ROOT-2

Interface Specification

Release 1.1 - January 26, 2006

RPM Systems Corporation

17371 N.E. 67th Court, Suite A-5

Redmond, Washington, 98052 USA

Ph: (425)869-3901 FAX: (425)883-9552

Contents

1. Introduction

Root 2 is a USB host controller which is targeted for test applications. It is USB 2.0 specification compliant for low-speed, full-speed and high-speed interfaces, and is designed to be employed in a development or manufacturing test environment for the purpose of testing Universal Serial Bus (USB) peripherals. The device provides a subset of the features typically provided by a USB Host and Root Hub, but its activity is controlled via an RS-232 serial or Ethernet communications interface. This allows the test engineer to completely control the low-level root hub functions without the interference of a complicated operating system such as would be encountered on a Windows or Mac PC. Root 2 provides the following basic capabilities:

- High-speed / Full-speed / Low-Speed USB Serial Interface Engine (SIE)
- Device connect/disconnect detection and reporting
- Automatic device configuration (automatic mode)
- Support for downstream hub (automatic mode)
- On-command suspend/resume capability
- Remote wake-up support
- On-command USB Reset generation
- Vbus current measurement
- Vbus voltage control (4.25V ~ 5.50V)
- External Trigger Inputs and Outputs
- 8-bit user Output port
- RootScript script language support
- Built-in Go-NoGo device testing with Monitor 1
- Controlled via High-speed RS-232 serial or Ethernet host communications

1.1 Definition of Terms

The terms defined in this section will be used throughout this document.

Controller The device used to control Root 2 via the serial or Ethernet communications port (e.g., a PC or ATE host).

Root Port The USB port on Root 2.

RootScript Root 2's scripting language, which allows pre-written scripts to be downloaded and stored in Root 2 and subsequently invoked by the Controller.

Low Speed A device or USB port operating at 1.5Mbps, as defined for low-speed USB devices.

Full Speed A device or USB port operating at 12Mbps, as defined for full-speed USB devices.

High Speed A device or USB port operating at 480Mbps, as defined for high-speed USB devices.

2. System Overview

This section provides a brief overview of Root 2's interfaces and capabilities. Some topics are dealt with in more detail in the later sections of the document.

2.1 LED Indicators

Root 2 includes several LED indicators on its front and rear panels, which provide information regarding the state of the Root 2 and the USB port. The indicators are identified on the Root 2 product label affixed to its top surface, and the following sections provide additional information regarding their interpretation.

At power-on, the top two of the three vertically oriented green Status indicators will light immediately, indicating good power. The four, multicolored horizontally oriented status indicators will flash while Root 2 performs its power-on self test. After several seconds, the third of the green Status indicators will light, indicating that the Root 2 has configured its USB SIE hardware.

The Host Comm LED will flash in response to any communication with the Controller, whether communications is via the serial port or the Ethernet port. When no communications is active, this LED will flash briefly once per second, indicating that Root 2 is alive.

When USB power is enabled and a full-speed device is connected and configured on the Root Port, the green Full Speed LED will be lit. When USB power is enabled, and a high-speed device is connected and configured on the Root Port, the High Speed LED will be lit.

Any error condition detected during USB operation will cause the red USB Error LED to be lit. Note that the LED will remain lit only until the next USB transaction is completed.

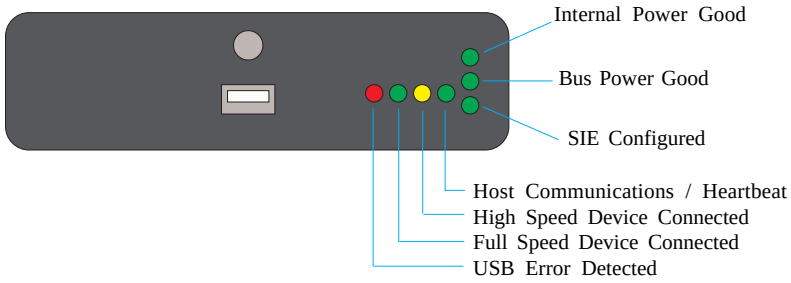

Figure 2-1 Root 2 Front Panel Indicators

2.2 Control Connector Signal Descriptions

Root 2 communicates with the Controller via a full-duplex RS-232 serial data connection. The default serial data format is 115.2K baud with one start bit, eight data bits and one stop bit. The baud rate may be reprogrammed after reset (see Section 3.7).

Root 2 provides a 25-pin female D subminiature connector containing the following signals:

Pin #	Signal	Description	In/Out	Notes
1	Aux_TxD	Auxiliary Serial Xmt	Out	Do Not Connect
2	RxD	Serial Receive Data	In	
3	TxD	Serial Transmit Data	Out	
4	Aux_RxD	Auxiliary Serial Rcv	In	Do Not Connect
5~13	GND	Signal Ground		
14	TrigIn0	Trigger Input 0	In	
15	TrigOut0	Trigger Output 0	Out	
16	TrigIn1	Trigger Input 1	In	
17	PD0	Parallel data bit 0 (LSB)	Out	
18	PD1	Parallel data bit 1	Out	
19	PD2	Parallel data bit 2	Out	
20	PD3	Parallel data bit 3	Out	
21	PD4	Parallel data bit 4	Out	
22	PD5	Parallel data bit 5	Out	
23	PD6	Parallel data bit 6	Out	
24	PD7	Parallel data bit 7 (MSB)	Out	
25	Mode	Parallel Data Mode	In	Reserved - Do Not Connect

2.3 Signaling Levels and Termination

The serial communications lines (TxD, RxD, Aux_TxD and Aux_RxD) employ EIA-232 electrical levels. All other signals are 3.3V logic compatible levels. The trigger output and parallel data lines are source terminated on Root 2 with 50-ohm series resistors. The TrigIn0 and TrigIn1 lines are provided with internal 4.7K-ohm pullup resistors to 3.3V on Root 2.

2.4 External Trigger Inputs

Root 2 provides two external, logic-level trigger inputs. The TrigIn0 and TrigIn1 inputs are asserted low, and filtered in hardware to remove noise. A trigger input will be detected and latched by Root 2 on the falling-edge transition. A Trigger input must be driven low and held low for at least 1uS to guarantee recognition by the Root 2. If the Root 2 is in immediate-mode (i.e., not executing a RootScript), the trigger event will be reported to the Controller as an Asynchronous Response (See *Asynchronous Responses* in Section 4). During RootScript execution, trigger inputs will be latched, but will not be acted upon until they are specifically addressed by the script (see *RS_Check* command in Section 5.5).

2.5 Trigger Bus Outputs

The Trigger Bus Outputs are essentially a parallel data port which can be written directly via a Controller or RootScript command. The data provided by the Controller or RootScript command is driven onto the parallel port, and the output strobe, TrigOut0, is toggled. The data will remain on the output port until a new piece of data is written by the Controller or RootScript. TrigOut0 is a low-going strobe approximately 10uS in duration. Approximately 10uS data setup time is provided prior to the assertion (low) of the strobe. The parallel port data lines are capable of sourcing or sinking up to 8mA each, and may be used to directly drive LED indicators if suitable current-limiting is employed (e.g., 150-ohm or larger value series resistors).

2.6 Data Communications Protocol

The Root 2 employs an escaped binary data communications protocol. Data is transmitted as variable length packets, and packet control is implemented using ASCII escape sequences. All serial data transmissions to and from the Root 2 are framed in the following message protocol:

<SOP><transmission code>{data}<EOP>

SOP is a start-of-packet marker consisting of two ASCII bytes: <Esc>S (0x1B 0x53)

EOP is an end-of-packet marker consisting of two ASCII bytes: <Esc>E (0x1B 0x45)

<transmission code> is a single byte code indicating the nature of the transmission. For commands sent to Root 2, this byte will contain a command code. In responses received from Root 2, this byte will contain a response code.

{data} is a string of binary data bytes 0 ~ 512Kbytes bytes in length.

The ASCII Escape character (0x1B) is used as a special character, to implement message control. Since the protocol supports binary data transfer, the data value 0x1B may be encountered in the data stream. The two-byte pattern <Esc><Esc> (0x1B 0x1B) is used to represent the occurrence of the single byte 0x1B if it is encountered in {data}.

2.7 Root 2 Commands and Responses

Commands which are transmitted to the Root 2 from the Controller over the serial port, using the message protocol discussed in the previous section, are referred to as *interactive commands*. Any properly formatted interactive command will invoke a response from Root 2. For all commands issued to Root 2, the <transmission code> referred to in the previous section will be a Root 2 command code. The interactive commands, and their responses, are discussed in section 3.

Responses transmitted by the Root 2 to the Controller are of two types: Command Responses and Asynchronous Responses. Command Responses are transmitted in direct response to a command received from the Controller. Asynchronous Responses are transmitted as the result of asynchronous conditions, for example, a USB device connect. Command Responses are discussed in Section 3, in coordination with the commands with which they are associated. Asynchronous responses are discussed in Section 4. In all Root 2

responses, Command Responses or Asynchronous Responses, the <transmission code> referred to in the previous section will be a Root 2 response code.

Whereever parameters are passed by Root 2 commands and responses which are larger than one byte, i.e., 16-bit word or 32-bit longword parameters, they are always passed MSB-first (big endien). For example, the 32-bit hexadecimal value 0x12344321 would be passed as four bytes: <0x12><0x34><0x43><0x21>.

2.8 Modes of Operation

Root 2 has two main modes of operation: Interactive Mode and Script Mode. Interactive Mode is the normal mode of operation, in which commands are issued by the Controller and responded to by Root 2. Root 2 may also issue Asynchronous Responses in Interactive Mode to indicate the occurrence of asynchronous conditions. Within Interactive Mode operation, Root 2 supports an additional mode of operation called Automatic Mode, in which many of the functions of a typical USB host controller are performed automatically by Root 2. Automatic Mode is discussed in the next section.

Root 2 supports a scripting language, called RootScript, which consists of the standard set of Root 2 Interactive Commands, plus additional scripting commands for program flow control. A RootScript is downloaded into Root 2 in Interactive Mode. Once the script has been loaded into Root 2, it may be executed, causing Root 2 to enter Script mode. Root 2 will remain in script mode, executing RootScript commands, until the script terminates or until a new Interactive Command is received on the serial port. Automatic Mode is note available in Script Mode. RootScripting is discussed in section 5.

At power up, Root 2 typically defaults to Interactive Mode operation with Automatic Mode enabled. However, Root 2 allows for a *default script* to be loaded and saved in on-board Flash memory. Once the default script has been saved, Root 2 can be programmed to enter Script Mode immediately after power up, executing the default script. Refer to Section 6 for more information on using a default script.

Functions intrinsic to the basic operation of the USB, such as SOF (Start of Frame packet) generation, overcurrent protection, etc., are always handled automatically by Root 2 hardware, regardless of the mode of operation.

2.8.1 Automatic Mode

The default mode of operation for the Root 2 is Automatic Mode. In this mode, the Root 2 will automatically detect the attachment of a new device to the root port, reset it, enumerate it and configure it. If the attached device is a hub, the Root 2 will read the hub's Hub Descriptor, configure its status endpoint and enable power to its downstream ports. The Root 2 will thereafter routinely poll the hub's status, identify connect events on the hub's downstream ports and reset and enumerate newly connected devices. In addition, non-hub devices connected to downstream ports of the hub device will be automatically reset and enumerated upon attachment. Non-hub low-speed or full-speed Interrupt devices attached either directly to the root port or to the downstream port of an attached hub will be configured for up to four endpoints, in addition to the default endpoint, based on each device's Interface and Endpoint descriptors. Interrupt endpoints will be automatically polled in accordance with their Endpoint and Interface descriptors. Root 2 operation in Automatic mode, with regard to

the connection of a new device, is essentially identical to the sequence that a typical USB host would exhibit in enumerating and configuring a new device.

In Automatic Mode, the Root 2 will generate an asynchronous response to the Controller any time an event occurs on the root port or on the downstream port of a hub attached to the root port. Examples of detectable events are Device Connect, Device Disconnect, Remote Wake-up, or a non-NAK response to polling on any endpoint.

Automatic Mode may be enabled and disabled by command from the Controller (*Root_Config* command). With Automatic Mode disabled, the Root 2 will not automatically enumerate new devices, nor will it automatically configure and poll downstream hubs or functions. The Controller then becomes responsible for issuing discrete device requests for all device and hub configuration, including polling hub status for new device connects on its downstream ports and polling downstream devices. Most Root 2 Interactive Commands can be used whether Root 2 is in Automatic Mode or not, however care must be taken that the commands issued do not modify the configuration of devices such that Automatic Mode activity is adversely affected.

It is recommended that the user application make use of Automatic Mode whenever possible. Automatic Mode performs a large amount of processing that will otherwise have to be handled from the user application. It performs all device detection and enumeration, and automatically handles operation of attached hubs. In addition, Automatic Mode operation provides Root 2 with a set of knowledge regarding devices attached to the USB. Since it enumerates and configures each device upon attachment, Root 2 is able to retain information regarding the device's speed and control endpoint parameters (e.g., bMaxPacketSize0). If the device is low-speed or full-speed and is connected downstream of a high-speed hub, Automatic Mode will retain information necessary to perform split transactions to that device. This knowledge simplifies the issuing of additional traffic to the device by the user application.

2.8.2 Built-In Go-NoGo Testing Using Monitor 1

Root 2 provides a special mode of operation in support of No-NoGo device testing using the Monitor 1 handheld test controller. This operation is enabled using the RootConfig command (Section 3.7), and is described in detail in RPM Application Note 3, *Go / No-Go Testing Using Monitor 1*. This application note is provided on the Root 2 support CD-ROM, and is also available from the RPM web site (http://www.rpmsys.com). Monitor 1 product information is also available on the web site.

2.9 Root 2 Host Controller Functions

Root 2 provides the basic host controller functions necessary for the proper operation of the USB. These include device speed detection on the root port, Vbus power switching and overcurrent protection, SOF (start of frame) generation, CRC generation and checking and packet handshaking. These functions are essentially invisible to the user, with the exception that SOF generation can be controlled by the Suspend and Resume commands. The Root 2 host controller is also responsible for the timing of transactions within a frame and the detection of overrun conditions, IGNORE conditions, etc.

3. Root 2 Immediate Mode Commands

This section details the set of interactive commands supplied by Root 2. The format of each command and its associated response are discussed. All transmissions, commands and responses, are conducted using the protocol discussed in section 2.4.

Many Root 2 commands return a Response Status, which will be designated <RespStatus> in the description of the response. Table 3-1 lists the possible values for <RespStatus>, for both Command responses and Asynchronous responses.

3.1 Device Request

The DevRqst command conducts a single control transfer, consisting of a valid USB device request, to be issued to any attached device. A valid USB device request is any properly formed request, not necessarily one that the target device is capable of handling. This command may be used with Automatic Mode enabled or disabled to issue any standard, class-specific or vendor-specific device request. When processing a DevRqst, the Root 2 will perform the entire request, including Setup, Data and Status phases. Multiple data transactions

Table 3-1: Response Status Values

Value	Indication	Description
0x00	Success	Command completed successfully
0x02	Ack	Recevied Ack from device
0x03	Data0	Received Data0 PID from device
0x06	Nyet	Received Nyet from device
0x07	Data2	Received Data2 PID from device
0x0A	Nak	Received Nak from device
0x0B	Data1	Received Data1 PID from device
0x0E	Stall	Received Stall from device
0x80	Ignore	Device failed to respond to traffic
0x81	Data CRC Error	Detected a Data CRC Error in received data
0x82	Data Toggle Error	Detected a Data Toggle Error in received data
0x83	Sync Error	The value of the Sync byte in a received data packet was incorrect
0x84	Babble Error	Detected traffic from device after end of frame
0x85	PID Error	The self-check bits of a received PID were incorrect
0x87	Configuration Error	Root 2 encountered an error while parsing a device's Configuration Descriptor in Automatic mode
0x8A	NAK Timeout	The target device Nak'd continuously for more than 500mS in response to any part of a Device Request
0x8B	Request Timeout	The target device failed to complete a Device Request within 5 seconds
0x8C	Command Active	A new USB command (data transfer, power, USB Reset, etc.) was issued while a BlockTrans command was active
0x8D	Unknown Device	An Auto Mode data transfer command was issued to a device that Root 2 does not recognize

will be conducted as necessary to complete the request.

As control transfers, DevRqst commands are always targeted to the control endpoint (endpoint 0) of the device. INs and Outs which are generated by the DevRqst command and are NAK'd by the device are retried until they are successful (ACK'd), or until they time out. Per the USB Specification, a time-out will be generated during the data portion of the control transfer if a device does not return with a non-NAK response after 500mS, or if the duration of an entire device request sequence exceeds 5 seconds.

If the device being addressed by the DevRqst command has been connected and automatically enumerated by Root 2 in Automatic Mode, Root 2 will use the information it has collected during device configuration regarding device speed (full speed or low speed) and maximum packet size (*bMaxPacketSize0* field of the Device Descriptor) to properly conduct the DevRqst command. If the device has not been configured in Automatic Mode, or if the user desires for whatever reason to override the automatic-mode parameters for device speed or *bMaxPacketSize0* , this may be done by setting the <OVRD> flag in the <Address> byte of the DevRqst command, and including the additional <Control> byte.

Note that DevRqst conducts a complete transaction, including Setup, Data and Status phases. DevRqst accommodates multiple data transfers if necessary to complete the entire data transaction, and automatically handles data-toggle checking and NAK retries on data and status packets. If the test engineer requires device interaction to be broken down onto a more detailed level than that provided by DevRqst, the DevTrans command should be used. However, conducting transactions using DevTrans requires that the test script or interactive test software handle each packet of the transaction separately, including generating NAK retries.

The reader is referred to Chapter 9 of the USB Specification for information regarding the proper construction of a USB device request. This reference also addresses the details of the Standard USB Device requests. The details of class-specific (e.g., HID class specific or Audio class specific) requests are addressed in the corresponding Class specifications.

The form of the DevRqst command is:

<SOP><CMD_DevRqst><Address>{<Control>}<Data><EOP>

<CMD_DevRqst> is the DevRqst command ID, 0x01.

<Address> contains the target device address (0 ~ 127) in bits 0~6, and the <OVRD> flag in bit 7. For requests to devices which have been configured via normal Automatic Mode operation, OVRD is typically set to 0, and the <Control> byte is omitted from the command. To override the Automatic Mode settings, or to issue requests to a device which has not been configured by Root 2 in Automatic Mode, the OVRD flag is set to 1, and the <Control> byte is included in the command.

The <Control> parameter is included in the DevRqst command *only* if the <OVRD> bit is set =1 in the <Address> byte. <Control> allows the device's speed and *bMaxPacketSize0* value to be defined for devices which are not configured in Automatic Mode. The layout of <Control> is shown in figure 3-1.

The <Speed> bits are set to indicate the bus speed of the target device, as follows:

Bit 1	Bit 0	Target Bus Speed
0	0	Low Speed
0	1	Full Speed

1	0	High Speed
1	1	<Invalid>

<MaxPacketSize> is the maximum data packet size to be used for transfers to and from the device. This information is normally obtained by Root 2 in Automatic Mode by reading the *bMaxPacketSize0* byte of the device's Device Descriptor, and is required to properly conduct some transactions with the device. The <MaxPacketSize> field of the <XferConfig> byte is encoded as follows:

Bit 1	Bit 0	MaxPacketSize
0	0	8 bytes
0	1	16 bytes
1	0	32 bytes
1	1	64 bytes

<Data> consists of binary data bytes comprising the valid USB device request, beginning with *bmRequestType* and ending with any data to be included with the request. At least eight bytes of data are required to make up the contents of the Setup packet. For device-to-host (Setup-IN) transactions, only these eight bytes of data are required. For host-to-device (Setup-OUT) transactions, any additional data to be sent with the request should be included in <Data> following the initial Setup information. Note that <Data> should include only actual data. Root 2 automatically generates PIDs, CRC's, etc, so this information should not be included in the <Data> parameter.

The device request is issued on the bus, and the Root 2 responds with the following:

<SOP><RESP_DevRqst><RespStatus>{<Response>}<EOP>

<RESP_DevRqst> is the DevRqst response ID, 0x81.

<RespStatus> is the response status as indicated in Table 3-1.

<Response> is the data, if any, returned by the target device.

DevRqst can accommodate an overall device response data size (the total number of bytes received in the data stage of a device-to-host request) of up to 4Kbytes on a single request.

Examples:

Read the device descriptor from device at address 2 configured in Automatic Mode:

Command: <SOP><0x01><0x02>{0x80 0x06 0x00 0x01 0x00 0x00 0x12 0x00}<EOP>
Response: <SOP><0x81><0x00>{0x12 0x01 0x01 0x01 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x??}<EOP>

Read the device descriptor from a low-speed device at address 0 - force maxPacketSize=8

Command: <SOP><0x01><0x80><0x00>{0x80 0x06 0x00 0x01 0x00 0x00 0x12 0x00}<EOP>
Response: <SOP><0x81><0x00>{0x12 0x01 0x01 0x01 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x??}<EOP>

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
<Reserved - Set to 0 >				Speed		MaxPacketSize	

Figure 3-1 DevRqst Control Byte Bit Definitions

For control transfers to high-speed devices which include OUT data, Root 2 employs the PING/NYET protocol as defined by the USB 2.0 specification.

3.1.1 DevRqst and External Hubs

If the target device is a low speed device, and the device attached to Root 2's Root Port is a full-speed device, Root 2 will assume that the low-speed device being address is downstream of a full-speed hub. In this case, Root 2 will generate a full-speed PRE PID followed by a low-speed transfer for each packet issued to the device.

If the target device is a Full Speed or Low Speed device, and the device attached to the Root Port is a high-speed device, Root 2 will assume that the device being addressed is downstream of a high-speed hub, and will generate SPLIT transactions to access the target device.

3.1.2 DevRqst and SPLIT Transactions

If the target device is a low-speed or full-speed device, and is connected downstream of a high-speed hub which is attached to Root 2's Root Port, Root 2 must generate SPLIT transactions to the device. If the target device was configured in Auto Mode, and the <OVRD> bit is not set, Root 2 will use the information it acquired during initial configuration of the device to manage the SPLIT transactions. If the target device was not configured in Auto Mode, Root 2 will not have the necessary information - in particular the hub address and hub port ID - to generate the SPLIT transactions. In this case, or in a case where the target was configured in Auto Mode but the user wishes to override the Auto Mode settings, the <OVRD> bit must be set, and Root 2 uses the default Split information, which is programmed using the SplitDef command. Note that any time OVRD is set and a SPLIT transaction is to be generated, the default Split information will be used.

3.2 Port Power On/Off

The Power command allows the Controller to turn Root port Vbus power on or off. Note that the execution of this command *only* switches bus power on or off; it does not affect the Vbus voltage setting, which is controller by the SetVcc command. The format of the command is:

<SOP><CMD_Power><Action><EOP>

<CMD_Power> is the command ID, 0x02.

<Action> is:

0x0 = Power Off or

0x1 = Power On.

The Root 2 will respond with :

<SOP><RESP_Power><EOP>

<RESP_Power> is the response ID, 0x82.

Example:Turn Vbus power on.

Command: <SOP><0x02><0x01><EOP>

Response: <SOP><0x82><EOP>

Vbus power defaults to OFF.

3.3 Global Suspend

Allows the Controller to command a global suspend, causing the Root 2 to discontinue SOF generation and polling of downstream devices. The format of the command is:

<SOP><CMD_Suspend><EOP>

<CMD_Suspend> is the command ID, 0x03.

The Root 2 will respond with :

<SOP><RESP_Suspend><EOP>

<RESP_Suspend> is the response ID, 0x83.

3.4. Global Resume

Allows the Controller to command a global resume, causing the Root 2 to resume SOF generation and polling of downstream devices. The format of the command is:

<SOP><CMD_Resume><EOP>

<CMD_Resume> is the command ID, 0x04.

The Root 2 will respond with :

<SOP><RESP_Resume><EOP>

<RESP_Resume> is the response ID, 0x84.

3.5. Set VCC

The Vcc command allows the Controller to set the root port Vbus voltage. Note that this command *only* sets the Vbus voltage. The state of the power on/off switch does not change. Vbus voltage may be changed with Vbus power on or off. The format of the command is:

<SOP><CMD_VCC><Vcc_Value><EOP>

<CMD_VCC> is the command ID, 0x05.

<Vcc_Value> is a one byte value in the range 40 ~ 125 decimal, indicating the voltage value for root port Vcc as follows:

Vcc = 4.00 + <Vcc_Value>/100 in volts

e.g., a <Vcc_Value> of 40 decimal would yield a Vcc of 4.40V, and a Vcc_Value of 125 would yield a Vcc of 5.25V.

Root port Vcc can be set within the range 4.25V ~ 5.50V, and will be accurate to within +/- 2% over that. Root port Vcc defaults to 5.00V.

The Root 2 will respond with :

<SOP><RESP_VCC><EOP>

<RESP_VCC> is the response ID, 0x85.

Example: Set Vbus voltage to 5.00V.

Command: <SOP><0x05><0x64><EOP>
Response: <SOP><0x85><EOP>

3.6 Vbus Current Measurement

Root 2 provides two commands for measuring current drawn from Vbus by the device plugged into the Root Port. Low resolution current measurement is provided for backwards compatibilty with Root 1, and reports current to an resolution of 3mA. High resolution current measurement provides a much higher resolution of approximately 3uA.

Root 2 provides two levels of Vbus current limits, for protection of the Root 2 itself and the downstream devices. First, Root 2 hardware limits Root Port current draw to a maximum of approximately 700mA. Second, current draw in excess of approximately 600mA will cause Root 2 firmware to disable Vbus power within approximately 150mS.

3.6.1 Low Resolution Current Measurement

The VccMeasI command returns a low resolution measurement of current being drawn from the Root Port Vcc (Vbus). The format of the command is:

<SOP><CMD_VccMeasI><EOP>

<CMD_VccMeasI> is the command ID, 0x06.

The Root 2 will respond with :

<SOP><RESP_VccMeasI><I_Value><EOP>

<RESP_VccMeasI> is the response ID, 0x86. <I_Value> is a 1-byte unsigned value in the range 0~250 decimal, indicating the measured current drawn from the root port Vcc as follows:

Current Drawn = <I_Value> * 3mA.

Current measurement will be accurate to +/-2% +/- 3mA.

Example: Measure Vbus current - Low Resolution.

Command: <SOP><0x06><EOP>
Response: <SOP><0x86><0x50><EOP> ' Root 2 returns Vbus current of 240mA

3.6.2 High Resolution Current Measurement

The VbusCurrent command returns a high resolution measurement of current being drawn from the Root Port Vcc (Vbus). The format of the command is:

<SOP><CMD_VbusCurrent><EOP>

<CMD_VbusCurrent> is the command ID, 0x0E.

The Root 2 will respond with :

<SOP><RESP_VbusCurrent><I_ValueL><EOP>

<RESP_VbusCurrent> is the response ID, 0x8E. <I_ValueL> is a four-byte unsigned value indicating the measured current drawn from the root port Vcc as follows:

Current Drawn = <I_ValueL> * 2.96uA.

Current measurements in excess of 2.5mA will be accurate to +/-2%. Current measurements below 2.5mA will be accurate to +/-50uA.

Example: Measure Vbus current - High Resolution

Command: <SOP><0x0E><EOP>
Response: <SOP><0x8E><0x00><0x01><0x3E><0x70><EOP> ' Root 2 returns Vbus current of 241.3mA (0x0013E70 = 81,520 * 2.96uA)

3.7 Root_Config

The Root_Config command allows the Controller to set various Root 2 configuration parameters. The parameters which can be affected by this command are:

Enable/Disable Automatic Mode.
Enable/Disable Trigger Inputs.
Enable/Disable AutoRecovery Mode.
Enable/Disable built-in Monitor 1 support.
Change the baud rate of the Root 2 serial port.
Inhibit High-Speed Connect

The format of this command is:

<SOP><CMD_Root_Config><Parameter><Data><EOP>

<CMD_Root_Config> is the command ID, 0x07. <Parameter> is a byte value indicating the parameter to be affected, and <Data> is a byte value indicating the setting to be applied to the parameter, as follows:

<Parameter> = 0: Automatic Mode
<Data> = 0: Disable
<Data> = 1: Enable
<Parameter> = 1: Trigger Inputs
<Data> = 0: TrigIn1: Disabled TrigIn0: Disabled
<Data> = 1: TrigIn1: Disabled TrigIn0: Enabled

<Data> = 2; TrigIn1: Enabled TrigIn0: Disabled
<Data> = 3: TrigIn1: Enabled TrigIn0: Enabled
<Parameter> = 2: Autorecovery Mode
<Data> = 0: Disable
<Data> = 1: Enable
<Parameter> = 3: Monitor 1 LED Status Support
<Data> = 0: Disable
<Data> = 1: Enable
<Parameter> = 4: Monitor 1 Push-button Support
<Data> = 0: Disable
<Data> = 1: Enable
<Parameter> = 5: Set Baud Rate
<Data> = 0: 19200 baud
<Data> = 1: 38400 baud
<Data> = 2: 57600 baud
<Data> = 3: 115200 baud
<Data> = 4: 230400 baud
<Data> = 5: 460800 baud
<Parameter> = 6: Inhibit High Speed Connect
<Data> = 0: High-speed connect enabled
<Data> = 1: High-speed connect disabled

At power up:

Automatic Mode defaults to Enabled,

both Trigger inputs default to Disabled,

AutoRecovery defaults to Disabled,

Monitor 1 LED Status and Monitor 1 Push-button Support default to Disabled,

Baud Rate defaults to 115200 baud.

The Root 2 will respond with :
<SOP><RESP_Root_Config><EOP>

<RESP_Root_Config> is the response I D, 0x87.

Example: Enable Both trigger Inputs.
Command: <SOP><0x07><0x01><0x03><EOP>
Response: <SOP><0x87><EOP>

3.7.1 Enable/Disable Automatic Mode

Automatic Mode is described in section 2.7. With Automatic Mode disabled, the user application must handle all device interactions. Root port and external hub status must be polled to detect device connects, devices must be reset enumerated and configured explicitly using Interactive Commands.

3.7.2 Enable/Disable External Trigger Inputs

The external Trigger Inputs are discussed in section 2.3. The Root_Config command allows the two trigger inputs to be individually enabled and disabled. In Interactive Mode, an Asynchronous Response will be generated by Root 2 in response to a trigger on an enabled input. If the trigger input is disabled, inputs on that trigger line will be ignored. In RootScript mode, a trigger on an enabled input will provide status to the RS_Check command, allowing the trigger input to be detected by the script.

3.7.3 Autorecovery Enable/Disable

The AutoRecovery mechanism, when enabled, causes Root 2 to automatically attempt to recover from a downstream overcurrent condition. When an overcurrent condition is detected on the Root Port, or on the downstream port of an external hub, power to the port is disabled in order to protect the host or hub. Enabling AutoRecovery causes power to any ports which have been disabled due to overcurrent to be reenabled on a two-second period. If the overcurrent condition has been removed, the port will then remain powered. If the overcurrent condition still exists, the port will be shut down again by the overcurrent detection circuitry. This mechanism effectively results in an automatic two-second retry of any overcurrent-disabled ports.

3.7.4 Monitor 1 LED Status and Pushbutton Support

Root 2 provides built-in support for Go-NoGo testing using the Monitor 1 handheld test controller. Monitor 1 provides eight LED status indicators which can be driven by Root 2's 8-bit data port, and two push-buttons which will drive Root 2's two trigger inputs. When LED Status support is enabled, Root 2 will report USB status on the Monitor 1 status indicators. When Pushbutton Support is enabled, the operator can use the pushbutton inputs on Monitor 1 to force a USB Reset or to cycle Vbus power. Go-NoGo testing with Monitor 1 is described in detail in a separate Application Note entitled *Go / No-Go Testing Using Monitor 1*, and is available on the Root 2 support CD-ROM or from the RPM web site (http://www.rpmsys.com).

3.7.5 Programmable Baud Rate

Root 2 allows the baud rate of its serial communications port to be reprogrammed. At power-on, the serial port baud rate will default to 115200 baud. Subsequently, the RootConfig command can be used to set the baud rate to any of the values enumerated above. Note that, when the RootConfig command is issued to change baud rates, Root 2 will issue the command response at the current baud rate, before switching to the new one. The host application must therefore be certain to wait until the response has been completely received before switching to the new baud rate.

3.7.6 Inhibit High Speed Connect

Enabling high-speed inhibit prevents Root 2 from conducting high-speed device detection signaling (Chirping) during USB Reset, essentially causing Root 2 to behave as a non-high-speed capable host (a USB 1.1

compliant host). In the absence of high-speed detect signaling, a high-speed capable device will connect at full speed. This feature allows testing of a high-speed capable device in its full-speed configuration.

3.8 USB_Reset Command

The USB_Reset command issues a bus Reset on the root port. In Automatic Mode, this command will result in all device connections being terminated. Asynchronous Responses indicating device disconnects will not be generated, however new connect messages will be generated as each device is re-enumerated following the USB Reset. The format of this command is:

<SOP><CMD_USB_Reset><EOP>

<CMD_USB_Reset> is the command ID, 0x08.

When the Reset has been completed, the Root 2 will respond with:

<SOP><RESP_USB_Reset><EOP>

<RESP_USB_Reset> is the response ID, 0x88.

3.9 DevTrans Command

This command allows a discrete transaction to be initiated to a user-defined Address and Endpoint. Unlike DevRqst, which completes an entire control transfer in a single command, DevTrans deals with only a single transaction at a time. That is, it issues one IN, OUT or Setup and accepts the device's immediate response. DevTrans is typically used for non-control transfers, such as INs and OUTs to bulk, interrupt or isochronous data endpoints. Control transfers are much more easily handled using the DevRqst command. Because it offers packet level control, however, it may be desirable to use DevTrans to generate control transfers in special situations, such as where the user desires to step through the transfer one transaction at a time.

Chapter 8 of the USB Specification describes the construction and handling of transactions on the USB. Note that CRC's and PID check bits are generated and checked automatically by the Root 2, and are not required to be provided by the user. Likewise, Root 2 provides the necessary handshaking to conduct the packet on the bus. The user is required only to provide the address, endpoint, transaction PID and any data. Valid token PIDs for the DevTrans command are IN, OUT, SETUP and PING. PING is only valid for high-speed devices. Allowable data PIDs are DATA0, DATA1, DATA2 and MDATA. The PID values can be

Table 3-2: USB PID Values

Token PIDs		Data PIDs	
Value	**PID Name**	**Value**	**PID Name**
0x1	OUT	0x3	DATA0
0xD	SETUP	0xB	DATA1
0x4	PING	0x7	DATA2
0x9	IN	0xF	MDATA

found in Table 8-1 of the USB Specification, but are listed below, in Table 3-2, for convenience.

Because of the timing variations associated with host communications via the serial or Ethernet port, new transactions are typically staged by Root 2 to be issued immediately following the next SOF, in order to prevent overrunning the frame or microframe. This typically results in a best-case throughput of one transaction per (1mS) frame for low-speed and full-speed devices, or one transaction per microframe (125uS) for high-speed devices. For RootScript applications, The DevTrans command provides a mechanism which allows the user to override this convention, allowing multiple transactions per frame to be conducted. This feature is available only in Script mode, and requires the user to ensure that the transactions fit within the current frame without overrun.

It is not recommended that DevTrans be used to support high-speed, high-bandwidth interrupt and isochronous transactions, in which multiple transactions per microframe are issued. Please refer to the BlockTrans command for support of these functions.

The format of the DevTrans command is:

<SOP><CMD_DevTrans><Address><Ept><PID><Control>{<DataPID><OutputData>}<EOP>.

<CMD_DevTrans> is the DevTrans command ID, 0x09.

<Address> is 1 byte = the USB address, 0~127, of the target device.

<Ept> is 1 byte = the target endpoint for the transaction.

<PID> is 1 byte = the 4-bit token PID which will be used in the token packet (IN, OUT, SETUP, PING) .

The <Control> parameter is 1 byte and provides information regarding the details of the data flow for the transaction. Figure 3-2 details the DevTrans Control byte.

Bit 0 of the <Control> parameter is the <Direction> bit, and indicates whether the direction of the data portion of the transaction will be In (direction = 0) or Out (direction = 1). If <Direction> = 1, such as it would for a SETUP or OUT transaction, <DataPID> will be the data PID, DATA0 or DATA1, to be transmitted with the data packet. Note that Root 2 performs no checking for data toggling when using the DevTrans command. This is the responsibility of the user's application. <OutputData> is the data packet to be transmitted in conjunction with a Setup or Out command, and may be from 0 to 1024 bytes in length. <DataPID> and <OutputData> are not included if <Direction> = 0.

Bits 1 and 6 of the <Control> parameter are the <FullSpeed> and <HighSpeed> bits, respectively. If <HighSpeed> is set (=1), it indicates to Root 2 that the function being addressed is a high-speed device. If <FullSpeed> is set (=1), it indicates to Root 2 that the function being addressed is a full-speed device. If both <FullSpeed> and <HighSpeed> bits are =0, then the target device is assumed to be low speed.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Immed	HighSpeed	FSplit	LoopBack	Transfer Type		FullSpeed	Direction

Figure 3-2 DevTrans Control Byte Bit Definitions

Bits 2 and 3 of the <Control> parameter indicate the transfer type. This field is used for isochronous transfers and for SPLIT transactions. It is encoded as follows:

Bit 3	Bit 2	Transfer Type
0	0	Control
0	1	Isochronous
1	0	Bulk
1	1	Interrupt

Bit 4 of the <Control> parameter is the <LoopBack> bit. If this bit is set (=1), device data is read from and written to the loopback buffer in Root 2 memory, and is not taken from or returned to the host.

Bit 5 of the <Control> parameter is the <FSplit> bit. If this bit is set (=1), Root 2 will use the default SPLIT settings for Hub Address and Hub Port, set by the SplitDef command. Otherwise, Root uses Auto Mode knowledge of the target device to generate the SPLIT. This bit applies only to SPLIT transactions.

Bit 7 of the <Control> parameter is the <Immed> bit. When set, this bit causes the DevTrans packet to be issued immediately, without waiting for the next SOF. This bit is valid only in a RootScript.

The device transaction is conducted on the bus, and the Root 2 responds with the following:

<SOP><RESP_DevTrans><RespStatus>{Device Response}<EOP>

<RESP_DevTrans> is the DevRqst response ID, 0x89. <RespStatus> is the response status as indicated in Table 3-1. In the case of an IN, the {Device Response} is the data packet returned by the target device. In the case of an Output command (Out or Setup), the device's response to the command is completely defined in the RespStatus byte, and {Device Response} is not present.

3.9.1 DevTrans Operation with External Hubs

If the target device of the DevTrans command is designated as low-speed (<FullSpeed> and <HighSpeed> bits = 0), and the device attached to the Root 2's Root Port is a full speed device, Root 2 will assume that the device being addressed is a low-speed device attached to the downstream port of a full-speed hub, and will generate a low-speed preamble (PRE PID) to indicate to the hub and other full-speed devices downstream that a low-speed packet follows.

If the target device of the DevTrans command is designated as a low-speed or full-speed device, and the device attached to Root 2's Root Port is a high-speed device, Root 2 will assume that the device being addressed is a low-speed or full-speed device attached to the downstream port of a high-speed hub, and a SPLIT transaction will be generated. Root 2 will complete the entire split transaction prior to returning a response (see following section).

3.9.2 DevTrans and SPLIT Transactions

If the target device for the transaction is a low-speed or full-speed device downstream of a high-speed hub

attached to Root 2, DevTrans must generate a SPLIT transaction to access the device. If the device was configured in Auto Mode, Root 2 will have knowledge of the hub address and the port on that hub to which the target device is connected, and it will use that knowledge to complete the SPLIT transaction transparently to the user application. If the device was not configured in Auto Mode - that is, it has not yet been configured, or was configured manually using DevTrans or similar commands - then the user must provide the hub address and hub port information for the target device. This is done by first programming the default SPLIT information (hub address and hub port) using the SplitDef command, then setting the <FSplit> bit in the DevTrans <Control> byte to force Root 2 to use the default SPLIT information.

In generating SPLIT transactions, Root 2 must be aware of the type of transaction being issued. This information is forwarded to the hub, and is used in determining how to carry out the SPLIT transaction.

3.9.3 DevTrans and Immediate Mode

Under normal circumstances (Immed bit = 0), Root 2 always begins a new bus transaction immediately after a SOF. This ensures that the transaction will not overrun the current frame. The <Immed> bit allows RootScripts to circumvent this safeguard. In using the <Immed> bit, the script writer must take responsibility for avoiding the condition in which traffic is generated on the bus which overruns the end of the frame or microframe. In general, the Root 2 has an overhead of approximately 50uS associated with each bus transaction. This includes the time required to fetch a new script command, setup and conduct the transaction and verify the response, exclusive of the actual data transmission. One could, for example, expect to issue up to eight (64-byte) full-speed bulk transfers per frame as follows:

```
{Packet 1} Immed = 0; DevTrans - OUT - 64 bytes      // sync to SOF
{Packet 2} Immed = 1; DevTrans - OUT - 64 bytes      //transfer immediately
...
{Packet 8} Immed = 1; DevTrans - OUT - 64 bytes      //transfer immediately
{Packet 9} Immed =0 ; DevTrans - OUT - 64 bytes      //resync to SOF
{Packet 10} Immed = 1; DevTrans - OUT - 64 bytes     //transfer immediately
...
```

In this example, the total bus time for each transfer is approximately 60uS, including both token and data packets, and accounting for possible bit stuffing. Adding 50uS for Root 2 overhead gives approximately 110uS per transfer. After the eighth transfer, it is necessary to wait again until the start of the next frame to avoid overrunning the end of frame. NOTE that the <Immed> bit can only be used in RootScripts. It is ignored in Immediate Mode DevTrans commands. In most cases, it is easier and safer to use the BlockTrans command to issue multiple-transaction-per-frame (or microframe) transfers.

3.9.4 DevTrans and Automatic Mode

Care must be taken in using the DevTrans command in conjunction with Automatic Mode, since it allows conditions to be created which will undermine the Root 2's knowledge of the state of downstream devices, causing it to potentially mishandle packets to or from the device. In particular, issuing commands via DevTrans which cause a device's address to change, or which cause it to operate using in a configuration different from that initially enabled by Automatic Mode can cause unforeseen problems.

3.9.5 DevTrans and Loopback Mode

When Bit 4 is set in the DevTrans Control byte, the transaction is directed to the loopback buffer. If the transaction is an IN, data received from the device is stored in the loopback buffer in Root 2 memory, and is not returned to the host. If the transaction is an OUT, no data is taken from the host command stream. Instead, data is taken from the loopback buffer, and the size of the data transfer will be the same as the size of the IN transfer which filled the loopback buffer. Loopback mode is intended to allow data to be read from an IN endpoint, then subsequently written to an OUT endpoint without having to move the data to or from the host, making it a much faster transaction. For example, data may be read from an isochronous audio IN function (e.g., a digital microphone), then written to an isochronous audio OUT function (e.g., a digital speaker) in the same frame. Note that, since the loopback buffer is a separate buffer, other, non-loopback traffic can be issued between loopback commands.

3.10 Data Port and Trigger Out

The DataPort command is used to strobe data onto the external data port. The data can be written using one of two methods: direct or masked. Using the direct method, the user simply supplies an 8-bit data value which is written to the data port. Using the masked method, the user supplies an 8-bit AND mask, and an 8-bit OR mask. The AND mask is first applied to the current data port data value. The result of this operation is then OR'd with the OR mask, and the resultant data is written to the data port.

Direct Method: DataPort <- NewData

Masked Method: DataPort <- ((DataPort & ANDMask) | ORMask)

The data written to the data port will be driven onto the Root 2's parallel data lines, and the TrigOut strobe will be toggled as described in section 2.4. Note that, while the TrigOut strobe only toggles once per execution of the command, the value written to the data port will continue to be driven on the outputs until a new value is written.

This command is useful in generating strobed data outputs, such as may be used as an input to a logic analyzer, using the TrigOut strobe as a clock. The command is also useful in generating static outputs on the data lines to be used, with the proper interface circuitry, in driving external indicators or controlling external actuators.

The format of the direct command is:

<SOP><CMD_DataPort><Data_Value><EOP>

<CMD_DataPort> is the command ID, 0x0A.

<Data_Value> is a one byte value to be written to the data port.

The format of the masked command is:

<SOP><CMD_DataPort><AND_Mask><OR_Mask><EOP>

<CMD_DataPort> is the command ID, 0x0A.

<AND_Mask> is a one byte AND mask.

<OR_Mask> is a one byte OR mask.

Root 2 distinguishes between the direct and masked methods simply based on the length of the command string. In either case, the Root 2 will respond with :

<SOP><RESP_DataPort><EOP>

<RESP_DataPort> is the response ID, 0x8A.

Example: Direct Method

```
Command: <SOP><0x0A><0x55><EOP>          'Parallel data output after command = 0x55
Response: <SOP><0x8A><EOP>
```

Example: Masked Method. Value of data port prior to command = 0x0F

```
Command: <SOP><0x0A><0x0C><0x81><EOP>    'Parallel data output after command = 0x8D
Response: <SOP><0x8A><EOP>
```

3.11 Read Root Hub Status

The Get_RootStatus command returns a byte containing various information regarding the status of the Root hub. The format of the command is:

<SOP><CMD_Get_RootStatus><EOP>

<CMD_Get_RootStatus> is the command ID, 0x0B.

The Root 2 will respond with :

<SOP><RESP_Get_RootStatus><Data_Value><EOP>

<RESP_Get_RootStatus> is the response ID, 0x8B.

The one byte value returned in <Data_Value> is encoded as follows:

Bit 0 - Low-speed Connect Status
=1 if a low-speed device is connected to the Root Port

Bit 1 - Full-speed Connect Status
=1 if a full-speed device is connected to the Root Port

Bit 2 - Power State
0 Root Port Power Off
1 Root Port Power On

Bit 3 - Suspend State
0 Root Port Active (not suspended)
1 Root Port Suspended

Bit 4 - Root Port Enabled State
0 Root Port Disabled
1 Root Port Enabled

Bit 5 - AutoRecovery Status

0 AutoRecovery Disabled
1 AutoRecovery Enabled

Bit 6 - High-speed Connect Status
=1 if a high-speed device is connected to the Root Port

Bit 7 - Not Used

The Power and Suspend State values returned by this command can be useful in the event that these factors change as a result of something other than a command from the test controller. For instance, in the event of an overcurrent condition on the Root port, Vbus power will be switched off. Similarly, if the Root port is Suspended by command from the Controller, it can be brought out of Suspend by Resume signaling from a downstream device.

Bit 4, the Root Port Enabled status, will typically be true (enabled) when power is on and a device is connected and operating properly. Certain circumstances, such as a device babbling on the bus, will cause the Root 2 host controller to disable the root port. In this instance, it is necessary to disable then reenable root port power to reenable traffic on the root port.

Bit 5 reflects the state of the AutoRecovery feature, which is described in Section 3.7.

3.11.1 RootStatus Connect Bits

The Connect bits, Bit 0, Bit 1 and Bit 6, will be set =1 to indicate connection of a low-speed, full-speed or high-speed device, respectively. If no device is connected, all three bits will be =0. If a device is connected, but has not yet been reset, its speed will be unknown, and *all three* connect bits will be =1.

3.12 Block Transfer

The BlockTrans command allows the operator to automate the implementation of large transfers to or from a device. In contrast to DevTrans, which requires the Controller or RootScript to conduct every transaction in a transfer, BlockTrans is capable of completing a large transfer, consisting of many transactions, without intervention. BlockTrans can be used to perform tightly-packed transfers consisting of multiple transactions per frame (or microframe) at all bus speeds and in all transfer modes, including high-speed, high-bandwidth interrupt and isochronous transfers. In addition, BlockTrans supports a looping mode, in which a data transfer up to 524,280 Bytes in size can be issued repeatedly. The number of transactions to be performed per frame or microframe is

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
-	-	-	-	-	-	NakEnd	Loop

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	HighSpeed	FSplit	-	Transfer Type		FullSpeed	-

Figure 3-3 BlockTrans Control Word Bit Definitions

programmable. BlockTrans is intended for interrupt, bulk or isochronous transfers to or from data endpoints. The DevRqst command should be used to perform this function for control transfers.

The format of the BlockTrans command is:

<SOP><CMD_BlockTrans><Addr><Ept><PID><Control><DataPID><ServiceInterval><MaxPacketSize><PktMult><DataLen>{<OutputData>}<EOP>

<CMD_BlockTrans> - 1 byte - is the BlockTrans command ID, 0x39.

<Address> - 1 byte - is the USB address, 0~127, of the target device.

<Ept> - 1 byte - is the target endpoint for the request.

<PID> - 1 byte - is the starting Token PID (IN, OUT or PING).

<Control> - 2 bytes - is a 16-bit control word. The format of <Control> is shown in Figure 3-3, and its bits are defined below.

<DataPID> - 1 byte - is the starting PID to be used for non-isochronous OUT transfers (DATA0 or DATA1).

<ServiceInterval> - 2 bytes -16 bit value which sets the frame or microframe interval between transactions

<MaxPacketSize> - 2 bytes - Default packet size for data transfer (1 ~ 1024)

<PktMult> - 1 byte - Indicates number of additional transactions per frame or microframe (0 ~ 31)

<DataLen> - 4 bytes - Length of transfer data in bytes

<Output Data> - Data to be sent to the device for OUT transfers

3.1.2.1 BlockTrans Control Word

The BlockTrans control word bits are defined as follows:

Bit 1: FullSpeed
This bit should be set (=1) if the target device is a full-speed device.

Bits 3,2: Transfer Type
These bits define the type of transfer being performed, and are used for Isochronous and SPLIT transfers.

Bit 3	Bit 2	Transfer Type
0	0	Control
0	1	Isochronous
1	0	Bulk
1	1	Interrupt

Bit 5: FSplit
This bit is set (=1) to force Root 2 to use the default Split information programmed by the SplitDef command.

Bit 6: HighSpeed
This bit is set (=1) if the target device is a high-speed device.

Bit 8: Loop
This bit is set (=1) to cause the transfer to be repeated until it is terminated by the StopTrans command or an error.

Bit 9: NakStop
THis bit is set (=1) to cause the transfer to be aborted if any transaction is NAK'd by the device.

3.12.2 BlockTrans PID Handling

<PID> is the starting token PID. On an OUT transfer to a high-speed device, the PID can be set either to PING or OUT, identifying the first token to be used by Root 2 when the transfer is initiated. Root 2 will then automatically handle OUT data flow using the OUT/PING/NYET protocol. For low-speed and full-speed transfers, only IN and OUT tokens are valid.

<DataPID> is the data PID that will be issued with the first OUT transaction of the transfer, or will be expected on the first IN transaction of the transfer. For bulk and interrupt transfers, <DataPID> can be set to either DATA0 or DATA1, as required by the state of the device, and subsequent data toggling will be handled by Root 2.

For full-speed isochronous transfers, and high-speed isochronous transfers which do not perform multiple transfers per microframe, USB does not support data toggle sequencing, and the DATA0 PID is typically used. For high-speed, high-bandwidth isochronous transfers, data toggle sequencing is used, and the sequencing is different for INs and OUTs. For high-speed high-bandwidth OUT isochronous transfers, the starting data PID is should be set to MDATA, and Root 2 will then manage the data toggle sequencing for the remainder of the transfer. For high-speed high-bandwidth IN isochronous transfers, <DataPID> should be set to DATA2 if <PktMult> is = 2, or DATA1 if <PktMult> is = 1. Root 2 will then manage the data sequencing for the remainder of the transfer.

3.12.3 Bulk and Interrupt Transfers using BlockTrans

For IN transfers from bulk and interrupt endpoints, the token PID should always be IN. For OUT transfers to low-speed and full-speed devices, the token PID should always be OUT. (If the target device is a low-speed or full-speed device downstream of a high-speed hub connected to Root 2, see the section below on Split Transactions using BlockTrans). For OUT transfers to a high-speed bulk or interrupt endpoint, the starting PID may be set to either OUT or PING. Setting <PID> to PING will cause the Root 2 to start the transfer by issuing PINGs to the endpoint until it receives an ACK, at which time it will switch to OUT and begin transferring data. If, at any time, Root 2 receives a NYET or NAK response from the device, it will again switch back to a PING token.

The <ServiceInterval> and <PktMult> parameters determine the flow of data packets throughout the transfer. <ServiceInterval> determines the number of unused frames, for low-speed and full-speed devices, or microframes for high-speed devices, between transactions. A <ServiceInterval> value of 0 will result in packets

being transferred every (micro)frame (i.e., 0 unused frames between transfers). <PktMult> determines the number of additional transactions (that is, in addition to the first one) to be performed in a frame, for low-speed and full-speed devices, or in a microframe for high-speed devices. Together, these two parameters define the transaction flow as <PktMult> + 1 transactions every <ServiceInterval>+1 (micro)frames. Here are some examples:

<ServiceInterval>	<PktMult>	Transfer Flow
0	0	1 packet per frame or microframe
1	0	1 packet every other frame or microframe
2	0	1 packet every third frame or microframe
0	1	2 packets every frame
0	10	10 packets every frame
1	10	10 packets every other frame

<MaxPacketSize> defines the default packet size for each transaction. For OUT transfers, all transactions will contain <MaxPacketSize> bytes of data, with the exception of the last packet of the transfer, which may be shorter that <MaxPacketSize> if the entire transfer is not an integer multiple of <MaxPacketSize> bytes. For IN transfers, all received transactions are expected to contain <MaxPacketSize> bytes, with the exception of the last transaction of the transfer which may be shorter than <MaxPacketSize> if the entire transfer is not an integer multiple of <MaxPacketSize> bytes. When receiving, any received packet with a data payload shorter than <MaxPacketSize> will be interpreted as a short packet, and will cause the transfer to be terminated.

For interrupt endpoints, the USB Specification limits the number of transactions to a given endpoint per microframe to three for high-speed, high-bandwidth interrupt endpoints, and to one transaction per frame for full-speed endpoints. For bulk transfers, there is no specified limit to the number of transactions per frame to a given endpoint, other than the implicit limit imposed by the available bus bandwidth. Root 2 does not impose any limitations, other than that it will support a maximum of 32 transactions per (micro)frame (<PktMult> = 31). It is the responsibility of the programmer to ensure that the number of packets requested per frame will fit in the frame. That is, the user must ensure that <PktMult> transactions, each containing <MaxPacketSize> bytes of data, plus token packets, CRC's, bit-stuffing, etc., will fit together with a SOF in the 1mS frame or 125uS microframe without overrunning the end of frame. Remember to include both the host and device portions of each transaction, as well as interpacket delays, in the calculation. Interpacket delays between back-to-back transmissions from Root 2, for instance, between a token and data OUT packet, are approximately 4 bit times. Interpacket delays between device transmission and the subsequent Root 2 transmission is also approximately 4 bit times. Here is an example for a full-speed bulk OUT transfer with <MaxPacketSize> = 64. The numbers shown are maximum bit times with no bit stuffing. The numbers in parentheses show maximum bit times with maximum bit stuffing.

Host Transmit	
Sync	8
Token PID	8
Token Packet	24 (28)
EOP	2
Gap	4
Sync	8
Data PID	8
Data	512 (597)
CRC	16 (19)

EOP	2
Turnaround Delay	13
Device Transmit	
Sync	8
Handshake PID	8
EOP	2
Gap	4
TOTAL	627 (719)

There are 12,000 bit times available in a full-speed frame. Approximately 40 bits times are lost to the SOF packet, leaving 11,960 bit times for data. This allows time for a maximum of 16 transactions per frame with maximum bit stuffing, or 19 transactions per frame with no bit stuffing.

3.12.4 Isochronous Transfers using BlockTrans

The <Isoch> bit in the control word must be set (=1), for all isochronous transfers. The token PID, <PID>, should be set to IN or OUT, depending upon the direction of the transfer. For full-speed isochronous transfers, or high-speed isochronous transfers which do not perform multiple transfers per microframe, no data sequencing is supported by USB, so the data PID is typically set to DATA0. For high-speed, high-bandwidth isochronous transfers, the starting data PID should be set as described in section 3.12.1.

<ServiceInterval> and <PktMult> are defined in the same way as

For OUTs, <MaxPacketSize> should be set to the number of bytes to be issued in each transaction. For full-speed transfers and high-speed transfers which perform only one transaction per microframe (<PktMult> = 0), one transaction

3.12.5 Transfer Initiation and Termination

Since BlockTrans commands can take a long time to execute, the command/response mechanism for this command is different than for most Root 2 commands. When a BlockTrans command is issued, Root 2 will return a response immediately. An interactive user application must then use the BlockTransStatus command to determine when the transfer has completed, and what the completion status is. If the application is a RootScript, it must use the RS_Check command to determine when the transfer is complete. The Root 2 response to the BlockTrans command is:

<SOP><RESP_BlockTrans><RespStatus><EOP>

<RESP_DevTrans> is the BlockTrans response ID, 0xB9. <RespStatus> is the response status as indicated in Table 3-1. Since BlockTrans returns its response before the transfer actually starts, its <RespStatus> will typically be Success. The command will fail under certain conditions, for instance, if it receives bad parameter information, or because another BlockTrans command is still active.

For all transfer modes, the <DataLen> parameter defines the total length of data expected for IN transfers, and the transfer will be terminated when <DataLen> bytes of data have been received. For OUT transfers, the data transfer will be defined by the smaller of <DataLen> and the number of bytes contained in <OutputData>,

and the transfer will be terminated when that number of bytes have been transferred. If the <Loop> bit in the <Control> word is set (=1), Root 2 will immediately restart the transfer once it has completed.

Any error condition (e.g., STALL, CRC error, Data Toggle error, no response from device) will cause the transfer to be terminated, regardless of the state of the <Loop> bit.

For interrupt and bulk modes, setting the <NakEnd> (=1) in the <Control> word will cause the transfer to be terminated if a Nak received from the device on any transaction, regardless of the state of the <Loop> bit.

Finally, the transfer can be forcibly stopped by issuing the StopTrans command. Once the command has been terminated, for whatever reason, BlockTransStatus can be used to examine its termination conditions If the transfer was an IN, ReadTrans can be used to retrieve the data returned by the device..

3.12.6 BlockTrans Looping

If the <Loop> bit is set in <Control>, the Root 2 will issue the same block transfer continuously until it encounters an error, until if encounters a Nak if the <NakEnd> bit is set, or until it is halted by a StopTrans command. If the transfer is an OUT, the same data will be transferred to the device on each pass of the loop. If the transfer is an IN, data read from the device will continuously overwrite the transfer buffer. The data left in the transfer buffer when the transfer is stopped can be read using the ReadTrans command.

3.13 Block Transfer Status (BlockTransStatus Command)

The BlockTransStatus command is used by an interactive application to determine when a BlockTrans command has completed, to examine its termination conditions, and to retrieve data returned from the target device. The format of the BlockTransStatus command is:

<SOP><CMD_BlockTransStatus><EOP>.
<CMD_BlockTransStatus> is the command byte, 0x38.

Root 2 responds with :

<SOP><RESP_BlockTransStatus><ExecStatus><TransStatus><EOP>
<RESP_BlockTransStatus> is the response byte, 0xB8.
<ExecStatus> - Execution Status

= 0: Block transfer is actively running

= 1: Block transfer is complete

<TransStatus> - Transaction Completion Status. This byte is valid only if <ExecStatus> = 1, and will contain the completion status of the block transfer, as defined in Table 3-1.

3.14 Stop Block Transfer (StopTrans Command)

StopTrans is issued by the Controller to forcibly stop a BlockTrans. The format of the StopTrans com-

mand is:

<SOP><CMD_StopTrans><Mode><EOP>.
<CMD_StopTrans> is the command byte, 0x3A.
<Mode> - indicates at what point the transfer should be halted
= 0: Halt at completion of transfer. This mode is useful to stop a looping transfer on a transfer boundary.
= 1: Halt immediately. This mode will halt immediately, or at the completion of the current transaction, if one is active.

Root 2 responds with :

<SOP><RESP_StopTrans><EOP>
<RESP_StopTrans> is the response byte, 0xBA.

Once the StopTrans command has been issued, the Controller must continue to poll, using the BlockTransStatus command, to determine when the block transfer has actually completed, and to acquire termination status.

3.15 Read Block Transfer (ReadTrans Command)

ReadTrans is used to read data from the block transfer buffer after a BlockTrans IN has completed. The format of the ReadTrans command is:

<SOP><CMD_ReadTrans ><Mode><EOP>.
<CMD_ReadTrans> is the command byte, 0x3B.

Root 2 responds with :

<SOP><RESP_ReadTrans >{Read Data}<EOP>
<RESP_ReadTrans > is the response byte, 0xBB.
{Read Data} is the data left in the transfer buffer at the completion of the BlockTrans IN.

The amount of data returned will be the amount actually returned by the device.

3.16 Program Default Split Information (SplitDef Command)

The SplitDef command programs the default hub address and hub port select to be used by Split transactions for which Root 2 does not have hub and port information. When a low- of full-speed device is connected downstream of a high-speed hub, transactions to and from that device must be issued as Split transactions. Split transactions require the bus address of the hub, and the port ID on the hub, to which the device is attached. If the device is configured automatically by Root 2 in Automatic mode, Root 2 acquires this information and can use it to generate Split transactions as necessary in future data transfer commands (DevRqst, DevTrans and BlockTrans commands). If Automatic Mode is disabled when the device is connected, the application must provide default information to be used by the data transfer commands. SplitDef allows this information to be provided by the application. The format of the SplitDef command is:

<SOP><CMD_SplitDef><Hub Address><PortID><EOP>
<CMD_SplitDef> is the SplitDef command ID, 0x37.
<HubAddress> is the bus address, 1 ~ 127 of the hub to which the target device is connected
<PortID> is the hub port ID for the port to which the target device is connected

Root 2 responds with:

<SOP><RESP_SplitDef><EOP>
<RESP_SplitDef> is the SplitDef response ID, 0xB7.

4. Asynchronous Responses

Asynchronous responses are generated by Root 2 to call attention to asynchronous events. The Transmission codes used in Asynchronous Responses are unique, allowing them to be easily differentiated from command responses.

4.1 Connect Event

When a new device connect is detected in Automatic mode, Root 2 resets the device, enumerates it and places it in its initial configuration. At this time, a Connect Event is issued to the Controller. The address assigned to each device is predictable, to facilitate the writing of test applications. A device plugged into the Root port will always be assigned address 2 by the host controller in Automatic mode. If an external hub is connected to the Root port, the address of the hub will be 2, and devices connected to the downstream ports of the hub will be enumerated beginning with the hub's downstream port 0 at address 3, port 1 at address 4, etc. Note that the address assignment in Automatic mode is determined by the physical port of the hub, regardless of the order in which devices are attached. For instance, a devices attached to physical ports 1 and 2 of an external hub will be assigned addresses 4 and 5, respectively, regardless of the order in which the devices are connected. Note that Root 2 does not support multiple levels of hubs. Hub support is limited to the hub connected directly to Root 2.

When the device subsequently disconnects, a Connect Event is issued to the Controller indicating that the device has disconnected. The disconnecting device is identified by the address which was assigned to it at connect time.

The format of the Connect Event is:

<SOP><RESP_Connect><Action><Address>{<Device Class><Vendor ID><Product ID>}<EOP>

<RESP_Connect> is the response ID, 0x90.

<Action> is a byte value 0x0 for a connect, or 0x1 for a disconnect.

<Address> is the USB address assigned to the device by the host controller.

<Device Class>, <Vendor ID> and <Product ID> are provided only in the case of a Connect, and are taken directly from the device's Device Descriptor.

<Vendor ID> and <Product ID> are two-byte values, transmitted LSB first. <Device Class> is a 1-byte value.

4.2 Status Event

A Status Event is an asynchronous response generated by the Root 2 in Automatic Mode to inform the Controller of a status change, other than a connect or disconnect, on the bus. This type of event is typically generated as the result of polling the status-change endpoint of a hub on the root port. The format of the re-

sponse is:

<SOP><RESP_Status><Hub Address><Port #><Status_Value><EOP>

<RESP_Status> is the response ID, 0x91. <Hub Address> is the enumerated address of the hub reporting the status, and <Port #> is the number of the port on that hub reporting the status change. <Status_Value> is the 16-bit Port Status value elicited from the hub for that port, per the USB Specification Rev 1.1, Table 11-5.

4.3 Data Event

In Automatic Mode, Root 2 automatically polls any interrupt IN endpoints defined in initial configuration for each device. Examples are a standard HID keyboard's data endpoint or a hub's status change endpoint. A Data Event is an asynchronous response generated by the Root 2 in Automatic Mode to inform the Controller that data has been returned by a downstream device in response to this polling. The format of the response is:

<SOP><RESP_Data><Address><Endpoint>{Data}<EOP>

<RESP_Data> is the response ID, 0x92.

<Address> and <Endpoint> are 1-byte values indicating the address and endpoint reporting the data

{Data} is the data reported from the device.

4.4 Error Event

An Error Event is an asynchronous response generated by the Root 2 in Automatic Mode to inform the Controller that an error was encountered in communicating with a downstream device. The format of the response is:

<SOP><RESP_Error><Address><Endpoint><Error><EOP>

<RESP_Error> is the response ID, 0x93.

<Address> and <Endpoint> are 1-byte values indicating the address and endpoint on which the error was encountered.

<Error> is a 1-byte value indicating the type of error encountered as shown in Table 3-1.

The majority of the error status responses are self-explanatory. The following paragraphs detail those less obvious responses.

A *PID Error* indicates that a received PID was either an invalid value, or its check bits did not match.

A *Babble Error* is returned if a device begins transmitting on the bus unexpectedly, or continues to transmit after an EOP.

The *Short Packet* response indicates that a received data packet was shorter than the shortest valid packet. The shortest valid data packet consists of a SYNC, a data PID and a 16-bit data CRC. Any received data packet which is shorter than this minimum will be flagged with this error.

The *Configuration Error* response is returned in Automatic Mode only, and indicates that Root 2 was not able to successfully parse the Device, Interface and Endpoint Descriptors of an attached device in order to configure the device. This typically indicates a descriptor formatting error in the device.

4.5 Root Fail Event

Root Fail is an asynchronous response generated by the Root 2 in response to a failure condition on the root port itself. Currently, the only Root Fail condition is an overcurrent detection on the root port. The format of the response is:

<SOP><RESP_Fail><Error><EOP>

<RESP_Fail> is the response ID, 0x94.

<Error> is an error code indicating the nature of the fault. Currently, the only defined <Error> value is 0x01, indicating a root port overcurrent. Once this condition is detected and reported, power to the root port is disabled automatically by the Root 2.

4.6 Command Error

Command Error is an asynchronous response generated by the Root 2 in response to an unrecognized or improperly-formatted command. After generating this response, Root 2 will seek for a new <SOP> sequence to being parsing the next command. The format of the Command Error response is:

<SOP><RESP_CmdError><EOP>

<RESP_CmdError> is the response ID, 0x95.

4.7 Trigger Event

The Trigger Event is an asynchronous response generated by the Root 2 in response to an external trigger on the TrigIn0 or TrigIn1 lines on the Control connector. Trigger activity will only be recognized, and this

response will only be generated, if the trigger input has been enabled using the Root_Config command. The format of the response is:

<SOP><RESP_Trigger><Source><EOP>

<RESP_Trigger> is the response ID, 0x96.

<Source> is a value 0x0 or 0x1 indicating the trigger source (TrigIn0 or TrigIn1, respectively).

5. RootScript

Command execution on the Root 2 as described thus far is referred to as "immediate mode" execution. Each command is received, executed immediately, and its response is returned to the Controller. RootScript provides a non-immediate mode of execution, or "script mode", in which a sequence of Root 2 and RootScript commands (a script) is loaded into Root 2, then executed as a program. Scripting allows commands to be executed in much more rapid succession than is possible in immediate mode. Scripts are also useful in that they can provide a more autonomous testing mechanism - one which does not rely on constant interaction with the test host.

The RootScript program language incorporates the normal set of Root 2 commands, plus additional RootScript commands which are available only in RootScript programs. Some RootScript commands are related to the definition of the script itself (e.g., program start and end commands) and some are included to provide simple program flow control such as branching and conditional execution.

A RootScript program is loaded into the Root 2 as a sequence of Root 2 and RootScript commands framed between a *Program* command and an *RS_End* command. Once loaded, program execution is initiated by a *Run* command. Each command in a RootScript is assigned an index, or line number, based on the order of the commands in the script. The first command in the script is Index #0, the second Index #1, etc.

Unlike normal Root 2 commands, which may be used either in immediate mode or in a RootScript, RootScript commands are not valid in immediate mode - they may be used only in a RootScript.

Like normal Root 2 commands, RootScript commands are framed between <SOP> and <EOP> markers, and consist of a 1-byte command ID followed, if necessary, by parameters. A RootScript program may consist of up to 512K commands, not including the Program command, but including RS_End. In addition, a RootScript is limited by the memory available in Root 2. A total of 4Mbytes is allocated for RootScript storage in Root 2. To summarize, a RootScript may not exceed 524,288 indices, including RS_END, or 4Mbytes in total size.

5.1 Program Command

Program is an immediate-mode command used to initiate loading of a new script into Root 2. Receipt of the Program command will cause any script currently existing in Root 2 memory to be erased, and will place Root 2 in "program load" mode. The format of the Program command is:

<SOP><CMD_Program><EOP>

<CMD_Program> is the command ID, 0x0C

The Root 2 will respond with :

<SOP><RESP_Program><EOP>

<RESP_Program> is the response ID, 0x8C

Once a Program command has been received and acknowledged by Root 2, all subsequent commands received from the host controller will be considered part of the newly defined script, until either an RS_End

command is encountered, or a new Program command is received. Each command, as it is received, is checked for syntax, assigned an index, and stored. Each script command is acknowledged as it is received, as follows:

<SOP><RESP_Script><Idx><Command_ID><EOP>

<RESP_Script> is a load-acknowledge byte of value 0xA0.

<Command_ID> is the command byte of the acknowledged command. For instance, if the script command being acknowledged is a DevTrans command, then <Command_ID> will be CMD_DevTrans.

<Idx> is a 16-bit value indicating the index at which the command is loaded, and will increment by 1 for each subsequent command in the script.

If a new Program command is received prior to an RS_End, the script in process of being loaded will be erased, and recording will being again at index 0.

Commands are checked during script loading. If a command error is encountered during program load, Root 2 will respond to the offending command with RESP_CmdError, rather than a RESP_Script. If the program being loaded exceeds the RootScript program limit of 1000 commands, or if the script being loaded exceeds the available program memory in Root 2 (approximately 180Kbytes), Root 2 will respond with a RESP_ScriptOvfl response code, 0x97. Once a Command Error or Script Overflow has been generated during a program load, Root 2 will respond to all commands with RESP_CmdError until either an RS_End is encountered or until a new Program command is received. Any command error encountered during a script download will result in no valid script being available for execution.

A very simple RootScript loading sequence, including the Program and RS_End commands, is shown below. Root 2 responses are indicated in color.

<SOP><CMD_Program><EOP>
<SOP><RESP_Program><EOP>
<SOP><CMD_VCC><0x64><EOP>
<SOP><RESP_Script><0x0000><CMD_VCC><EOP>
<SOP><CMD_Power><0x01><EOP>
<SOP><RESP_Script><0x0001><CMD_Power><EOP>
<SOP><RS_End><EOP>
<SOP><RESP_Script><0x0002><RS_End><EOP>

When executed, this RootScript would cause the root port Vcc to be set to 5.00V, and root port Vbus power to be switched ON.

Any RootScript command (program-mode command) received by Root 2 when not in program-load mode will elicit a Command Error response.

5.2 RS_End

The RS_End command terminates the "program load" mode, completing the load of a RootScript into Root 2 memory. The format of the RS_End command is:

<SOP><RS_End><EOP>

<RS_End> is the command ID, 0x21.

If Root 2 is in program mode, it will respond with the acknowledge:

<SOP><RESP_Load><RS_End><EOP>

By definition, it is possible to have only one RS_End command in a script.

5.3 Run

Once a script has been loaded into Root 2, it may be invoked by the *Run* command. Run is an immediate mode command; it is not part of the script itself, and can be issued any time after a script has been successfully loaded. The format of this command is:

<SOP><CMD_Run><EOP>

<CMD_Run> is the command ID, 0x0D.

Before beginning execution of the script, the Root 2 will respond to the *Run* command as follows:

<SOP><RESP_Run><EOP>

<RESP_Run> is the response ID, 0x8D.

If no script is currently loaded, the *Run* command will return a Command Error response.

Once a script has begun execution, it will execute either to completion (RS_End) or until it is terminated by another command from the test host. ANY traffic received from the Controller during script execution will terminate the script immediately.

5.4 Responses during Script Execution

Root 2 provides two modes of response handling during script execution: Full-Response mode and Quiet mode. In Full-Response mode, each Root 2 command executed as part of the script will return its normal response over the serial port to the test controller, prepended with a RESP_Script byte and the program index. In addition, encountering an RS_End command will cause the Root 2 to generate a script termination message consisting of the RESP_Script byte, the program index, the RESP_End byte and a termination index. The termination index is the index of the last instruction executed prior to the RS_End. For example, execution of the sample script in section 5.1 would result in the following messages being generated in Full Response mode:

```
<SOP><RESP_Script><0x0000><RESP_VCC><EOP>
<SOP><RESP_Script><0x0001><RESP_Power><EOP>
<SOP><RESP_Script><0x0002><RESP_End><0x0001><EOP>
```

<RESP_Script> is the script-response ID byte (0xA0), indicating that the message is a script-command response. The index words indicate the program index associated with the command that generated the response. Byte RESP_End, value 0xA1, indicates execution of the *RS_End* command, after which Root 2 is returned to immediate mode.

In Quiet mode, all responses associated with Root 2 commands executed as part of the script will be suppressed, with the exception of RS_Message and RESP_End responses. Execution of the sample script in section 3.1 running in Quiet mode would result in only a RESP_End response, with the VCC and Power

responses being suppressed. The advantage to quiet mode is that is greatly reduces the amount of traffic which the host controller must handle. In addition, although the Root 2 buffers its serial transmit data, even a simple script executing at high speed can generate a substantial amount of response traffic, most of which the host controller does not require access to, and this traffic will eventually impact the execution speed of the script. The termination index in the RESP_End response can be useful in quiet mode as an indication of which code path led to script termination.

Note that only normal Root 2 commands and the *RS_End* commands generate execution responses, even in full-response mode. RootScript commands (with the exception of *RS_End* and RS_Message) do not generate responses.

5.4.1 ResponseMode Command

The response mode during script execution is determined using the ResponseMode command. The format of this command is as follows:

<SOP><RS_Response><Mode><EOP>

<RS_Response> is the RootScript ResponseMode command ID, 0x22.

<Mode> is a 1-byte parameter = 0 for Full Response mode or =1 for Quiet mode.

As a RootScript command, ResponseMode is not available in immediate mode. The RS_Run command sets the response mode to Quiet, so it must be explicitly changed using a ResponseMode command if Full Response execution is desired. The response mode may be modified dynamically, switching between quiet mode or full-response mode as necessary to report to the test controller only that information which is desired by the script writer.

5.5 Conditions and Flow Control

RootScript provides a set of commands relating to program flow control and the evaluation of certain conditions which may be used by the script writer to control program flow. Program flow is controlled by allowing the script to transfer execution control to a new execution index, either explicitly as in the case of a Goto, or implicitly as in the case of Return. Program indexes are always represented in RootScripts as 2-byte values stored most-significant byte first (big-endien). These commands are described in the following sections.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
<UnUsed>	Timeout	Clear TrigIn1	Clear TrigIn0	Resume	<UnUsed>	Dis-Connect	Connect

Figure 5-1 *RS_Check* <Inits> Byte

5.5.1 RS_Goto Command

The RS_Goto command is the simplest mechanism available in RootScript to change program flow, allowing program control to be transferred to a new program index. The format of the command is:

<SOP><RS_Goto><Idx><EOP>

<RS_Goto> is the command ID, 0x23.

<Idx> is the 16-bit index to which control is to be transferred.

An index value which is outside the scope of the script (i.e., greater than the index of the RS_End command) will cause control to transfer to RS_End. In order to provide a simple mechanism for the script writer to indicate "goto end", index 0xFFFF is reserved as a forced end-of-script. That is, "RS_Goto 0xFFFF" will always terminate the script. Note that the index is represented MSB first, so an RS_Goto index 0x0002 would be represented as:

<SOP><RS_Goto><0x0><0x2><EOP>

5.5.2 RS_If Command

The RS_If command is used to conditionally transfer program control to a new program index if a certain condition is found to be true. The conditions upon which RS_If can operate are given in Table 3-1. The scope of RS_If is limited to the most-recently completed USB transaction. The format of the command is:

<SOP><RS_If><Cond><Idx><EOP>

<RS_If> is the command ID, 0x24.

<Idx> is the 16-bit index to which control will conditionally be transferred.

Like RS_Goto, RS_If identifies index 0xFFFF as a default "end-of-script" index. <Cond> is a one-byte parameter indicating the condition to be checked. Valid <Cond> values are those listed in Table 3-1.

5.5.3 RS_Check and RS_Cond

The RS_Check command is used to transfer program control to a new index depending selected conditions. The conditions upon which RS_Check depends are programmed individually by the RS_Cond command.

Command RS_Cond is used to set a program index which will be associated with each of the possible conditions, and to which control will be transferred upon execution of the RS_Check command if the condition is true. Once an index is set by RS_Cond for a given condition, the index will not change unless a new RS_Cond command is issued for that condition. The format of the RS_Cond command is:

<SOP><RS_Cond><Cond><Idx><State><EOP>

<RS_Cond> is the command ID, 0x25.

<Idx> is the 16-bit program index to which control will be passed by RS_Check if the condition is true. Again, a program index of 0xFFFFis used as default end-of-script.

<Cond> is the condition 1-byte ID, as shown below:

<Cond>	Condition
0	Connect
1	Disconnect
2	<Not Used>
3	Resume
4	TriggerIn0
5	TriggerIn1
6	TimerTimeout
7	BlockTrans Complete

<State> is a 1-byte value =1 to enable the condition, or =0 to disable it. If the condition is being disabled, the value of <idx> is unimportant. All conditions default to the OFF state upon execution of RS_Run.

When the RS_Check command is executed, it polls the enabled conditions until one of them is found to be true, at which time control is transferred to the associated index. Note that execution of an RS_Check command with no conditions enabled will cause RS_Check to wait indefinitely. The conditions are polled by RS_Check in the order they are listed in the above table, so an enabled Connect condition will take precedence over an enabled Timer Timeout, for instance.

The format of the RS_Check command is:

<SOP><RS_Check><Inits><EOP>

<RS_Check> is the command ID byte, 0x26. <Inits> is a one-byte value which indicates whether or not certain latched conditions should be cleared prior to execution of RS_Check. The bits of <Inits> correspond to the RS_Cond conditions listed above (See figure 5-1), however only the TriggerIn0, TriggerOut0 and SOF conditions are latched conditions, so only these conditions are affected by the <Inits> byte. If for instance, a TriggerIn0 has been latched by the Root 2 prior to the execution of RS_Check, and its associated bit in the <Inits> byte is set, the latched TriggerIn0 condition will be cleared prior to execution of RS_Check - so RS_Check will wait for the next TriggerIn0 transition before succeeding for TriggerIn0. If a TriggerIn0 condition has been latched but is *not* cleared by <Inits>, RS_Check will succeed for TriggerIn0 immediately.

In addition, each time RS_Check succeeds on a latched condition (SOF, TriggerIn0 or TriggerIn1) the condition is cleared. Two subsequent RS_Check's for SOF, for instance, would impose a delay of one SOF time between detections. Likewise, two subsequent RS_Check's for TriggerIn0 would require two transitions of the trigger line to traverse.

5.5.4 RS_Timer

The RS_Timer command is used to set a counter which is decremented by the Root 2's 1mS-tick timer. Upon reaching 0, the counter will create a timer timeout condition which may be used by RS_Cond and RS_Check. Once the counter reaches its timeout condition, it will remain at 0 until reloaded by the RS_Timer command. The format of this command is:

<SOP><RS_Timer><Count><EOP>

<RS_Timer> is the command ID byte, 0x27.

<Count> is a 32-bit value which will be loaded into the timeout counter. <Count> is represented most significant byte first (big endian).

Loading the timer with a 0 will result in a timeout condition being True. Note that, as <Count> is a 32-bit value, extremely long timeout periods, on the order of 50 days, can be generated.

5.5.5 RS_Call and RS_Return

RS_Call and RS_Return together provide call return functionality in RootScript. RS_Call pushes the index of the command immediately following it onto a program stack, then transfers program control to the specified index. The format of this command is:

<SOP><RS_Call><Idx><EOP>

<RS_Call> is the command ID, 0x29.

<Idx> is the 16-bit index to which control is to be transferred. An index value which is outside the scope of the script (i.e., greater than the index of the RS_End command) will cause control to transfer to RS_End.

Execution of an RS_Return command causes the last index pushed onto the program stack to be popped, and control to be transferred to that index. The format of the command is:

<SOP><RS_Return><EOP>

<RS_Return> is the command ID, 0x2A.

RootScript provides a maximum stack depth of 256 calls. In the event that the stack is overflowed (the number of RS_Calls exceeds the number of RS_Returns by more than 256 at any given time) or underflowed (more RS_Returns than RS_Calls are executed), the script will terminate immediately at RS_End. The termination index will point to the RS_Call command which caused the overflow, or the RS_Return command which caused the underflow.

5.6 RS_Message

The RS_Message command forces a response to the test controller, regardless of the response mode. The format of the RS_Message command is:

<SOP><RS_Message><data><EOP>

<RS_Message> is the command ID, 0x28. <data> is a data string of 0 to 63 binary bytes which will be included in the forced response. The format of the RS_Message forced response is:

<SOP><RESP_Script><Idx><RESP_Message><Timer><data><EOP>

<RESP_Script> is the normal script-response ID byte.

<Idx> is the index of the RS_Message command responsible for the message.

<RESP_Message> is an RS_Message ID byte, 0xA8.

<Timer> is a 32-bit value indicating the remaining count in the timeout counter.

<data> is the data programmed with the RS_Message command.

6. Script Management

Scripts are normally loaded into RAM on the Root 2 using the Program command, and executed directly from RAM using the Run command. Root 2 offers the option, however, to load a script into on board Flash memory, and enable that script to be executed after power up rather than entering Immediate Mode. This is referred to as a "default script". The default script is loaded into Root 2 RAM in the normal fashion, using the Program command. Once the script is successfully loaded, it can be burned into Flash using the Flash command. The Flash command can also be used to enable and disable the default script, without removing it from Flash, and to determine the status of the default script.

6.1 The Default Script

If a default script is present in Flash, and is enabled, Root 2 will, after power up initialization, load the default script into RAM and execute it. If the script terminates, or if any traffic is received on the serial port, Root 2 will abort the script and return to Immediate Mode operation. Due the amount of available Flash in Root 2, the size of the defaul script is limited to 65,536 commands or 1.3Mbytes total size.

6.2 The Flash Command

The Flash command is and immediate mode command used to manage the programming and status of the default script. It allows the script to be written to Flash, allows it to be enabled or disabled, and allows its status to be queried. The format of the Flash command is:

<SOP><CMD_Flash><Script_ID><Action>{<Name>}<EOP>

<CMD_Flash> is the Flash command ID, 0x31.

<Script_ID> is the script identifier. The only valid value is 0, indicating the default script.

<Action> indicates the action to be taken by the Flash command. Valid Actions are:

0: Disable Script
1: Enable Script
2: Burn script in RAM
3: Return status of Script

<Name> is present only if <Action> = 2. <Name> must be exactly 8 bytes in length and, while it may contain any value, it is typically used to hold an ASCII description of the default script.

The Flash-Enable, Flash-Disable and Flash-Burn commands, if successful, return the following:

<SOP><RESP_Flash><EOP>

<RESP_Flash> is the Flash command response code, 0xB1.

If there is no default script programmed, the Flash-Enable and Flash-Disable commands have no effect, and a Command Error event is returned.

In order for a Flash-Burn command to be successful, a script must have been loaded into Root 2 RAM using the Program command. In addition, the script must fit in the Flash memory area available for default script storage (approximately 160Kbytes). If either of these conditions is not met, a Command Error event is returned.

The Flash-Status command returns the following:

<SOP><RESP_Flash><Status>{<Name>}<EOP>

<RESP_Flash> is the Flash command response code, 0xB1.

<Status> is a byte indicating the status of the default script, as follows:

0: Default Script Disabled
1: Default Script Enabled
0xFF: Default Script Undefined

A status of Disabled indicates that a script is programmed into Flash, but is currently disabled, versus a status of Undefined, which indicates that no default script is programmed in Flash.

<Name> is returned only if <Status> is = 0 or 1, and contains the 8-byte Name field which was programmed with the script.