Developing PC-Based Host Applications For Root Testers

Version 1.7
June 16, 2008

1 Introduction

A Root tester's communications interface includes a comprehensive set of commands that can be used to control the unit from a PC. Commands are transmitted to the Root tester via an RS-232 port or Ethernet connection and are subject to syntax requirements as outlined in the Interface Specification document for the appropriate model (Root1 or Root2).

In order to simplify the task of creating a custom PC-based control application, RPM Systems provides a set of C-callable routines which allows the application to communicate with a Root tester. These routines satisfy the communication and syntax requirements of the Root tester interface, freeing the designer to concentrate on the control aspects of the application. The routines are provided in the form of a MS Visual C/C++-compatible dynamically-linked library called **RootComm.dll**. This library can be linked in with a custom application to form a stand-alone PC executable, providing the designer with a simple yet powerful means of creating a USB test environment.

RootComm.dll is provided as part of this distribution, as is a demonstration project, which contains examples of all of the concepts described herein and can be used as a starting template for creating your own custom Root tester control application.

You should be familiar with the document(s) entitled Root-1 Interface Specification and/or Root-2 Interface Specification prior to reading this document – paying particular attention to that Root tester's basic set of commands and operating modes. In general, there is a one-to-one correspondence between the higher-level commands provided by RootComm.dll, and the commands outlined in the interface specification; hence you can refer to it for additional details.

2 Example Projects

Visual C/C++ example projects are collected within a single workspace called Root1 Examples.dsw in the directory Examples.

Copy the entire RootComm directory to a convenient directory on you local machine. It is imperative that you maintain the directory hierarchy of this distribution; the example projects depend on relative directory paths for compilation and execution.

To view the examples, open the Root1 Examples.dsw workspace under Visual C/C++ (version 6.0 or greater). From there, select an active project within the workspace. You can then compile and run the example from the debug environment of VCC.

There are 5 different example projects. The first three: DeviceTest, PollEndpoint1, and SetAddress – all work with either Root1 or Root2 and were tested with a low-speed USB mouse. The BlockInOut and BlockLoopInOut projects are Root2-specific and concentrate on Root2's ability to move large blocks of data at high speed without intervention from the PC. They were tested using an EZ-USB FX2 development board with custom firmware.

2.1 DeviceTest

The DeviceTest project shows how to detect a device connection and subsequently launch a test. Once a device has been detected, the example prints out the current drawn by the device in both the suspended and normal operating modes. This simple example can get you on your way to developing more sophisticated tests pertinent to your instrument.

2.2 PollEndpoint1

The PollEndpoint1 project demonstrates how to poll a device endpoint from a host application. In this example, the Root tester's automatic mode is used to establish a device connection; it is then switched to manual mode to disable any further automatic polling. The example then repeatedly polls endpoint 1, prints out any data received from the poll (including sync, data pid, and 16-bit CRC of the data packet) , and maintains a count of any received NAK packets or errors. Due to the communication delays incurred by the serial port, this example works best when used with a low-speed device such as a mouse or keyboard.

2.3 SetAddress

The SetAddress project demonstrates how to sequentially address a device from addresses 1 to 127 and verify that the device addressed correctly. For each new address, this example first resets the device, sets its new address, then verifies that it can communicate with the newly addressed device.

2.4 BlockInOut

The BlockInOut project shows how to use Root2's ability to move large blocks of data at high rates of speed in a single block data command. This particular example sends 1K of data to a bulk endpoint with a packet size of 64 bytes and a service interval of 8 packets per frame (microframe if the device under test is high speed). The parameters of the test(packet size, service interval, device speed, data size, and transfer type) are all easily modified to suit your particular device.

2.5 BlockLoopInOut

The BlockLoopInOut project is similar to the BlockInOut project, except that it demonstrates how to generate a looping block transfer. The transfer parameters of the previous example (64-byte packet size, 8 packets per frame) are used, but the 1K data is looped continuously, creating traffic every frame. The loop is allowed to run for 1 second before being terminated.

3 Getting Started

3.1 Nomenclature

Functions that work for either version of Root tester (Root1 or Root2) maintain the prefix "R1_" for compatibility with earlier releases. Functions that pertain to new features that are supported only by Root-2 use the "R2_" prefix.

3.2 Required Files

The following files are included with the rootcomm distribution and are necessary for building a custom application:

Stddefs.h:	type declarations; required for calling application;
RootComm.h:	constant declarations and function prototypes; required for calling application;
RootComm.lib:	corresponding library file for RootComm.dll; required in link with calling application.

3.3 Type Declarations

RootComm function interface prototypes use the following type declarations for byte, word, and long data sizes:

```
typedef unsigned short  UWORD;
typedef unsigned char   UBYTE;
typedef unsigned long   ULONG;
```

These data types are declared in the file "stddefs.h". Any source file that uses routines in RootComm.dll should include this file.

3.4 Standard Status and Error Codes

All RootComm function calls that communicate directly with a Root tester return a status code which reflects the success or failure of the transmission of the command or reception of its response. This code can assume one of the following predefined values:

```
0:                      Command was successfully executed
```

```
R1_TIMEOUT_ERROR:       timed out waiting for a response
                        from Root-2
R1_LENGTH_ERROR:        There was an error in the length of the
                        response
R1_RESPONSE_ERROR:      There was an error in the format of the
                        response
R1_DOWNLOAD_ERROR:      There was an error during script download
                        (R1_Download only)
R1_FILE_ERROR:          There was an error opening the file
                        (R1_Download only)
```

This code is referred to in the following sections as the comm status code. For example the following code fragment initializes power to Root-2's downstream port:

```
//power the device
Comm_Status = R1_Power(POWER_ON);   //enable power
```

In addition, certain Root tester commands (R1_DevRqst, R1_DevRqstMan, R1_DevTransIn, R1_DevTransOut, and R2_BlockTransactionStatus) return another status code which reflects the success or failure of the USB transaction that was generated during processing of the command. This is referred to as the "USB transaction status" byte and can assume one of the following values:

```
STS_Success             // transaction successful
STS_ACK                 // transaction resulted in ACK
STS_NAK                 // transaction resulted in NAK
STS_STALL               //  "             "      " STALL
STS_IGNORE              //  "             "      " IGNORE
STS_DCRCErr             //  received incorrect CRC
STS_DTogErr             //  received incorrect data toggle pid
STS_SyncErr             //  received incorrect sync byte
STS_Babble              //  device babble
STS_PIDErr              //  incorrect pid format
STS_ShPktErr            //  packet too short
STS_ConfigErr           //  error in device or configuration descriptor
STS_ScheduleErr         //  error in traffic scheduling
STS_TimeoutErr          //  device did not respond to a request
                        //  within current frame
STS_NakTimeoutErr       //  device NAKed control stage for longer
                        //  than 500 msec
STS_CmdTimeoutErr       //  devi
CmdTimeoutErr           //  device did not complete control transaction
                        //  within 3 seconds
STS_BlockCommandRunningErr //tried to execute command while block
                           //command is still running
STS_NoDeviceErr         //tried to communicate with device at illegal
                          //address
```

The transaction status is almost always indirectly returned to the caller via a pointer.

3.5 Miscellaneous Predeclarations

The following declarations are also included in rootcomm.h for your convenience, and are referenced in the following sections:

```
//useful declarations for valid values for Root-1 Commands

//predeclarations for R1_RootConfig command
#define ROOT1_MODE                  0
#define AUTO_MODE                   1
#define MANUAL_MODE                 0
#define ROOT1_TRIGGER               1
#define TRIGGERS_DISABLED           0
#define TRIGGER1_ENABLED            1
#define TRIGGER2_ENABLED            2

#define ROOT1_AUTORECOVERY          2
#define AUTORECOVERY_ENABLED        1
#define AUTORECOVERY_DISABLED       2

#define ROOT1_LED_INDICATORS        3
#define ENABLE_LED_INDICATORS       1
#define DISABLE_LED_INDICATORS      0

#define ROOT1_PUSHBUTTONS           4
#define ENABLE_PUSHBUTTONS          1
#define DISABLE_PUSHBUTTONS         0

#define ROOT1_BAUD                  5
//Baud set according to following table:
#define ROOT1_BAUD_2400                   0  //-- 2400
#define ROOT1_BAUD_9600                   1 // -- 9600
#define ROOT1_BAUD_19200                  2 // -- 19200
#define ROOT1_BAUD_38400                  3 // -- 38400
#define ROOT1_BAUD_57600                  4 // -- 57600
#define ROOT1_BAUD_115200                 5 // -- 115200
#define ROOT1_BAUD_230400                 6 // -- 230400
#define ROOT1_BAUD_460800                 7 // -- 460800


#define ROOT1_CONNECT_SPEED_CONTROL       6
#define INHIBIT_HS                        1
#define ALLOW_HS                          0

//predeclarations for R1_Power
#define POWER_OFF                   0
#define POWER_ON                    1


//on device request commands to the control endpoint, usb timeouts
//can be enabled or disabled. If enabled, a device request command
//will fail with a timeout error if the device NAKs for over 500mec,
//or the entire request takes over 5 seconds to complete. If disabled,
//the device request will process to completion regardless of timing.
```

```
#define ROOT1_USB_TIMEOUT                7
#define DISABLE_TIMEOUT 1
#define ENABLE_TIMEOUT  0

//useful PID declarations for R1_DevRqst, R1_DevTransXXX
#define OUT_PID                     0x1
#define ACK_PID                     0x2
#define DATA0_PID                   0x3
#define PING_PID                    0x4
#define NYET_PID                    0x6
#define DATA2_PID                   0x7
#define IN_PID                      0x9
#define NAK_PID                     0xa
#define DATA1_PID                   0xB
#define SETUP_PID                   0xD
#define STALL_PID                   0xe
#define MDATA_PID                   0xf

// DevTrans Control Byte Defines
//bit 0 is reserved!
#define DT_LOW_SPEED    0x0000 //perform transaction at low speed
#define DT_FULL_SPEED   0x0002 //perform transaction at full speed
#define DT_HIGH_SPEED   0x0008 //perform transaction at high speed
#define DT_ISOCH        0x0004 //perform isochronous transaction
#define DT_BULK         0x0008 //perform bulk transaction
#define DT_INTERRUPT    0x000c //perform interrupt transaction
#define DT_USE_ALT_BUFFER 0x0010 //use alternate buffer
#define DT_IMMED        0x0080 //issue transaction immediately
#define DT_SPLIT        0x0020 //force split transaction

//these can  only be used with block dev trans command
#define DT_LOOP         0x0100 //loop this transaction indefinitely
#define DT_STOP_ON_NAK  0x0200 //stop transaction when device NAKs
//command only)
#define DT_USE_NEXT_DATAPID 0x400 //use next datapid in sequence

//used in conjunction with response from R1_RootStatus command
#define ROOTSTATUS_CONNECT_MASK                 0x43
#define ROOTSTATUS_POWER_MASK                   4
#define ROOTSTATUS_SUSPEND_MASK                 8
#define ROOTSTATUS_ENABLED_MASK                 0x10
#define ROOTSTATUS_AUTORECOVERY_MASK            0x20
#define ROOTSTATUS_NOT_CONNECTED                0
#define ROOTSTATUS_FULL_SPEED_CONNECT           2
#define ROOTSTATUS_LOW_SPEED_CONNECT            1
#define ROOTSTATUS_HIGH_SPEED_CONNECT           0x40
#define ROOTSTATUS_CONNECTED                    0x43



//DevRqst predefines
//DevRqst predefines
#define DR_PACKETSIZE_8       0
#define DR_PACKETSIZE_16      1
#define DR_PACKETSIZE_32      2
#define DR_PACKETSIZE_64      3
```

```
#define DR_HIGH_SPEED         2
#define DR_FULL_SPEED         1
#define DR_LOW_SPEED          0
```

4 RootComm Functions

4.1 R1_OpenComm --- Open Communications (RS-232) Port

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


void * R1_OpenComm(char * port,UWORD PacketTimeout );
```

Description

R1_OpenComm() opens a COMM port at 19200 baud for communication with Root1.

port:	pointer to string indicating comm port (i.e., "comm1:")
Baud:	baud rate, in decimal
PacketTimeout:	communication timeout, in seconds

Most commands to Root-1 respond within a few milliseconds. However, some transactions are entirely dependent on the speed of the device under test and its ability to respond to commands. PacketTimeout sets the maximum time to wait for a command response; in most applications 10 seconds is adequate. It may be set to a much larger number for development purposes.

Returns

0:	error occurred opening Comm port
Comm Handle:	port opened successfully.

The handle of the comm port is returned for convenience only.

Example:

```
{
      ..
      if (!(R1_OpenComm ( "com1:",10 )))
      {
            printf("error opening comm port\r\n");
      }
}
```

4.2 R2_OpenComm --- Open Communications (RS-232) Port

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


void * R2_OpenComm(char * port,int Baud, UWORD PacketTimeout );
```

Description

R2_OpenComm() opens a COMM port for communication with the Root2. Root2 supports different baud rates; however, upon power up Root2 defaults to 115200 baud. In order to switch baud rate, communication must first be established at this baud rate, then subsequently changed (see the command R1_RootConfig()), and the COMM port closed and re-opened at the new rate.

port:	pointer to string indicating comm port (i.e., "comm1:")
Baud:	baud rate, in decimal
PacketTimeout:	communication timeout, in seconds

Most commands to Root-2 respond within a few milliseconds. However, some transactions are entirely dependent on the speed of the device under test and its ability to respond to commands. PacketTimeout sets the maximum time to wait for a command response; in most applications 10 seconds is adequate. It may be set to a much larger number for development purposes.

Returns

0:	error occurred opening Comm port
Comm Handle:	port opened successfully.

The handle of the comm port is returned for convenience only.

Example:

```
{
      ..
      if (!(R2_OpenComm ( "com1:",115200, 10 )))
      {
            printf("error opening comm port\r\n");
      }
}
```

4.3 R1_CloseComm --- Close Communications (RS-232) Port

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


void  R1_CloseComm(void );
```

Description

R1_CloseComm() closes the COMM port previously opened using R1_OpenComm().

Returns

Nothing.

Example:

```
{
      ..
      R1_CloseComm ();
      printf("comm port closed \r\n");
}
```

4.4 R2_NetConnect --- Connect to Root2 via Ethernet

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


int R2_NetConnect(char * netaddr,UWORD PacketTimeout );
```

Description

R2_NetConnect() opens an ethernet connection for communication with Root1.

Netaddr:	pointer to string indicating IP address or valid host name
PacketTimeout:	communication timeout, in seconds

The netaddr string can be either a valid IP string, such as "192.168.0.1", or a valid host name, such as "root2_lab1". In either case, Root2 must be preconfigured with an IP address compatible with your network.

Most commands to Root-2 respond within a few milliseconds. However, some transactions are entirely dependent on the speed of the device under test and its ability to respond to commands. PacketTimeout sets the maximum time to wait for a command response; in most applications 10 seconds is adequate. It may be set to a much larger number for development purposes.

Returns

-1:	could not connect to device
0:	connection successful.

Example:

```
{
      ..
      if (R2_NetConnect( "192.168.0.5",10 ))
      {
            printf("error connecting to device\r\n");
      }
}
```

4.5 R2-NetDisconnect --- Disconnect from Ethernet

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


void  R2_NetDisconnect(void );
```

Description

R2_NetDisconnect() closes a previously open net connection.

Returns

Example:

```
{
      ..
      R2_Netdisconnect();
}
```

4.6 R1_USB_Reset --- Resets the USB device

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_USB_Reset(void);
```

Description

R1_USB_Reset() issues a USB reset to the Root-1's downstream port. The function returns after the reset signaling is complete.

If the Root-1 is in Automatic Mode prior to the call to R1_USB_Reset(), and a device is attached, the Root-1 will attempt to enumerate and configure the device immediately after the reset signaling and prior to returning from the function call.

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      UBYTE CurrentStatus;
      //reset the device
      Comm_Status = R1_USB_Reset();
      if (!Comm_Status)
      {
            Comm_Status =  R1_RootStatus(&CurrentStatus);
            if (!Comm_Status &&
                  (CurrentStatus & ROOTSTATUS_CONNECT_MASK))
            {
                  //device connected after a reset, continue…
                  //..
                  //..
            }
      }
      else
      {
            printf("Reset Command failed!!\r\n");
      }
}
```

4.7 R1_Suspend --- Suspend Device

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_Suspend(void);
```

Description

R1_Suspend() suspends all bus activity to the Root-1's downstream port, including full-speed and low-speed Start of Frame (SOF) signaling.

Returns

Comm Status Word

Example:

```
{
      UBYTE PortStatus;
      UWORD Comm_Status;
      //suspend the device
      Comm_Status = R1_Suspend();
      if (!Comm_Status)
      {
            //command transmission successful, Get root portstatus
            Comm_Status = R1_RootStatus(&PortStatus);
            if (!Comm_Status && (PortStatus & ROOTSTATUS_SUSPEND_MASK))
            {
                  //device successfully suspended, continue
                  //..
                  //..
            }
      }
      printf("R1_Suspend Command failed!!\r\n");
}
```

4.8 R1_Resume --- Resume Device

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_Resume(void);
```

Description

R1_Resume() resumes all bus activity to the Root-1's downstream port, SOF signaling. If the device is in Automatic mode, all Automatic mode functionality (downstream port polling, interrupt endpoint polling) is restored.

Returns

Comm Status Word

Example:

```
{
      UBYTE PortStatus;
      UWORD Comm_Status;
      //resume the device
      Comm_Status = R1_Resume();
      if (!Comm_Status)
      {
            //device Resume successful, Get root portstatus
            Comm_Status = R1_RootStatus(&PortStatus);
            if (!Comm_Status &&
                  (!(PortStatus & ROOTSTATUS_SUSPEND_MASK)))
            {
                  //device successfully resumed, continue
                  //..
                  //..
            }
      }
      printf("R1_Resume Command failed!!\r\n");
}
```

4.9 R1_Power --- Control Power to USB Port

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_Power(UBYTE On);
```

Description

R1_Power() controls the power applied to the downstream port according to the passed Parameter. If "On" is non-zero, power is applied to the port, otherwise it is turned off.

Returns

Comm Status Word

Example:

```
{
      UBYTE PortStatus;
      UWORD Comm_Status;
      //power the device
      Comm_Status = R1_Power(POWER_ON);
      if (!Comm_Status)
      {
            //device power applied successful, Get root portstatus
            Comm_Status = R1_RootStatus(&PortStatus);
            if (!Comm_Status &&
                  (!(PortStatus & ROOTSTATUS_POWER_MASK)))
            {
                  //device successfully powered, continue
                  //..
                  //..
            }
      }
      printf("R1_Power Command failed!!\r\n");
}
```

4.10 R1_VCC --- Set VCC

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_VCC(UBYTE Value);
```

Description

R1_VCC() controls the level of VCC applied to the downstream port. The relationship between the parameter "Value" and the level of VCC is given in volts by:

VCC = 4 + Value/100

Thus, a value of 100 would give a VCC of 5 volts.

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      //power the device
      Comm_Status = R1_VCC(100);
      If (!Comm_Status)
      {
            //device VCC successful, Turn on power
            Comm_Status = R1_Power(POWER_ON);
            //..
      }
      printf("R1_VCC Command failed!!\r\n");
}
```

4.11 R1_VccMeasI --- Single Precision Current Measurement

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_VccMeasI(UBYTE *Current);
```

Description

R1_VccMeasI() returns the single-precision (8-bit) value of the current drawn at the downstream port. The relationship between the byte value returned and the current drawn at the port is given in milliamps by:

ICC = Current * 3mA

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      UBYTE current;
      //power the device
      Comm_Status = R1_VCC(100);
      Comm_Status = R1_Power(POWER_ON);
      Comm_Status = R1_VccMeasI(&current);
      if (!Comm_Status)
      {
            //current received.. continue
            //..
      }
}
```

4.12 R1_VccMeasI_DP --- Double Precision Current Measurement

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_VccMeasI(ULONG *Current);
```

Description

R1_VccMeasI_DP() returns the Double-precision (32-bit) value of the current drawn at the downstream port.

For Root2, the relationship between the byte value returned and the current drawn at the port is given in microamps by:

ICC (A) = Current * 2.959e-6

For Root1, the relationship between the byte value returned and the current drawn at the port is given in microamps by:

ICC (A) = Current * 250e-6

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      ULONG current;
      //power the device
      Comm_Status = R1_VCC(100);
      Comm_Status = R1_Power(POWER_ON);
      Comm_Status = R1_VccMeasI_DP(&current);
      if (!Comm_Status)
      {
            //current received.. continue
            //..
      }
}
```

4.13 R1_RootStatus --- Get Root1 Port Status

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_RootStatus(UBYTE *Status);
```

Description

R1_RootStatus() returns the single-byte value of Root-1's port status. The following predeclarations are available for interpreting the Status byte:

```
#define ROOTSTATUS_CONNECT_MASK                0x43
#define ROOTSTATUS_POWER_MASK                  4
#define ROOTSTATUS_SUSPEND_MASK                8
#define ROOTSTATUS_ENABLED_MASK                0x10
#define ROOTSTATUS_AUTORECOVERY_MASK           0x20

#define ROOTSTATUS_NOT_CONNECTED               0
#define ROOTSTATUS_FULL_SPEED_CONNECT          2
#define ROOTSTATUS_LOW_SPEED_CONNECT           1
#define ROOTSTATUS_HIGH_SPEED_CONNECT          0x40
#define ROOTSTATUS_CONNECTED                   0x43
```

Root-2's Autorecovery feature is on if the bit in the corresponding mask position of the status byte is "1". For details on the Autorecovery feature, see the description of `R1_RootConfig()`.

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      UBYTE PortStatus;
      //power the device
      Comm_Status = R1_VCC(100);
      Comm_Status = R1_Power(POWER_ON);
      Comm_Status = R1_RootStatus(&PortStatus);
      if (!Comm_Status)
      {
            if ((PortStatus & CONNECT_MASK) ==
            ROOTSTATUS_FULL_SPEED_CONNECT)
```

```
            {
                  //full speed device connected.. continue
                  //..
            }
            if ((PortStatus & CONNECT_MASK) ==
            ROOTSTATUS_LOW_SPEED_CONNECT)
            {
                  //LOW speed device connected.. continue
                  //..
            }
            if (PortStatus & ROOTSTATUS_ENABLED_MASK)
            {
                  // device enabled.. continue
                  //..
            }
            if (PortStatus & ROOTSTATUS_POWER_MASK)
            {
                  // device powered.. continue
                  //..
            }
            if (PortStatus & ROOTSTATUS_SUSPEND_MASK)
            {
                  // device suspended.. continue
                  //..
            }
      }
}
```

4.14 R1_RootConfig --- Configure Root-1

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_RootConfig(UBYTE Parameter,UBYTE Value);
```

Description

R1_RootConfig() allows the caller to set the Root-1's configurable parameters. The valid parameters and their allowable values are:

Parameter	Value(s)
ROOT1_MODE	MANUAL_MODE, AUTO_MODE
ROOT1_TRIGGER	TRIGGERS_DISABLED, TRIGGER1_ENABLED, TRIGGER2_ENABLED, (TRIGGER1_ENABLED | TRIGGER2_ENABLED)
ROOT1_AUTORECOVERY(2)	AUTORECOVERY_DISABLED, AUTORECOVERY_ENABLED
ROOT1_LED_INDICATORS	ENABLE_LED_INDICATORS, DISABLE_LED_INDICATORS
ROOT1_PUSHBUTTONS	ENABLE_PUSHBUTTONS,DISABLE_PUSHBUTTONS
ROOT1_BAUD*	ROOT1_BAUD_2400,ROOT1_BAUD_9600, ROOT1_BAUD_19200, ROOT1_BAUD_38400, ROOT1_BAUD_57600, ROOT1_BAUD_115200, ROOT1_BAUD_230400 ROOT1_BAUD_460800
ROOT1_CONNECT_SPEED_CONTROL*	INHIBIT_HS,ALLOW_HS
ROOT1_USB_TIMEOUT	DISABLE_TIMEOUT, ENABLE_TIMEOUT

*= Root2 only. This parameter not supported by Root1.

Autorecovery refers to a feature which when enabled, forces Root-2 to attempt to re-enable power, reset and reconfigure a connected device in the event of overcurrent detection on either the root port or on a device downstream of a hub.

Led indicators and pushbuttons options allow a Monitor-1 device to be inserted in-line with the RS-232 comm cable; it has buttons to power-cycle and/or reset the device, and uses different led patterns to indicate device status.

Connect speed can be set so that high speed connections are inhibited or allowed.

Timeouts of 5 seconds maximum for a Device Request to complete, and 500 msec maximum between ACKs on any phase of the transaction (as specified by section 9.2.6.4 of the USB 1.1 Specification), can be enforced by setting ROOT1_USB_TIMEOUT to ENABLE_TIMEOUT. If your device exceeds these limits, the Device Request will abort with a timeout error. Disable this timeout to allow your device to take longer to complete the request.

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      //configure root1 for manual mode
      Comm_Status = R1_RootConfig(ROOT1_MODE,MANUAL_MODE);
      //configure root1 for auto mode
      Comm_Status = R1_RootConfig(ROOT1_MODE,AUTO_MODE);
      //disable both triggers
      Comm_Status = R1_RootConfig(ROOT1_TRIGGER,
                  TRIGGERS_DISABLED);
      //enable both triggers
      Comm_Status = R1_RootConfig(ROOT1_TRIGGER,
                  TRIGGER1_ENABLED | TRIGGER2_ENABLED);
      //enable autorecovery
      Comm_Status = R1_RootConfig(ROOT1_AUTORECOVERY,
                  AUTORECOVERY_ENABLED);
      //set baud
      //careful here! After this command is issued,
      //you'll have to switch baud rates to be able to continue
      //talking!
      Comm_Status = R1_RootConfig(ROOT1_BAUD,
                  ROOT1_BAUD_19200);
      //force high speed devices to connect at full speed
      Comm_Status = R1_RootConfig(ROOT1_CONNECT_SPEED_CONTROL,
                  INHIBIT_HS);
}
```

4.15 R1_DataPort --- Strobe Data Port

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_DataPort(UBYTE data);
```

Description

R1_DataPort() writes the Root-2 data port with the contents of "data".

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      //toggle the dataport output
      Comm_Status = R1_DataPort(0xff);
      Comm_Status = R1_DataPort(0x00);
}
```

4.16 R1_MaskDataPort --- Mask Data Port

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_MaskDataPort(UBYTE AndValue,UBYTE OrValue);
```

Description

R1_MaskDataPort() logically "and"s the Root-1 data port with the contents of "Andvalue", then logically "or"s the result with "OrValue", then writes the result back out to the Data Port.

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      //write the dataport output
      Comm_Status = R1_DataPort(0xf0);
      //write bit1 and bit0 of dataport to 1, leaving
      //other bits undisturbed
      Comm_Status = R1_MaskDataPort(0xfc,0x03);
}
```

4.17 R1_DevRqst --- Issue Device Request

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_DevRqst( UBYTE Address, UBYTE *DataBuff, UWORD DataLen,
                  UBYTE *RespBuff,UWORD *RespLen);
```

Description

R1_DevRqst() issues a Device Request command. The arguments are as follows:

Address:	address of the device
DataBuff:	pointer to data to send to the device
DataLen:	size of data pointed to by "data", in bytes
RespBuff:	pointer to buffer to receive device response. Response consists of a status byte in RespBuff[0] followed by optional Data bytes in RespBuff[1..n]
RespLen:	pointer to UWORD containing maximum allowable length of receive Data

Entire USB SETUP transactions – the SETUP through STATUS phases -- can be issued to a downstream device using a single R1_DevRqst command. R1_DevRqst commands can be issued to devices configured by the Root-1 in automatic mode. You MUST match the address of the targeted device to the address assigned by Root-1 when it configured the device. The Root-1 will use this address to match the speed and packet size of the device in question when it issues the R1_DevRqst command (see R1_DevRqstMan() to issue device requests with explicit packet size and speed settings).

The data in DataBuff should minimally contain an 8-byte SETUP command – and it follows that DataLen should not be less than 8. In the case of a host-to-device directional transfer (OUT transactions following the SETUP phase), the additional data to be sent in OUT transactions should be concatenated to the SETUP command in DataBuff and the additional length of this data accounted for in DataLen.

Returns

R1_DevRqst returns a Comm Status Word. If the device also returns data in response to the transaction, it will be concatenated to a 1-byte transaction status word (generated by the Root-1) and returned in the receive buffer pointed to by RcvBuff. RespLen will contain the length of the data in bytes that was returned, including the prepended status byte.

Example:

```
{
      #define STANDARD_CONFIG_DESCRIPTOR_LEN 0x12
      UWORD Comm_Status;
      //here's a SETUP command to get the configuration descriptor.
      //Note that if the direction of the SETUP was HOST to DEVICE, the
      //additional data to be sent after the SETUP would be defined
      //after the initial 8 bytes in the array below.
      UBYTE get_config[8] = {0x80,0x06,0x00,0x01,0x00,0x00,0x12,0x00};

      //allocate a buffer to receive it
      UBYTE Buffer[256];

      //and a buffer to receive the length of the descriptor
      UWORD length;

      //set length to maximum we are willing to receive
      length = sizeof(Buffer);

      //call R1_DevRqst to get the config descriptor.
      //This example assumes that prior to this code fragment, a device
      //has been plugged into the Root-1 and was configured in
      //Automatic mode.  Therefore its address will be 2 (Root1 always
      //assigns the root downstream device an address of 2)
      //
      Comm_Status = R1_DevRqst(2,get_config,
                        sizeof(get_config),Buffer,&length);

      //if this request is successful, the first byte of the data
      //returned should be the status of the transaction and the
      //remaining bytes  will contain the
      //configuration descriptor itself.  Check that this is the case.
            if ( (!Comm_Status) && (length ==
            STANDARD_CONFIG_DESCRIPTOR_LEN+1) &&
                  (Buffer[0] == STS_Success))
      {
            //all checks successful!
            //config descriptor begins at Buffer[1]
            //process it here
            //..
            //..
            //..

      }
}
```

4.18 R1_DevRqstMan --- Issue Device Request Manually

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_DevRqst( UBYTE Address, UBYTE Speed,
                  UBYTE Packetsize, UBYTE *DataBuff, UWORD DataLen,
                  UBYTE *RespBuff,UWORD *RespLen);
```

Description

R1_DevRqstMan() issues a Device Request command. The arguments are as follows:

Address:	address of the device
Speed:	speed of transaction: 2 = high speed, 1 = full speed, 0 = low speed
Packetsize:	max packet size to use in request: 8,16,32, or 64 bytes (encoded)
DataBuff:	pointer to data to send to the device
DataLen:	size of data pointed to by "data", in bytes
RespBuff:	pointer to buffer to receive device response. Response consists of a status byte in RespBuff[0] followed by optional Data bytes in RespBuff[1..n]
RespLen:	pointer to UWORD containing maximum allowable length of receive Data

R1_DevRqstMan is identical to R1_DevRqst, except that it allows the speed and packet size of the request to be explicitly set prior to the transaction. You can use this command to "manually" issue device requests to devices that have not been automatically configured by Root-2. Typically this command is used when Root-2 is running in manual mode, to communicate with devices when more control or customization over the setup process is desired.

Returns

R1_DevRqstMan returns a Comm Status Word. If the device also returns data in response to the transaction, it will be concatenated to a 1-byte transaction status word (generated by the Root-1) and returned in the receive buffer pointed to by RcvBuff. RespLen will the length of the data in bytes that was returned, including the prepended status byte.

Example:

```
{

      UWORD Comm_Status;
```

```
	//here is an example of how to set the address of a device in
	//manual mode, using R1_DevRqstMan.

	//define a command to set the address to 0x55
	UBYTE set_address[8] = {0,5,0x55,0,0,0,0,0};
	//allocate a buffer for received data
	UBYTE Buffer[256];
	//and a buffer to receive the length of received data
	UWORD length;
	//set length to maximum we are willing to receive
	length = sizeof(Buffer);

	Comm_Status = R1_RootConfig(ROOT1_MODE,MANUAL_MODE);
	Comm_Status = R1_VCC (100);
	Comm_Status = R1_Power (POWER_ON);

	//assume the device is  plugged in!
	Comm_Status = R1_USB_Reset();

	//device should respond to address 0 at this point.
	//call R1_DevRqstMan to set the address
	Comm_Status = R1_DevRqstMan(0, DR_FULL_SPEED,
	                  PACKETSIZE_8, set_address,
	                  sizeof(set_address),Buffer,&length);

	//if this request is successful, the first byte of the data
	//returned is  the status of the transaction.
	if ( (!Comm_Status && length == 1 &&
	            Buffer[0] == STS_Success)
	{
	      //all checks successful!
	      //
	      //..
	      //..
	      //
	      do something with the data
	      print_data(&buffer[1],length-1);

	}
}
```

4.19 R1_DevTransOut – issue a DevTrans Host-to-Device Request

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_DevTransOut( UBYTE Address, UBYTE Endpoint, UBYTE pid,
                    UBYTE Control, UBYTE DataPID, UBYTE* DataBuff,
                   UWORD DataLen,UBYTE *RespStatus);
```

Description

R1_DevTransOut() issues a device "out" request – that is, a single USB transaction with a host-to-device directional data transfer. This command differs from R1_DevRqst in that only a single phase of a transaction is generated. Multiple R1_DevTransOut and R1_DevTransIn commands can accomplish the same thing as a single R1_DevRqst command, but allows greater visibility and control over the individual phases of the transaction.

Address:	address of the device
Endpoint:	endpoint of the transfer
Pid:	PID
Control:	Control Byte for the transfer*
DataPID:	Data PID
DataBuff:	pointer to data to send to the device
DataLen:	size of data pointed to by "DataBuff", in bytes
RespStatus:	pointer to UBYTE for returning status of the transaction.

*The Control field is used to specify other transfer mechanisms using the logical-or of the following flags:

```
DT_LOW_SPEED            //perform transaction at low speed
DT_FULL_SPEED           //perform transaction at full speed
DT_HIGH_SPEED           //perform transaction at high speed
DT_ISOCH                //perform isochronous transaction
DT_BULK                 //perform bulk transaction
DT_INTERRUPT            //perform interrupt transaction
DT_USE_ALT_BUFFER       //use alternate buffer
DT_IMMED                //issue transaction immediately
                        //without waiting for SOF
DT_SPLIT                //force split transaction
```

The "Alternate Buffer" refers to an internal buffer which can be used for looping back data. If `DT_USE_ALT_BUFFER` is set, the data for the OUT transaction will be sourced from this internal buffer. This is most commonly used after directing an IN transaction

TO the alternate buffer, so that contains available data for the subsequent OUT. See the example code for the function R1_DevTransIn().

Forcing a transaction to split is only valid when directing transactions to a low or full speed device through a high speed hub.

Returns

R1_DevTransOut returns a Comm Status Word. It also returns a status byte indicating the result of the USB transaction in the byte pointed to by RespStatus.

Example:

```
{
      UWORD Comm_Status;
      //this example uses R1_DevTransOut to generate the SETUP phase
      //of a set address command

      UBYTE set_address[8] = {0,5,2,0,0,0,0,0};
      UBYTE retval;

      //reset the device
      Comm_Status = R1_USB_Reset();

      //Use R1_DevTransOut to generate the SETUP phase of a SET_ADDR
      //command.
      Comm_Status = R1_DevTransOut(
            0,                //address = 0
            0,                //endpoint = 0
            SETUP_PID,        //SETUP pid
            DT_FULLSPED,      //full-speed transaction
            DATA0_PID,        //data0 pid for data portion of
                              //transaction
            set_address,      //pointer to SET_ADDRESS command
            sizeof(set_address),//size of command
            &retval           //pointer to storage for return status
            );

            //see if we got back an ACK in response to setup phase
            if (retval !=  STS_ACK)
            {
                  printf ("\ack not received");
            }

}
```

4.20 R1_DevTransIn – issue a DevTrans Device-to-Host Request

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_DevTransIn(UBYTE Address, UBYTE Endpoint, UBYTE pid,
             UBYTE Control,UBYTE *RespBuff,UWORD *RespLen)
```

Description

R1_DevTransIn() issues a device "in" request – that is, a single USB transaction with a device-to-host directional data transfer. This command differs from R1_DevRqst in that only a single phase of a transaction is generated. Multiple R1_DevTransOut and R1_DevTransIn commands can accomplish the same thing as a single R1_DevRqst command, but allows greater visibility and control over the individual phases of the transaction.

Address:	address of the device
Endpoint:	endpoint of the transfer
Pid:	PID
Control:	Control Byte for the transfer*
Respbuff:	pointer to buffer in which to place returned data
RespLen:	size of returned data

*The Control field is used to specify other transfer mechanisms using the logical-or of the following flags:

```
DT_LOW_SPEED            //perform transaction at low speed
DT_FULL_SPEED           //perform transaction at full speed
DT_HIGH_SPEED           //perform transaction at high speed
DT_ISOCH                //perform isochronous transaction
DT_BULK                 //perform bulk transaction
DT_INTERRUPT            //perform interrupt transaction
DT_USE_ALT_BUFFER       //use alternate buffer
DT_IMMED                //issue transaction immediately
                        //without waiting for SOF
DT_SPLIT                //force split transaction
```

The "Alternate Buffer" refers to an internal buffer which can be used for looping back data. If `DT_USE_ALT_BUFFER` is set, the data from the IN transaction will directed to this internal buffer – where it can be subsequently sent back to the device under test with the function R1_DevTransOut(). The communication delay associated with moving the data to and from the host is eliminated using this method. See the example below.

Returns

R1_DevTransIn returns a Comm Status Word. It also returns the status of the transaction in the first byte of the returned data, and any data received from the device during the transaction is appended to this byte.

Example:

```
{
      //this example uses R1_DevTransIn to generate the IN phase
      //of a set address command (see R1_DevTransOut example).
      UWORD Comm_Status;
      UBYTE Resp[16];               //intermediate buffer
      UWORD in_length;              //length of data returned
      UBYTE retval;

      do
      {
            in_length = 13;         //we're expecting at most 13 bytes:
                                    //status, sync,pid,data bytes (0 to
                                    //8), and 2 crc bytes

            Comm_Status = R1_DevTransIn(
            2,                      //device address = 2
            0,                      //endpoint 0
            IN_PID,                 //IN pid
            DT_FULL_SPEED,          //full speed transaction
            Resp,                   //point to buffer for return data
            &in_length              //point at length we are prepared
                                    //to receive
            );

      //keep doing this until response is something other than a NAK
      }while ((in_length == 1) && (Resp[0] == STS_NAK));

      //assuming the transaction completed, we expect the following:
      //Resp[0] = STS_Success (transaction ok)
      //in_length = 5 (status, sync, pid, 0 data bytes,  2 crc bytes)

      if  ((Resp[0] != STS_Success) || (in_length != 5))
      {
            //something not right..
      }
      else
      {
            //buffer contains data. Do something with it….
            Print_Data(&Resp[1],length-1);
      }
```

```
        //now try it again. Use the alternate buffer to loop back
        //the data
        in_length = 13;         //we're expecting at most 13 bytes:
                                //status, sync,pid,data bytes (0 to
                                //8), and 2 crc bytes

        Comm_Status = R1_DevTransIn(
              2,                        //device address = 2
              0,                        //endpoint 0
              IN_PID,                   //IN pid
              (DT_FULL_SPEED |
              DT_USE_ALT_BUFFER),       //full speed transaction - data
                                        //goes to internal buffer
              Resp,                     //point to buffer for return data
              &in_length);              //point at length we are prepared
        //now send the data in the internal buffer
        //back to the device at the same endpoint
        Comm_Status = R1_DevTransOut(
              2,                        //device address = 2
              0,                        //endpoint 0
              OUT_PID,                  //OUT pid
              (DT_FULL_SPEED |
              DT_USE_ALT_BUFFER),       //full speed transaction - data
                                        //goes to internal buffer
              DATA0_PID,                //use data pid
              (UBYTE *) 0,              //no data
              0,                        //no data size - the internal
                                        //buffer's size will be used
              &retval);         //pointer to storage for return status

}
```

4.21 R1_Download--- Download Script File

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_Download(char *filepath);
```

Description

R1_Download () downloads the Root-1 Script file located via "filepath". For a description of script files and how to create and use them, see the document entitled "Developing And Using Root-1 Scripts".

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      //set up the ROOT1 in manual mode
      printf("********Rootscript Download test************\r\n");
      Comm_Status = R1_RootConfig (ROOT1_MODE,MANUAL_MODE);
      printf("\r\ndownloading file :rs_msg.rs....\r\n");
      Comm_Status = R1_Download("rs_msg.rs");
}
```

4.22 R1_Run--- Run Script File

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_Run(void);
```

Description

R1_Run() executes a Root-1 Script.

Returns

Comm Status Word

Example:

```
{
      UWORD Comm_Status;
      printf("\r\nexecuting rootscript..........\r\n");
      Comm_Status = R1_Run();
}
```

4.23 R1_SetupCallback--- Set up Call Back Function

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


void R1_SetupCallback ( PFVP fptr );
```

Description

Root-1 generates asynchronous messages in response to events that occur outside of normal host command processing. For instance, if Root-1 is in automatic mode and a device is plugged in, an asynchronous connect message is generated. When Root-1 polls devices in automatic mode, any data received by the devices will be returned – again asynchronously. To allow applications to field these messages outside of normal program flow, R1_SetupCallBack() can be called with a pointer to a function which the DLL will call any time an asynchronous message is received. This function should be of the syntax:

Void handle_msg(UBYTE *message,int length)

Where "message" is a pointer to the asynchronous message and length is its total length.

See the firmware specification for details on the format of asynchronous responses.

Returns

Example:

```
//define a function which can parse all possible asynchronous
//messages generated by Root-1.
//
void STDCALL MessageHandler(UBYTE *message, int length)
{
      int i;
      switch (*message)
      {
      case RSP_Connect:
            if (!*(message+1))
                  printf("connect: addr:%02x
                  class:%02x vendor:%02x%02x product:%02x%02x\r\n",
                  *(message+2),*(message+3),*(message+5),
                  *(message+4),*(message+7),*(message+6));
```

```
      else
            printf("disconnect: addr:%02x\r\n",*(message+2));
      break;

case RSP_Data:
      printf("data: addr:%02x endpt:%02x value: ",
      *(message+1),*(message+2));
      message+=3;
      length -=3;
      if (length > 0)
      {
            for (i = 0; i < length-1; i++)
                  printf("%02x:",*message++);
            printf("%02x\r\n",*message);
      }
      break;
case RSP_Status:
      printf("status: hub addr:%02x port:%02x
            value: %02x ",*(message+1),*(message+2)
            ,*(message+3));
      break;
case RSP_Error:
      printf("error:addr:%02x endpt:%02x type:",
      *(message+1),*(message+2));
      switch (*(message+3))
      {
      case STS_IGNORE:
            printf("Ignore Error\r\n");
            break;
      case STS_DTogErr:
            printf("Toggle Error\r\n");
            break;
      case STS_SyncErr:
            printf("Sync Error\r\n");
            break;
      case STS_Babble:
            printf("Babble Error\r\n");
            break;
      case STS_PIDErr:
            printf("PID Error\r\n");
            break;
      case STS_ShPktErr:
            printf("Short Packet Error\r\n");
            break;
      case STS_ConfigErr:
            printf("Config Error\r\n");
            break;
      }
      break;
case RSP_Fail:
      printf("fail: value:%02x\r\n",*(message+1));
      break;
case RSP_CmdError:
      printf("Command Error\r\n");
      break;
case RSP_Trigger:
      printf("trigger: value:%02x\r\n",*(message+1));
```

```
            break;
      case RSP_ScriptOverflow:
            printf("Script Overflow\r\n");
            break;
      case RSP_Script:
            printf("Script Response: index:%02x%02x value: %02x\r\n",
                  *(message+1),*(message+2),*(message+3));
            if (*(message+3) == RSP_Message)
            {
                  //this is a script response message.
                  //print the message itself
                  printf("\ttimer:%02x%02x%02x%02x data:",
                        *(message+4),*(message+5),
                  *(message+6),*(message+7));
                  message += 8;
                  length -= 8;
                  if (length)
                  {
                        for (i = 0; i < length-1; i++)
                              printf("%02x:",*message++);
                        printf("%02x\r\n",*message);
                  }
            }
            break;
      default:
            printf("Unrecognized response: value:");
            if (length > 0)
            {
                  for (i = 0; i < length-1; i++)
                        printf("%02x:",*message++);
                  printf("%02x\r\n",*message);
            }
            break;
      }
}



void main(void)
{
      //in main section of code, install MessageHandler
      UWORD Comm_Status;
      printf("\r\installing message handler..........\r\n");
      R1_SetupCallback ( MessageHandler);
      //set up for automatic mode
      Comm_Status = R1_RootConfig(ROOT1_MODE,AUTO_MODE);
      Comm_Status = R1_VCC (100);
      Comm_Status = R1_Power (POWER_ON);
      //at this point, Root1 should be running in auto mode.  Any
      //asynchronous messages will cause the callback to be executed
      //data from root-1.
      while (1)
      {
            //device plugged in will cause a call to MessageHandler()
            //which will print the received asynchronous message
            //..
            //..
```

```
                //..
        }
}
```

4.24 R1_Get_DLL_Version--- Get Version

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R1_Get_DLL_Version(void);
```

Description

R1_Get_DLL_Version() returns the version of the DLL.

Returns

16-bit version number in hexadecimal form (i.e., 0x0100 = version 1.00).

Example:

```
{
      UWORD Version = R1_Get_DLL_Version();
      if (Version < 0x100)
      {
            printf("DLL out of date\r\n");
      }
      //..
      //..
      //..
}
```

4.25 R2_BlockDevTrans--- Issue Block Transaction

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R2_BlockDevTrans (BLOCK_DEV_TRANS_STRUCT *p,UBYTE *RespStatus);
```

Description

R2_BlockDevTrans() starts up a block device transaction.

Root-2 introduces new capabilities for generating high speed (and high bandwidth) USB transactions. Because these transactions can occur much more rapidly than at full or low speeds, Root-2 can transfer large blocks of data without software intervention. With R2_BlockDevTrans(), you can download up to 510K of data and command Root-2 to send this to your device using specific transfer parameters set to any speed condoned by USB 2.0 – or, conversely, you can command Root-2 to gather up to 510K of data from your device and upload it when it is done.

Significant communication delays may be observed in moving the potentially large amounts of data (510K) to and from Root-2: notably, with a serial connection at lower baud rates. However, these delays occur outside the time frame of the actual transactions to/from your device.

R2_BlockDevTrans() initiates the block transfer and the call returns immediately. Since the actual USB traffic generated by the command may take some time, the status of the transfer can be polled using the R2_BlockDevTransStatus() command, which is described subsequently.

Using R2_BlockDevTrans(), you can accomplish such tasks as:

Send or receive up to 510K of data in a single command to your device, at high bandwidth rates, using either Bulk or Isochronous OUTs or INs. Root-2 takes care of data toggling, retrying NAKed packets, and transitioning from PINGs to OUTs in the case of bulk OUT transactions.

Loop a transfer indefinitely, with the same capabilities as in the single block case. You can specifiy a transfer of up to 510K of data and Root-2 will restart the transfer once it completes, until an error occurs or you send another command to stop the looping function.

Because there are many parameters to this command, parameter passing takes place via a pointer to a structure which is filled in prior to calling. This structure is defined as follows:

```
typedef struct
{
      UBYTE Address;                //device address
      UBYTE Endpoint;               //endpoint of transfer
      UBYTE Pid;                    //pid: IN, OUT, PING, SETUP
      UBYTE DataPid;                //data pid: DATA1,DATA2,DATA0,MDATA
      UWORD Control;                //see below
      UWORD ServiceInterval;        //frame interval
      UBYTE PacketsPerMicroframe;   //microframe interval
      UWORD Packetsize;             //packet size
      UBYTE *DataOut;               //pointer to optional data to send
      ULONG Datalen;                //maximum length of data to
                                    //send or receive
}BLOCK_DEV_TRANS_STRUCT;
```

Set the Address and endpoint fields to the corresponding address and endpoint of your test device.

Set the Pid field to establish the direction of the block transfer. Setting this field to IN will initiate a transfer from the device to Root-2; setting it to PING, OUT, or SETUP will initiate a transfer in the opposite direction.

In the IN case, set the Datalen field to the amount of data you want to collect from the device (the data is retrieved using a separate command BlockDevTransStatus() -- see below). Set the DataPid field to the initial expected data pid which will be returned from the device. Root-2 will confirm correct data pid toggling for subsequent INs of the transfer. If you are issuing multiple Block IN commands in a row, you can set the Control flag DT_USE_NEXT_DATAPID for the second and subsequent block commands. Root2 will continue datapid toggle checking in the correct sequence, freeing you from having to precalculate the initial expected pid each time. See also the command R2_GetNextDataPid().

In the OUT/PING case, set the Datalen field to the amount of data you are sending; set the DataOut pointer to the buffer containing this data. Set the DataPid field to the intial data pid you want to send with the first data packet; Root-2 will handle data toggling for successive packets. If you are issuing multiple Block OUT commands in a row, you can set the Control flag DT_USE_NEXT_DATAPID for the second and subsequent block commands. Root2 will continue datapid toggling in the correct sequence, freeing you from having to precalculate the correct DataPid field each time. See also the command R2_GetNextDataPid().

Use the ServiceInterval field to establish the rate of transfer. For full and low speed devices, set this field to the frequency of frames for which a transfer will occur (1 = every frame, 2 = every other frame, etc). For high speed devices, this number refers to microframes: 1 = every microframe, 2 = every other microframe, etc.

In addition, for high speed, high bandwidth transfers, use PacketsPerMicroframe field to establish the number of transfers required per microframe: 1 = one transfer, 2 = two, etc. This can be set to a maximum of 32 per microframe.

Set the Packetsize field to any number up to 1024 bytes to establish the packet size for the transfer.

The Control field is used to specify other transfer mechanisms using the logical-or of the following flags:

```
DT_LOW_SPEED            //perform transaction at low speed
DT_FULL_SPEED           //perform transaction at full speed
DT_HIGH_SPEED           //perform transaction at high speed
DT_ISOCH                //perform isochronous transaction
DT_BULK                 //perform bulk transaction
DT_INTERRUPT            //perform interrupt transaction
DT_LOOP                 //loop this transaction indefinitely
DT_STOP_ON_NAK          //stop transaction when device NAKs
DT_SPLIT                //force split transaction
DT_USE_NEXT_DATAPID     //use next data pid in transaction
```

Returns

BlockDevTrans returns a Comm Status Word. It also returns a status byte in the byte pointed to by RespStatus which indicates the success (0) or failure in launching the block command.

Example:

```
{
      //set up to transfer 128K bytes of data to a device
      //using 1k packets, isochronous, 3 packets per microframe.
      BLOCK_DEV_TRANS_STRUCT TestH;
      UBYTE MyDataBuffer[131072];
      int Status;
      TestH.Address = 2;
      TestH.Endpoint = 2;
      TestH.Pid = OUT_PID;
      TestH.DataPid = DATA2_PID;          //start with this pid
      TestH.ServiceInterval = 1;          //transfer every frame
      TestH.PacketsPerFrame = 3;          //and transfer 3 packets
                                          //every frame.
      TestH.Packetsize = 512;             //512 bytes per packet
      TestH.DataOut = &MyDataBuffer[0];   //pointer to buffer full
                                          //data to transfer.
      TestH.Datalen = 131072;             //128K of data
                                          //perform isochronously,high
                                          //speed
      TestH.Control = (DT_ISOCH | DT_HIGH_SPEED);
      Status = R2_BlockDevTrans (&TestH);
      //..
      //..
      //.. see description of R2_BlockDevTransStatus
      //.. on how to poll status.
      //.. for now, assume it will complete in 10
      //.. seconds.

      Sleep(10000);
      //..
      //..
      //now do a In using same addr and endpoint
      //just need to switch PID. This will do INs
      //until 128K of data is received or an error occurs.
      TestH.Pid = IN_PID;
      //everything else ok
      Status = R2_BlockDevTrans (&TestH);
      Sleep(10000);                       //let it run for 10 seconds
}
```

4.26 R2_BlockDevTransStatus --- Get BlockTransaction Status

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R2_BlockDevTransStatus (UBYTE *p,ULONG *RespLen);
```

Description

R2_BlockDevTransStatus() queries the status of a previously issued R2_BlockDevTrans() command.

R2_BlockDevTransStatus() always returns at least one byte in the buffer pointed to by the pointer argument p; If this returned byte is zero, the block command is still executing. If it is nonzero, the block command has completed.

If the block command was an OUT, the 2nd byte in the buffer contains the last response (ACK, NAK, etc) from the device.

If the block command was an IN, and the block transaction completed without errors, the 2nd byte in the buffer contains the datapid of the last transaction. Otherwise, the status byte will contain the cause of the termination (NAK, or error condition).

The pointer argument p should refer to a buffer large enough to receive the expected results of the block command. If the block command was an IN, set the initial contents of RespLen to the amount of data you expect will be returned, plus 2 for the status byte and "last response" byte. If the block command was a OUT, set the initial contents of RespLen to 2.

Returns

Comm Status Word

Example:

```
{
      //set up to transfer 128K bytes of data to a device
      //using 1k packets, isochronous, 3 packets per microframe.
      UBYTE MyOutBuffer[131072];
      UBYTE MyInBuffer[131072+2];         //note 2 extra bytes
      ULONG RespLen;
      int tries = 10;
      BLOCK_DEV_TRANS_STRUCT TestH;
      int Status;
      //set up to transfer 128K bytes of data to a device
      //using 1k packets, isochronous, 3 packets per microframe.
      TestH.Address = 2;
      TestH.Endpoint = 2;
      TestH.Pid = OUT_PID;
      TestH.DataPid = DATA2_PID;          //start with this pid
      TestH.ServiceInterval = 1;          //transfer every frame
      TestH.PacketsPerFrame = 3;          //and transfer 3 packets
                                          //every frame.
      TestH.Packetsize = 512;             //512 bytes per packet
      TestH.DataOut = &MyOutBuffer[0];    //pointer to buffer full
                                          //data to transfer.
      TestH.Datalen = 131072;             //128K of data

                                          //loop the transaction
                                          //endlessly
      TestH.Control = (DT_ISOCH | DT_HIGH_SPEED | DT_LOOP);

      Status = R2_BlockDevTrans (&TestH);
      Sleep(4000);                        //let it run a few seconds
      Status = R2_StopBlockDevTrans();
      Sleep(1000);                        //give it a second to stop
      RespLen = 2;                        //expecting  status byte
      Status = R2_BlockDevTransStatus(&MyInBuffer[0],&RespLen);
      if (!MyInBuffer[0])
            printf("command did not stop!!!!\r\n");
            //probably should wait longer and try again here….
      else
      {
            printf("transaction result:\t%02x\r\n",MyInBuffer[1]);
      }

}
```

4.27 R2_StopBlockDevTrans --- Stop A Block Transaction

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R2_ StopBlockDevTrans (UBYTE OPTION);
```

Description

R2_StopBlockDevTrans() stops a previously issued R2_BlockDevTrans() command.

Block Transactions can be halted at the end of the next individual packet transmission within a block, or at the end of the next block transmission.

The option byte can assume one of the following values:

```
STOP_AT_END_OF_PACKET   //stop block command at end of next packet
STOP_AT_END_OF_BLOCK    //stop block command at end of next block
```

Returns

Comm Status Word

Example:

See description of R2_BlockDevTransStatus.

4.28 R2_GetNextDataPid --- Get Next DataPid

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R2_GetNextDataPid(UBYTE *pid);
```

Description

R2_GetNextDataPid() returns the next expected data pid from a previous transaction. Your application can use this command to keep track of datapid sequencing to a particular endpoint.

For OUT transfers, R2_GetNextDataPid() returns the next datapid expected by the device for an OUT packet. For IN transfers, R2_GetNextDataPid() returns the next datapid expected by Root2 for an IN packet.

Returns

Comm Status Word

Example:

```
{
      // this example sends a 1K packet to the device using a block
      //transfer, then sends a single 64-byte OUT using the result
      //of a call to R2_GetNextDataPid() for the initial PID.

      //set up to transfer 1K bytes of data to a device
      //using 64 byte packets, bulk, full speed device
      UBYTE MyOutBuffer[1024];
      UBYTE MyInBuffer[1024+2];           //note 2 extra bytes
      ULONG RespLen;
      UBYTE NextPid;
      int tries = 10;
      BLOCK_DEV_TRANS_STRUCT TestH;
      int Status;
      //set up to transfer 1K bytes of data to a device
      //using 64 byte packets.
      TestH.Address = 2;
      TestH.Endpoint = 2;
      TestH.Pid = OUT_PID;
      TestH.DataPid = DATA0_PID;          //start with this pid
      TestH.ServiceInterval = 1;          //transfer every frame
      TestH.PacketsPerFrame = 8;          //and transfer 8 packets
                                          //every frame.
      TestH.Packetsize = 64;              //512 bytes per packet
      TestH.DataOut = &MyOutBuffer[0];    //pointer to buffer full
                                          //data to transfer.
      TestH.Datalen = 1024;               //128K of data
                                          //loop the transaction
                                          //endlessly
      TestH.Control = (DT_BULK | DT_FULL_SPEED);

      Status = R2_BlockDevTrans (&TestH);
      Sleep(4000);                        //let it run a few seconds
      Status = R2_StopBlockDevTrans();
      Sleep(1000);                        //give it a second to stop
      RespLen = 2;                        //expecting  status byte
      Status = R2_BlockDevTransStatus(&MyInBuffer[0],&RespLen);
      if (!MyInBuffer[0])
            printf("command did not stop!!!!\r\n");
            //probably should wait longer and try again here….
      else
      {
            printf("transaction result:\t%02x\r\n",MyInBuffer[1]);
            Status = R2_GetNextDataPid(&NextPid);
            //issue a single 64-byte OUT, using the next expected data
            //pid
            R1_DevTransOut(2,2,OUT_PID,DT_FULL_SPEED,NextPid,
            MyOutBuffer64,&Status);
      }
}
```

4.29 R2_SplitDef --- Set up Split Transaction Parameters

Synopsis

```
#include "stddefs.h"
#include "rootcomm.h"


UWORD R2_ SplitDef (UBYTE HubAddress,UBYTE portID);
```

Description

R2_SplitDef() presets the Hub address and port ID for a subsequent DevTrans command to be issued to a low or full speed device downstream of a hub. See section 3.16 of the Root2 Interface Spec, and the examples for device transaction commands in this document, for further details.

Returns

Comm Status Word

Example:

```
{
      int Status = R2_SplitDef(2,4);       //set hub address to 2
                                           //port to 4
                                           //for split transaction
}
```