battery charger terminal voltage.

State from Reset is 0. This is loaded with the contents of PS Key PSKEY_CHARGER_TRIM at boot time. To change a non-zero value use a figure other than 0 (setting to 0 will not change the value).

If a battery is connected and the value specified gives a voltage above 4.2 volts, this can cause the battery to overheat.

## 13.2 Buck_Reg

| Varid | Type | Data |
|---|---|---|
| 0x701f | Buck_Reg | On/Off |

Enables the Switch-mode regulator (EN_BUCK_REG).

The hardware state is 0. The firmware will latch it on at boot time. It can be used to power off the chip.

## 13.3 Mic_En

| Varid | Type | Data |
|---|---|---|
| 0x701f | Mic_En | On/Off |

Enables the linear regulator (EN_LIN_REG).

If SMPSU is being used, EN_LIN_REG is used as an extension of the mic bias and can be used to turn it on and off.

## 13.4 LED0

| Varid | Type | Data |
|---|---|---|
| 0x701f | LED0 | On/Off |

Controls the state of LED 0.

## 13.5 LED1

| Varid | Type | Data |
|---|---|---|
| 0x701f | LED1 | On/Off |

Controls the state of LED 1.

# 14 Example Transactions

These example transactions are provided for illustration only, but they may help clarify the structure and flow of data in the BCCMD protocol.

## 14.1 Read ChipVer

To read the value of variable ChipVer:

host → BlueCore BCCMDPDU_GETREQ:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 0x0000 | (BCCMDPDU_GETREQ) |
| 2 | BCCMDPDU.pdulen | 0x0009 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 0x1234 | (value chosen by host) |
| 4 | BCCMDPDU.varid | 0x281a | (BCCMDVARID_CHIPVER) |
| 5 | BCCMDPDU.status | 0x0000 | (BCCMDPDU_STAT_OK) |
| 6 | BCCMDPDU.d.u16 | 0x0000 | (empty) |
| 7 → 9 | (ignored) | 0x0000 | (zero padding) |

BlueCore → host BCCMDPDU_GETRESP:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 0x0001 | (BCCMDPDU_GETRESP) |
| 2 | BCCMDPDU.pdulen | 0x0009 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 0x1234 | (host's value returned) |
| 4 | BCCMDPDU.varid | 0x281a | (BCCMDVARID_CHIPVER) |
| 5 | BCCMDPDU.status | 0x0000 | (BCCMDPDU_STAT_OK) |
| 6 | BCCMDPDU.d.u16 | 0x0003 | (value of ChipVer) |
| 7 → 9 | (ignored) | 0x0000 | (zero padding) |

## 14.2 Write Config_UART

To set the value of variable Config_UART to the value `0x1234`:

host → BlueCore BCCMDPDU_SETREQ:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 0x0002 | (BCCMDPDU_SETREQ) |
| 2 | BCCMDPDU.pdulen | 0x0009 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 0x1235 | (value chosen by host) |
| 4 | BCCMDPDU.varid | 0x6802 | (BCCMDVARID_CONFIG_UART) |
| 5 | BCCMDPDU.status | 0x0000 | (BCCMDPDU_STAT_OK) |
| 6 | BCCMDPDU.d.u16 | 0x1234 | (data) |
| 7 → 9 | (ignored) | 0x0000 | (zero padding) |

BlueCore → host BCCMDPDU_GETRESP:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 0x0001 | (BCCMDPDU_GETRESP) |
| 2 | BCCMDPDU.pdulen | 0x0009 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 0x1235 | (host's value returned) |
| 4 | BCCMDPDU.varid | 0x6802 | (BCCMDVARID_ CONFIG_UART) |
| 5 | BCCMDPDU.status | 0x0000 | (BCCMDPDU_STAT_OK) |
| 6 | BCCMDPDU.d.u16 | 0x1234 | (data) |
| 7 → 9 | (ignored) | 0x0000 | (zero padding) |

## 14.3 Write Non-Existent Variable

Failing to set the value of non-existent `uint8` variable NONEXISTENT to the value `0x23`:

host → BlueCore BCCMDPDU_SETREQ:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 0x0002 | (BCCMDPDU_SETREQ) |
| 2 | BCCMDPDU.pdulen | 0x0009 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 0x1236 | (value chosen by host) |
| 4 | BCCMDPDU.varid | 0x9066 | (BCCMDVARID_NONEXISTENT) |
| 5 | BCCMDPDU.status | 0x0000 | (BCCMDPDU_STAT_OK) |
| 6 | BCCMDPDU.d.u16 | 0x0023 | (dummy data) |
| 7 → 9 | (ignored) | 0x0000 | (zero padding) |

BlueCore → host BCCMDPDU_GETRESP:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 0x0001 | (BCCMDPDU_GETRESP) |
| 2 | BCCMDPDU.pdulen | 0x0009 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 0x1236 | (host's value returned) |
| 4 | BCCMDPDU.varid | 0x9066 | (BCCMDVARID_NONEXISTENT) |
| 5 | BCCMDPDU.status | 0x0001 | (BCCMDPDU_STAT_NO_SUCH_VARID) |
| 6 | BCCMDPDU.d.u16 | 0x0000 | (dummy data) |
| 7 → 9 | (ignored) | 0x0000 | (zero padding) |

## 14.4 Write PS

To write the value of PSKEY_COUNTRYCODE to the setting for "France":

host → BlueCore BCCMDPDU_SETREQ:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 0x0002 | (BCCMDPDU_SETREQ) |
| 2 | BCCMDPDU.pdulen | 0x0009 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 0x1235 | (value chosen by host) |
| 4 | BCCMDPDU.varid | 0x7003 | (BCCMDVARID_PS) |
| 5 | BCCMDPDU.status | 0x0000 | (BCCMDPDU_STAT_OK) |
| 6 | BCCMDPDU.d.ps.pskey | 0x0002 | (PSKEY_COUNTRYCODE) |
| 7 | BCCMDPDU.d.ps.len | 0x0001 | (length) |
| 8 | BCCMDPDU.d.ps.stores | 0x0000 | (default stores) |
| 9 | BCCMDPDU.d.ps.data[0] | 0x0001 | (France) |

BlueCore → host BCCMDPDU_GETRESP:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 0x0001 | (BCCMDPDU_GETRESP) |
| 2 | BCCMDPDU.pdulen | 0x0009 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 0x1235 | (host's value returned) |
| 4 | BCCMDPDU.varid | 0x7003 | (BCCMDVARID_PS) |
| 5 | BCCMDPDU.status | 0x0000 | (BCCMDPDU_STAT_OK) |
| 6 | BCCMDPDU.d.ps.pskey | 0x0002 | (PSKEY_COUNTRYCODE) |
| 7 | BCCMDPDU.d.ps.len | 0x0001 | (length) |
| 8 | BCCMDPDU.d.ps.stores | 0x0000 | (default stores) |
| 9 | BCCMDPDU.d.ps.data[0] | 0x0001 | (France) |

## 14.5 Radiotest

To run the `txstart` command:

host → BlueCore BCCMDPDU_SETREQ:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 2 | (SETREQ) |
| 2 | BCCMDPDU.pdulen | 9 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 1235 | (value chosen by host) |
| 4 | BCCMDPDU.varid | 20484 | (VARID_RADIOTEST) |
| 5 | BCCMDPDU.status | 0 | (STAT_OK) |
| 6 | BCCMDPDU.d.test | 1 | (RADIOTEST_TXSTART) |
| 7 | BCCMDPDU.d.radiotest.txsi.lo_freq | 0x1234 | (lo_freq) |
| 8 | BCCMDPDU.d.radiotest.txsi.level | 0x1234 | (level) |
| 9 | BCCMDPDU.d.radiotest.txsi.mod_freq | 0x1234 | (mod_freq) |

BlueCore → host BCCMDPDU_GETRESP:

| uint16 | Field | Value | Meaning |
|---|---|---|---|
| 1 | BCCMDPDU.type | 1 | (GETRESP) |
| 2 | BCCMDPDU.pdulen | 9 | (number of uint16s in PDU) |
| 3 | BCCMDPDU.seqno | 1235 | |
| 4 | BCCMDPDU.varid | 20484 | (VARID_RADIOTEST) |
| 5 | BCCMDPDU.status | 0 | (STAT_OK) |
| 6 | BCCMDPDU.d.test | 1 | (RADIOTEST_TXSTART) |
| 7 | BCCMDPDU.d.radiotest.txsi.lo_freq | 0x1234 | (lo_freq) |
| 8 | BCCMDPDU.d.radiotest.txsi.level | 0x1234 | (level) |
| 9 | BCCMDPDU.d.radiotest.txsi.mod_freq | 0x1234 | (mod_freq) |

# 15 Build Definition Values

This section describes the set of "build definition" values accessible via the Get_Next_Builddef command, described in section 4.9.

The list of definitions will be increased as new types of firmware build are generated.

The values listed will not be changed, even though some values will become obsolete.

- BUILDDEF_NONE (0x0000)

  This is a special value, used to indicate "not a build definition".

- BUILDDEF_CHIP_BASE_BC01 (0x0001)

  Firmware has been written to run on BlueCore01b (the only “bc01” chip).

- BUILDDEF_CHIP_BASE_BC02 (0x0002)

  Firmware has been written to run on one of the BlueCore2 chips (BlueCore2-External, BlueCore2-ROM, etc.).

- BUILDDEF_CHIP_BC01B (0x0003)

  Firmware has been written to run on BlueCore01b.

- BUILDDEF_CHIP_BC02_EXTERNAL (0x0004)

  Firmware has been written to run on BlueCore2-External.

- BUILDDEF_BUILD_HCI (0x0005)

  Normal HCI firmware build.

- BUILDDEF_BUILD_RFCOMM (0x0006)

  Normal RFCOMM firmware build.

- BUILDDEF_BT_VER_1_1 (0x0007)

  Firmware has been written to support version 1.1 of the Bluetooth specification.

- BUILDDEF_TRANSPORT_ALL_HCI (0x0008)

  All *HCI* host transports are supported. These are currently BCSP, USB, H4, H4DS and (for chips other than BlueCore01b and BlueCore2-External) the Three-wire UART. This identifier does not imply the presence of support for the “user” UART configuration See BUILDDEF_TRANSPORT_USER. It also does not imply support for the “none” configuration. See BUILDDEF_TRANSPORT_NONE.

- BUILDDEF_TRANSPORT_BCSP (0x0009)

  BCSP host transport is supported.

- BUILDDEF_TRANSPORT_H4 (0x000a)

  H4 host transport is supported.

- BUILDDEF_TRANSPORT_USB (0x000b)

  (Bluetooth device) USB host transport is supported.

- BUILDDEF_MAX_CRYPT_KEY_LEN_56 (0x000c)

  Maximum effective encryption key length is 56 bits.

- BUILDDEF_MAX_CRYPT_KEY_LEN_128 (0x000d)

  Maximum effective encryption key length is 128 bits.

- BUILDDEF_TRANSPORT_USER (0x000e)

  Supports a VM application having direct control of the UART.

- BUILDDEF_CHIP_BC02K (0x000f)

  Firmware for the first version of BlueCore2-ROM (TSMC variant, aka bc02k).

- BUILDDEF_TRANSPORT_NONE (0x0010)

  The firmware can be configured to support no host connection.

- BUILDDEF_REQUIRE_8MBIT (0x0012)

  This firmware requires an 8 Mbit flash device. (At present, the absence of this definition implies the firmware requires a 4 Mbit flash device.)

- BUILDDEF_RADIOTEST (0x0013)

  Some radiotest (BIST) commands are present. (These commands are normally used by bluetest.exe.)

- BUILDDEF_RADIOTEST_LITE (0x0014)

  A reduced set of radiotest (BIST) commands is present. If BUILDDEF_RADIOTEST is present and this identifier is not present, then the full set of commands is available.

- BUILDDEF_INSTALL_FLASH (0x0015)

  The firmware includes drivers for flash devices. This is normally used to provide storage for a VM application and/or PS Keys.

- BUILDDEF_INSTALL_EEPROM (0x0016)

  The firmware includes drivers for EEPROM devices. This is normally used to provide storage for a VM application and/or PS Keys. Normally, this support can only be used if the system either has no flash storage or the firmware has no support for flash.

- BUILDDEF_INSTALL_COMBO_DOT11 (0x0017)

  The firmware provides a mechanism for interoperating with an 802.11b radio. (Bluetooth/802.11b “combo” support.)

- BUILDDEF_LOWPOWER_TX (0x0018)

  The firmware includes support for very low power transmissions.

- BUILDDEF_TRANSPORT_TWUTL (0x0019)

  The Three Wire UART Transport Layer host transport is supported. (This is sometimes referred to as “H5”.)

- BUILDDEF_COMPILER_GCC (0x001a)

  The firmware was built using a C compiler based on gcc. (Absence of this definition currently means that the firmware has been built using a Norcroft XAP C cross compiler.)

- BUILDDEF_CHIP_BC02_BC02C (0x001b)

  Firmware for the second source of BlueCore2-ROM (ST variant, aka bc02c).

  **Note:** BUILDDEF_CHIP_BC02_BC02K may also be present, as code for the two devices is almost identical. If both are present, the code is for bc02c.

- BUILDDEF_CHIP_BC02_BC02T (0x001c)

  Firmware for the stripped-down version of bc02c (ST variant of BlueCore2-ROM). Known as bc02t.

  **Note:** BUILDDEF_CHIP_BC02_BC02K and/or BUILDDEF_CHIP_BC02_BC02C may also be present, as code for the devices is almost identical. If more than one is present, the code is for bc02t.

- BUILDDEF_CHIP_BASE_BC3 (0x001d)

  Firmware for the BlueCore3 family of chips.

- BUILDDEF_CHIP_BC3_BC3N (0x001e)

  Firmware for BlueCore3-ROM (aka bc3n).

- BUILDDEF_CHIP_BC3_BC3K (0x001f)

  Firmware for BlueCore3-Multimedia (aka bc3k).

- BUILDDEF_INSTALL_HCI_MODULE (0x0020)

  The HCI interface is available.

- BUILDDEF_INSTALL_L2CAP_MODULE (0x0021)

  The L2CAP interface is available.

- BUILDDEF_INSTALL_DM_MODULE (0x0022)

  The DM interface is available.

- BUILDDEF_INSTALL_SDP_MODULE (0x0023)

  The SDP interface is available.

- BUILDDEF_INSTALL_RFCOMM_MODULE (0x0024)

  The RFCOMM interface is available.

- BUILDDEF_INSTALL_HIDIO_MODULE (0x0025)

  The HID functionality is available.

- BUILDDEF_INSTALL_PAN_MODULE (0x0026)

  The PAN interface is available.

- BUILDDEF_INSTALL_IPV4_MODULE (0x0027)

  The IPV4 functionality is available.

- BUILDDEF_INSTALL_IPv6_MODULE (0x0028)

  The IPV6 functionality is available.

- BUILDDEF_INSTALL_TCP_MODULE (0x0029)

  The TCP interface is available.

- BUILDDEF_BT_VER_1_2 (0x002a)

  Firmware has been written to support version 1.2 of the Bluetooth specification.

- BUILDDEF_INSTALL_UDP_MODULE (0x002b)

  The UDP interface is available.

- BUILDDEF_REQUIRE_0_WAIT_STATES (0x002c)

  The firmware must be run from “fast flash” memory (the CPU does not use wait states when reading from flash).

- BUILDDEF_CHIP_BC3_PADDYWACK (0x002d)

  Firmware for BlueCore3-Flash.

- BUILDDEF_CHIP_BC3_COYOTE (0x002e)
  BUILDDEF_CHIP_BC4_COYOTE

  Firmware for BlueCore4-External.

- BUILDDEF_CHIP_BC3_ODDJOB (0x002f)
  BUILDDEF_CHIP_BC4_ODDJOB

  Firmware for BlueCore4-ROM.

- BUILDDEF_TRANSPORT_H4DS

  The H4DS host transport is supported.

- BUILDDEF_CHIP_BASE_BC4

  Firmware for the BlueCore4 family of chips.

# Document References

| Document ID | Document Title | CSR Reference |
|---|---|---|
| [BCCMD] | BCCMD Protocol | bcore-sp-002P |
| [BCCMDSEC] | BCCMD Protocol Security | bcore-sp-006P |
| [BCSP] | BlueCore Serial Protocol (BCSP) | bcore-sp-012P |
| [BCHCI] | HCI Implementation for pre-HCIStack1.2v18.1 releases<br>for HCIStack1.2v18.1 and later releases | bcore-me-007P<br>bcore-an-032P |
| [BOOTMODES] | Understanding and Using Bootmodes | bcore-an-019P |
| [BT2.0] | Specification of the Bluetooth System, Version 2.0 + EDR, Core Package, 4 November 2004 | n/a |
| [BTTEST] | Bluetooth SIG Test Specification (RF); rev 0.91, 4 July 2001 | n/a |
| [HQCMDS] | HQ Commands | bcore-sp-003P |
| [I2C] | The $I^2C$-Bus Specification, version 2.1, January 2000 | n/a |
| [ROMBOOT] | Booting BlueCore ROM | bcore-me-014P |
| [SCATTERNET] | Scatternet Support | bcore-me-003P |
| [TWUTL] | Three-wire UART Transport Layer, Revision V0.95, 22 January 2004 | n/a |

# Terms and Definitions

| | |
|---|---|
| ACL | Asynchronous Connection-Less |
| ADC | Analogue to Digital Converter |
| AGC | Auto Gain Control |
| AMUX | Analogue Multiplexor |
| BCCMD | BlueCore Command |
| BCSP | BlueCore Serial Link Protocol, described in [BCSP] |
| BER | Bit Error Rate |
| BIST | Built-In Self-Test |
| BlueCore™ | Group term for CSR's range of Bluetooth chips |
| BlueCore01b | CSR Bluetooth chip |
| BlueCore2-External | CSR Bluetooth chip |
| BlueCore2-ROM | CSR Bluetooth chip (TSMC and ST variants, aka bc02k and bc02c respectively) |
| BlueCore3-Flash | CSR Bluetooth chip |
| BlueCore3-Multimedia | CSR Bluetooth "multimedia" chip (aka bc3k) |
| BlueCore3-ROM | CSR Bluetooth chip (aka bc3n) |
| BlueCore4-External | CSR Bluetooth chip; supports EDR |
| BlueCore4-ROM | CSR Bluetooth chip; supports EDR |
| BlueLab™ | CSR's software development kit for applications running in BlueCore's VM |
| Bluetooth® | Set of technologies providing audio and data transfer over short-range radio connections |
| CODEC | Coder Decoder |
| CRC | Cyclic Redundancy Check |
| CSR | Cambridge Silicon Radio |
| DAC | Digital to Analogue Converter |
| DFU | Device Firmware Upgrade |
| DM | Data Medium Rate |
| EDR | Enhanced Data Rate |
| EEPROM | Electrically Erasable Programmable Read Only Memory |
| FEC | Forward Error Correction |
| gcc | GNU C Compiler |
| H4 | HCI UART Transport Layer |
| H4DS | H4 Deep Sleep |
| H5 | See TWUTL |
| HCI | Host Controller Interface; interface between host and Bluetooth module |
| HID | Human Interface Device |
| HQ | Host Query |
| $I^2C$ | Inter Integrated Circuit |
| IF | Intermediate Frequency |
| IrDA | Infrared Data Association |
| IQ | In-phase and Quadrature components of a signal |
| L2CAP | Logical Link Control and Adaptation Protocol |
| LAP | Lower Address Part |
| LC | Link Controller |

| | |
|---|---|
| LMP | Link Manager Protocol |
| LNA | Low Noise Amplifier (external circuitry used to improve radio reception) |
| LO | Local Oscillator |
| LS | Least Significant (bit) |
| LUT | Lookup Table |
| MMU | Memory Management Unit |
| NACK | Negative Acknowledge |
| PA | Power Amplifier |
| PAN | Personal Area Network |
| PCM | Pulse Code Modulation |
| Persistent Store | Storage of BlueCore's configuration values in non-volatile memory |
| PIO | Parallel Input Output (port) |
| PRBS | Pseudo-Random Bit Sequence |
| PS Key | Persistent Store Key |
| PSU | Power Supply Unit |
| RCS | Revision Control System |
| RFCOMM | Protocol layer providing serial port emulation over L2CAP; element of Bluetooth |
| RO | Read Only |
| RPC | Remote Procedure Call |
| RSSI | Received Signal Strength Indicator |
| RW | Read Write |
| RX | Receiver |
| SCO | Synchronous Connection Oriented |
| SDP | Service Discovery Protocol |
| SIG | (Bluetooth) Special Interest Group |
| ST | STMicroelectronics |
| TWUTL | Three-wire UART Transport Layer, defined by [TWUTL] – an HCI host transport that bears a striking resemblance to BCSP. Informally called "H5". |
| TSMC | Taiwan Semiconductor Manufacturing Company |
| TX | Transmitter |
| UAP | Upper Address Part |
| UART | Universal Asynchronous Receiver Transmitter |
| UDP | User Datagram Protocol |
| USB | Universal Serial Bus |
| varid | variable identifier |
| VCO | Voltage Controller Oscillater |
| VDD | Voltage Drain Drain |
| VM | Virtual Machine |
| WO | Write Only |
| XAP | Low-power silicon-efficient RISC microprocessor |
| XTAL | Crystal |

# Document History

| Revision | Date | History |
|---|---|---|
| a | 29 Nov 02 | Document originally published as CSR reference bc01-an-069 (revisions a through c; versions through HCIStack1.1v15.x builds. New revision control number allocated to align with HCIStack1.1v16.x builds, which includes the following amendments: Re-organised commands' sections; added commands Cancel_Page, ChipAnaVer and BuildID_Loader; added extra BUILDDEF values; changed meaning of BUILDDEF_TRANSPORT_ALL; added BlueCore2-External/PIO pull up/down description; updated ChipRev information; updated Map_SCO_PCM description; improved description of Crypt_Key_Length. |
| b | 26 Mar 03 | Not published. |
| c | 16 Jul 03 | Corrected meaning of BUILDDEF_RADIOTEST_LITE. Added BER_Threshold, introduced in builds HCIStack16.7.4. Added BUILDDEF_COMPILER_GCC and BUILDDEF_TRANSPORT_TWUTL. Added comment that BER_Threshold command does not appear in HCIStack17.x builds. Added PIO_Strong_Bias and Bypass_UART commands. Corrected command name: "ChipAnaRev" to "ChipAnaVer". Renamed BUILDDEF_TRANSPORT_ALL to BUILDDEF_TRANSPORT_ALL_HCI. |
| d | 26 Jul 04 | Created one document covering all BCCMD commands by combining BCCMD Persistent Store Commands (bcore-sp-004P), BCCMD Test Commands (bcore-sp-001P) and BCCMD Protocol Radio Test Command set (bc01-sp-045P) into this one document.<br>Added new test commands for BlueCore3-Flash, details for the RSSI_ACL command and references to the Three-wire UART protocol. Updated formatting and corrected typos. |
| e | 27 Sep 05 | Added information on new commands Ana_Ftrim_Readwrite, Get_External_Clock_Period and HostIO_Enable_Debug. Re-ordered sections and made minor formatting updates. |

# BCCMD Commands

## bcore-sp-005Pe

## September 2005