USBUART Datasheet USBUART V 1.60

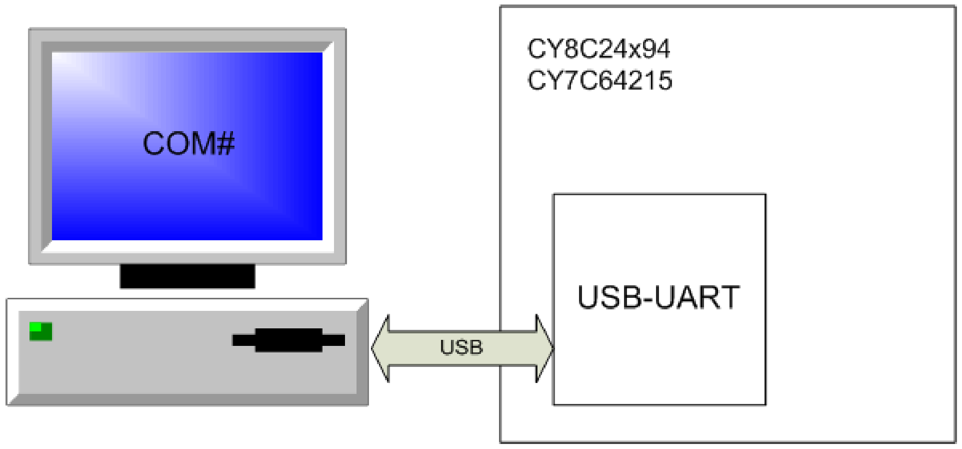
Supported Devices	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	Flash	RAM	
CY8C24x94, CY7C64215, CY8CLED04, CY8CTST120, CY8CTMG120, CY8CTMA120, CY8C20396A, CY8C20496A/L, CY8C20646A/AS/L, CY8C20666A/AS/L, CY7C643xx, CYRF89235, CY8C24493	0	0	0	1900	60	2

Note SysClk*2 is always needed for proper USB timing. In Global Resources, set **SysClk*2 Disable** to **No** for proper USB operation.

Features and Overview

- The USBUART device uses a USB interface to emulate a COM port.
- UART-like high level functions are available on the PSoC device side.

Figure 1. USBUART Device Block Diagrams



Functional Description

Many embedded applications use the RS-232 interface to communicate with external systems such as PCs, especially when debugging. But in the PC world, the RS-232 COM port will soon disappear from most new computers, leaving USB as the replacement for serial communication. The simplest way to migrate a device to USB is to emulate RS-232 over the USB bus. The primary advantage of this method is that PC applications use the USB connection as an RS-232 COM connection, making it very simple to debug. This method uses a standard Windows® driver that is included with all versions Microsoft® Windows from Windows 98SE through Windows XP.

The USB Communication Device Class (CDC) specification defines many communication models, including an abstract control model for serial emulation over USB in Section 3.6.2.1. See the CDC Specification version 1.1 for details. The Microsoft Windows USB modem driver, usbser.sys, conforms to this specification.

When a new device connects to a Windows PC the first time, Windows asks you to give a driver. An INF file is required to install drivers on Windows 2000 and later. Microsoft Windows does not give a standard INF file for the usbser.sys driver. To install a device that emulates RS-232 over USB, you must supply an INF file that maps the attached device to the Microsoft CDC drivers. The necessary INF file for USBUART projects is generated automatically and is located in the project LIB folder. After supplying the INF file, the driver allows the USB device to be enumerated as a COM port.

The settings in a terminal application (baud rate, data bits, parity, stop bits, and flow control) do not affect the performance of data transmissions because it is a USB device and the USB protocol is used to control data flow. However, the terminal settings with the exception of flow control can be retrieved with specific API calls to use with an RS-232 device if needed. The flow control setting cannot be retrieved because it is not supported by Microsoft's CDC driver, usbser.sys.r.

Use the following API calls to retrieve specific settings:

- USBUART_dwGetDTERate
- USBUART_bGetCharFormat
- USBUART_bGetParityType
- USBUART_bGetDataBits
- USBUART_bGetLineControlBitmap

USB Compliance

USB drivers may present various bus conditions to the device, including Bus Resets, and different timing requirements. Not all of these can be correctly illustrated in the examples given. You must design applications that conform to the USB spec.

USB Compliance for Self-Powered Devices

In the *USB Compliance Checklist* there is a question that reads, “Is the device’s pull up active only when V_{BUS} is high?”

The question lists Section 7.1.5 in the *Universal Serial Bus Specification Revision 2.0* as a reference. This section reads, in part, “The voltage source on the pull up resistor must be derived from or controlled by the power supplied on the USB cable such that when V_{BUS} is removed, the pull up resistor does not supply current on the data line to which it is attached.”

If the device that you are creating is self powered, you must connect a GPIO pin to V_{BUS} through a resistive network and write firmware to monitor the status of the GPIO. Application Note AN15813, *Monitoring the EZ-USB FX2LP VBUS*, explains the necessary hardware and software components required. You can use the USBFS_Start() and USBFS_Stop() API routines to control the D+ and D- pin pull ups. The pull up resistor does not supply power to the data line until you call USBFS_Start(). USBFS_Stop() disconnects the pull up resistor from the data pin.

Section 9.1.1.2 in the *Universal Serial Bus Specification Revision 2.0* says, “Devices report their power source capability through the configuration descriptor. The current power source is reported as part of a device’s status. Devices may change their power source at any time, for example, from self to bus-powered.” The device responds to GET_STATUS requests based on the status set with the USBFS_SetPowerStatus() function. To set the correct status, USBFS_SetPowerStatus() should be called at least once if your device is configured as self powered. You should also call the USBFS_SetPowerStatus() function any time your device changes status.

Sending Packets of Length 64 Bytes or More

In this user module, the USB endpoint buffer size is set to 64 bytes for incoming and outgoing data. It means that all write and read APIs are limited to work with a 64-byte buffer length.

In addition, there is a limitation to send exactly 64 bytes of data. The PC side driver accepts a packet as fully received if the payload size is less than 64, or transfers a zero-length packet. If all 64 bytes are received, the PC driver assumes that it has not received all the data and immediately asks for the next packet. The following code gives the successful transfer of data to PC immediately after function call:

```
while (!USBUART_bTxIsReady());
USBUART_Write(pData, 61); //Length is less than 64 data appear on
//terminal application after this function is executed.
```

The following code is an example that shows how to send exactly 64 bytes of data:

```
while (!USBUART_bTxIsReady());
USBUART_Write(pData, 64); //Length is equal to 64
while (!USBUART_bTxIsReady());
USBUART_Write(pData, 0); //No actual data transfer, but zero-length packet
// indicates PC driver that data is completely received.
// 64 bytes of data appears on terminal application
// after this function is executed.
```

Note that sending a 64 byte length packet with the following zero-length packet is faster than sending two packets of 32 bytes. This happens because a full 64 bytes-length packet forces the driver to continue the transfer.

The following code is an example that shows a transfer of 150 bytes:

```
while (!USBUART_bTxIsReady());
USBUART_Write(pData, 64); //Length is equal to 64
while (!USBUART_bTxIsReady());
USBUART_Write(pData, 64); //Length is equal to 64. 128 bytes is transferred to
// PC. No data appears on terminal application at the
// moment
while (!USBUART_bTxIsReady());
USBUART_Write(pData, 2); //Two bytes transferred. Full 130 bytes-length packet
//appears on terminal application after this function
// is executed.
```

Timing

The USBUART Device User Module supports USB 2.0 Full Speed operation on the CY8C24x94and CY7C64215 devices.

Placement

The USBUART User Module is a software only user module and does not consume device resources other than flash and RAM.

Parameters and Resources

Vendor ID

Each USB product must have a unique combination of vendor ID (VID) and product ID (PID). This 2-byte string contains the VID. The USB Implementers Forum assigns the VIDs.

Product ID

Each USB product must have a unique combination of VID and PID. This 2-byte string contains the PID. The manufacturer assigns the PIDs and these must be unique to the product.

VendorString

A free form string describing the manufacturer of the product. Do not use apostrophes (‘) in the VendorString.

ProductString

A free form string describing the product. Do not use apostrophes (‘) in the ProductString.

SerialNumberType

Choose the type of serial number. The possible settings are listed in the following table:

Parameter	Description
None	This device have no serial number. Value entered in SerialNumberString parameter is ignored.
Automatic	The serial number is automatically generated from PSoC device serialization number. Serial number is 24 hex characters. Value entered in SerialNumberString parameter is ignored.
Manual	Used value entered in the SerialNumberString parameter.

SerialNumberString

Sets the serial number for the device. Is recommended to use numeric value. Applied only if Serial-NumberType parameter is set to Manual.

DevicePower

Choose the device power source. The device can be self powered or powered from the USB.

MaxPower

Set the power (in mA) consumed from the USB bus when the device is powered from the USB bus. If the device is self powered this parameter is ignored. The minimum is 1 mA and the maximum is 500 mA. Normally, you can set this to 100 mA for a low power device or 500 mA for a high power device.

Application Programming Interface

The Application Programming Interface (API) routines are given as part of the user module to allow you to deal with the module at a higher level. This section specifies the interface to each function together with related constants given by the include files.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns the USBUART_1 to the first instance of this user module in a given project. It can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of every global function name, variable, and constant symbol. In the following descriptions the instance name has been shortened to USBUART for simplicity.

Note

In this, as in all user module APIs, you can alter the values of the A and X register by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This “registers are volatile” policy is selected for efficiency and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must ensure their code observes the policy, too. Though some user module API functions may leave A and X unchanged, there is no guarantee they may do so in the future

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

The following table lists the USBUART supplied API functions:

Table 1. USBUART API

Function	Description
void **USBUART_Start**(BYTE bVoltage)	Enable the user module for use with the device.
void **USBUART_Stop**(void)	Disable the user module.
BOOL **USBUART_Init**(void)	Initialize the USBUART module. Returns a nonzero value if the USBUART is successfully initialized.
void **USBUART_Write**(BYTE * pData, BYTE bLength)	Sends bLength bytes from pData array to the PC.
void **USBUART_CWrite**(const BYTE * pData, BYTE bLength)	Sends bLength bytes from constant (ROM) pData array to the PC.
void **USBUART_PutString**(BYTE * pStr)	Sends a NULL terminated string pStr to the PC.
void **USBUART_CPutString**(const BYTE * pStr)	Sends a constant (ROM) NULL terminated string pStr to the PC.
void **USBUART_PutChar**(BYTE bChar)	Sends one character to the PC
void **USBUART_PutCRLF**(void)	Sends a carriage return (0x0D) and a line feed (0x0A) to the PC.
void **USBUART_PutSHexByte**(BYTE bValue)	Sends a two character hex representation of bValue to the PC.
void **USBUART_PutSHexInt**(INT iValue)	Sends a four character hex representation of iValue to the PC.
BYTE **USBUART_bGetRxCount**(void)	Returns the current byte count ready for read.
BYTE **USBUART_bTxIsReady**(void)	Returns a nonzero value if USBUART is ready to send data.

Function	Description
BYTE **USBUART_Read**(BYTE * pData, BYTE bLength)	Reads the specified number of bytes from the RX buffer and places it in the RAM array specified by pData. The function returns the number of bytes remaining in RX buffer and operation status.
void **USBUART_ReadAll**(BYTE * pData)	Reads all available data from the RX buffer and places it in the RAM array specified by pData.
WORD **USBUART_ReadChar**(void)	Returns one byte from the RX buffer in the LSB of the return value. The function also returns the operations status and number of bytes remaining in the RX buffer in the MSB of the return value.
BYTE **USBUART_bCheckUSBActivity**(void)	Checks and clears the USB Bus Activity Flag. Returns a one if the USB was active since the last check, otherwise returns zero.
DWORD ***USBUART_dwGetDTERate**(DWORD * dwDTERate)	Returns the data terminal rate set for this port in bits per second.
BYTE **USBUART_bGetCharFormat**(void)	Returns the number of stop bits.
BYTE **USBUART_bGetParityType**(void)	Returns the parity type.
BYTE **USBUART_bGetDataBits**(void)	Returns the number of data bits.
BYTE **USBUART_bGetLineControlBitmap**(void)	Returns the DTE and RTS signal state.
void **USBUART_SendStateNotify**(BYTE bState)	Sends notification about the current UART state to the PC.
void **USUART_SetPowerStatus**(BYTE bPowerStatus)	Sets the device to self powered or bus powered

USBUART_Start

Description:

Performs all required operations to start the USBUART Device User Module.

C Prototype:

```
void USBUART_Start(BYTE bVoltage)
```

Assembly:

```
mov   A, USBUART_5V_OPERATION   ; Select the Voltage level
lcall  USBUART_Start             ; Call the Start Function
```

Parameters:

bVoltage is the operating voltage of the chip, passed in the Accumulator. This determines whether the voltage regulator is enabled for 5 V operation or pass through mode is used for 3.3 V operation. Symbolic names are given in C and assembly, and their associated values are listed in the following table.

Mask	Value	Description
USBUART_3V_OPERATION	0x02	Disable the voltage regulator and pass-through Vcc for pull up
USBUART_5V_OPERATION	0x03	Enable the voltage regulator and use the regulator for pull up

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_Stop

Description:

Performs all necessary shutdown tasks required for the USBUART Device User Module.

C Prototype:

```
void  USBUART_Stop(void)
```

Assembly:

```
lcall  USBUART_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the CUR_PP page pointer register is modified.

USBUART_Init

Description:

Try to initialize the USBUART device and set up communication with the PC.

C Prototype:

```
BOOL  USBUART_Init(void)
```

Assembly:

```
lcall  USBUART_Init
```

Parameters:

None

Return Value:

Returns a nonzero value in the accumulator if the device initializes successfully. Returns a 0 if initialization failed. The user module can operate only after successful initialization.

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and CUR_PP page pointer registers is modified.

USBUART_Write

Description:

Sends bLength characters from the location specified by (RAM) pointer pData to the PC. Refer to Sending Packets of Length 64 Bytes and More when sending large packets.

C Prototype:

```
void  USBUART_Write(BYTE * pData, BYTE bLength)
```

Assembly:

```
mov   A,20                 ; Load array count
push  A
mov   A,>pData             ; Load MSB part of pointer to RAM string
push  A
mov   A,<pData             ; Load LSB part of pointer to RAM string
push  A
lcall  USBUART_Write        ; Make call to function
add   SP,253               ; Reset stack pointer to original position
```

Parameters:

pData is a pointer to a data array. The maximum length of the data array is 64 bytes.

bLength is the number of bytes to be transferred from the array and sent to the PC. Valid values are between 0 and 64.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_CWrite

Description:

Sends bLength characters from the location specified by (ROM) pointer pData to the PC. Refer to Sending Packets of Length 64 Bytes and More when sending large packets.

C Prototype:

```
void  USBUART_CWrite(const BYTE * pData, BYTE bLength)
```

Assembly:

```
mov   A,20                ; Load array count
push  A
mov   A,>pData            ; Load MSB part of pointer to ROM string
push  A
mov   A,<pData            ; Load LSB part of pointer to ROM string
push  A
lcall  USBUART_CWrite      ; Make call to function
add   SP,253              ; Reset stack pointer to original position
```

Parameters:

pData is a pointer to a data array in ROM. The maximum length of the data array is 64 bytes.

bLength is the number of bytes to be transferred from the array and sent to the PC. Valid values are between 0 and 64.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_PutString

Description:

Sends a null terminated (RAM) string to the PC. Refer to Sending Packets of Length 64 Bytes and More when sending large packets.

C Prototype:

```
void USBUART_PutString(BYTE * pStr)
```

Assembler:

```
mov  A,>pStr              ; Load MSB part of pointer to RAM based null
                          ; terminated string
mov  X,<pStr              ; Load LSB part of pointer to RAM based null
                          ; terminated string
lcall USBUART_PutString    ; Call function to send string out
```

Parameters:

pStr: Pointer to the string to be sent to PC. The MSB is passed in the Accumulator and the LSB is passed in the X register. The maximum string length is 64 bytes including the terminating null character.

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_CPutString

Description:

Sends a null terminated (ROM) string to the PC. Refer to Sending Packets of Length 64 Bytes and More when sending large packets.

C Prototype:

```
void USBUART_CPutString(const BYTE * pStr)
```

Assembler:

```
mov   A,>pStr             ; Load MSB part of pointer to ROM based null
                          ; terminated string
mov   X,<pStr             ; Load LSB part of pointer to ROM based null
                          ; terminated string
lcall USBUART_PutString   ; Call function to send string out
```

Parameters:

pStr: Pointer to the string to be sent to the PC. The MSB is passed in the accumulator and the LSB is passed in the X register. The maximum string length is 64 bytes including the terminating null character.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_PutChar

Description:

Writes a single character to the PC.

C Prototype:

```
void USBUART_PutChar(BYTE bChar)
```

Assembler:

```
mov   A,0x33             ; Load ASCII character "3" in A
lcall USBUART_PutChar    ; Call function to send single character to PC
```

Parameters:

bChar: Character to be sent to the PC. Data is passed in the Accumulator.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_PutCRLF

Description:

Sends a carriage return (0x0D) and line feed (0x0A) to the PC.

C Prototype:

```
void USBUART_PutCRLF(void)
```

Assembler:

```
lcall USBUART_PutCRLF       ; Send a carriage return and line feed out
```

Parameters:

None

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_PutSHexByte

Description:

Sends a two byte ASCII Hex representation of the data to the PC.

C Prototype:

```
void USBUART_PutSHexByte(BYTE bValue)
```

Assembler:

```
mov  A,0x33                ; Load data to be sent
lcall USBUART_PutSHexByte   ; Call function to output hex representation of
                           ; data. The output for this value would be "33".
```

Parameters:

bValue: Byte to be converted to an ASCII string (hex representation). Data is passed in the Accumulator.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_PutSHexInt

Description:

Sends a four byte ASCII hex representation of the data to the PC.

C Prototype:

```
void USBUART_PutSHexInt(INT iValue)
```

Assembler:

```
mov   A,0x34            ; Load LSB in A
mov   X,0x12            ; Load MSB in X
lcall UART_PutSHexInt   ; Call function to output hex representation of data.
                        ; The output for this value would be "1234".
```

Parameters:

iValue: Integer to be converted to ASCII string (hex representation). The MSB is passed in the X register and the LSB is passed in Accumulator.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_bGetRxCount

Description:

This function returns the number of bytes that were received from the PC and are waiting in the RX buffer.

C Prototype:

```
BYTE  USBUART_bGetRxCount(void)
```

Assembly:

```
lcall  USB_bGetEPCount
```

Parameters:

None

Return Value:

Returns the current byte count in A.

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP page pointer register are modified.

USBUART_bTxIsReady

Description:

Returns a nonzero value if the TX buffer is ready to send more data. Otherwise it returns zero.

C Prototype:

```
BYTE  USBUART_bTxIsReady(void)
```

Assembly:

```
lcall  USBUART_bTxIsready
```

Parameters:

None

Return Value:

If TX buffer can accept data then this function returns a nonzero value. Otherwise a zero is returned.

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

USBUART_Read

Description:

Reads bLength bytes of received data from the RX Buffer and places it in a data array specified by pData.

C Prototype:

```
BYTE  USBUART_Read(BYTE * pData, BYTE bLength)
```

Assembly:

```
mov A, 25               ; Load count
push A
mov A, >pData           ; Load MSB part of pointer to RAM array
push A
mov A, <pData           ; Load LSB part of pointer to RAM array
push A
lcall  USBUART_Read
```

Parameters:

pData is a pointer to a data array. Maximum length of the data array is 64 bytes.

bLength is the number of bytes to be read to the array. Valid values are between 0 and 64.

Return Value:

Returns the number of bytes remaining in the RX buffer using bit 0..6 of the Accumulator and the MSb (bit 7) of the Accumulator indicates an error condition. Error conditions usually occur when you request more bytes than are available in the buffer. The data from the RX buffer is placed in the data array specified by pData.

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_ReadAll

Description:

Reads all bytes of received data from the RX buffer and places it in a data array specified by pData.

C Prototype:

```
void  USBUART_ReadAll(BYTE * pData)
```

Assembly:

```
mov  A,>pData           ; Load MSB part of pointer to RAM buffer
mov  X,<pData           ; Load LSB part of pointer to RAM buffer
lcall  USBUART_ReadAll
```

Parameters:

pData is a pointer to a data array. The MSB is passed in the Accumulator and the LSB is passed in the X register. The maximum size of the data array is 64 bytes.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_ReadChar

Description:

Reads one byte of received data from the RX Buffer.

C Prototype:

```
WORD  USBUART_ReadChar(void)
```

Assembly:

```
lcall  USBUART_ReadChar
```

Parameters:

None

Return Value:

The MSB of the returned value (Accumulator) contains the number of bytes remaining in the RX buffer using bits 0..6. Bit 7 indicates error status. Bit 7 is set to one if the buffer is empty when the function is called. The LSB of the returned value (X) contains a character from buffer.

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_bCheckUSBActivity

Description:

Checks for activity on the USB.

C Prototype:

```
BYTE  USBUART_bCheckUSBActivity(void)
```

Assembly:

```
lcall  USB_bCheckUSBActivity
```

Parameters:

None

Return Value:

Returns one in the Accumulator if the USB was active since the last check, otherwise returns zero.

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

USBUART_dwGetDTERate

Description:

Returns the data terminal rate set for this port in bits per second. Pass the function a pointer to a DWORD. The function returns the DTE rate in the location referenced by the pointer.

C Prototype:

```
DWORD * USBUART_dwGetDTERate(DWORD * dwDTERate)
```

Assembly:

```
mov  A,>dwDTERate       ; Load MSB part of pointer
mov  X,<dwDTERate       ; Load LSB part of pointer
lcall  USBUART_dwGetDTERate
```

Parameters:

dwDTERate: A pointer to where the DTE rate is stored when the function returns.

Return Value:

Stores the DTE rate DWORD value in the location referenced by the pointer it was passed, and then returns a pointer to that location.

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the IDX_PP and the CUR_PP page pointer registers are modified.

USBUART_bGetCharFormat

Description:

Returns the number of stop bits.

C Prototype:

```
BYTE  USBUART_bGetCharFormat(void)
```

Assembly:

```
lcall  USBUART_bGetCharFormat
```

Parameters:

None

Return Value:

Returns number of stop bits in Accumulator. Symbolic names given in C and assembly, and their associated values are listed in the following table.

Mask	Value	Description
USBUART_1_STOPBITS	0x00	1 stop bit
USBUART_1_5_STOPBITS	0x01	1.5 stop bits
USBUART_2_STOPBITS	0x02	2 stop bits

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the CUR_PP page pointer registers are modified.

USBUART_bGetParityType

Description:

Returns the parity type.

C Prototype:

```
BYTE  USBUART_bGetParityType(void)
```

Assembly:

```
lcall  USBUART_bGetParityType
```

Parameters:

None

Return Value:

Returns the parity type in Accumulator. Symbolic names given in C and assembly, and their associated values are listed in the following table.

Mask	Value	Description
USBUART_PARITY_NONE	0x00	No parity
USBUART_PARITY_ODD	0x01	Parity odd
USBUART_PARITY_EVEN	0x02	Parity even
USBUART_PARITY_MARK	0x03	Mark parity
USBUART_PARITY_SPACE	0x04	Space parity

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the CUR_PP page pointer registers are modified.

USBUART_bGetDataBits

Description:

Returns the number of data bits.

C Prototype:

```
BYTE  USBUART_bGetDataBits(void)
```

Assembly:

```
lcall  USBUART_bGetDataBits
```

Parameters:

None

Return Value:

Returns the number of data bits in the Accumulator. The number can be 5, 6, 7, 8 or 16.

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the CUR_PP page pointer registers are modified.

USBUART_bGetLineControlBitmap

Description:

Returns a bitmap with the state of the RS-232 style control signal.

C Prototype:

```
BYTE  USBUART_bGetLineControlBitmap(void)
```

Assembly:

```
lcall  USBUART_bGetLineControlBitmap
```

Parameters:

None

Return Value:

Returns a bitmap with the state of the control signal in the Accumulator. Each bit of the bitmap can be treated individually. Bits D7..D2 are reserved. Symbolic names are given in C and assembly, and their associated values are listed in the following table.

Mask	Value	Description
USBUART_RTS	0x02	RTS (1 – activate carrier; 0 – deactivate carrier)
USBUART_DTR	0x01	DTR (1 – present; 0 – not present)

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the CUR_PP page pointer registers are modified.

USBUART_SendStateNotify

Description:

Sends notification to the PC about the UART status.

Note The Microsoft usbser.sys driver does not support these signals.

C Prototype:

```
void  USBUART_SendStateNotify(BYTE bState)
```

Assembly:

```
mov  A, (USBUART_DCD + USBUART_DSR)
lcall  USBUART_SendStateNotify
```

Parameters:

bState bitmap with the state of the control signal in Accumulator. Each of the bits in the bitmap can be treated individually. Symbolic names given in C and Assembly, and their associated values are listed in the following table:

Mask	Value	Description
USBUART_DCD	0x01	RS-232 DCD signal
USBUART_DSR	0x02	RS-232 DSR signal
USBUART_BREAK	0x04	State of the break detection mechanism
USBUART_RING	0x08	State of the ring detection signal.
USBUART_FRAMING_ERR	0x10	A framing error has occurred.
USBUART_PARITY_ERR	0x20	A parity error has occurred.
USBUART_OVERRUN	0x40	Received data has been discarded due to an overrun.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. The same is true for all RAM page pointer registers in the large memory model. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently only the CUR_PP and IDX_PP page pointer registers are modified.

USBUART_SetPowerStatus

Description:

Sets the current power status. Set the power status to one for self powered or zero for bus powered. The device replies to USB GET_STATUS requests based on this value. This allows the device to properly report its status for USB Chapter 9 compliance. Devices may change their power source from self powered to bus powered at any time and report their current power source as part of the device status. You should call this function any time your device changes from self powered to bus powered or vice versa, and set the status appropriately.

C Prototype:

```
void USBUART_SetPowerStatus(BYTE bPowerStaus);
```

Assembly:

```
MOV   A, USB_DEVICE_STATUS_SELF_POWERED      ; Select self powered
lcall  USBUART_SetPowerStatus
```

Parameters:

bPowerStatus contains the desired power status, one for self powered or zero for bus powered. Symbolic names are given in C and assembly, and their associated values are listed here:

State	Value	Description
USB_DEVICE_STATUS_BUS_POWERED	0x00	Set the device to bus powered.
USB_DEVICE_STATUS_SELF_POWERED	0x01	Set the device to self powered.

Return Value:

None

Side Effects:

This function or its future implementations can modify the A and X registers. This is true for all RAM page pointer registers in the Large Memory Model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions.

Sample Firmware Source Code

The following code illustrates how to use the USBUART User Module in a simple application. When a new device connects to a Windows PC for the first time, Windows asks you to give a driver. An INF file is required to install drivers on Windows 2000 and later. Microsoft Windows does not give a standard INF file for the usbser.sys driver supplied with Windows. In order to install a device that emulates RS-232 over USB, you must supply an INF file that maps the attached device to the Microsoft usbser.sys driver. The necessary INF file for USBUART projects is generated automatically and is located in the project LIB folder. Use this INF file to install the device. Once the driver is installed, this device enumerates as a COM port and simply echoes any received symbol back to the PC.

```
//
//  This sample shows how to use The USBUART UM to emulate a COM port
//
// OVERVIEW:
//
//The following changes need to be made to the default settings in the Device Editor:
//
//   1. Select USBUART user module.
//   2. Rename User Module's instance name to USBUART.
//
// CONFIGURATION DETAILS:
//
//   The UM's instance name must be shortened to USBUART.
//
// USER MODULE PARAMETER SETTINGS:
//
//  -------------------------------------------------------------------------------
//  UM          Parameter         Value                  Comments
//  -------------------------------------------------------------------------------
//  USBUART     Name             USBUART                UM's instance name
//  -------------------------------------------------------------------------------

/* Code begins here */

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

BYTE Len;
BYTE pData[32];

void main(void)
{
    M8C_EnableGInt;                           //Enable Global Interrupts
    USBUART_Start(USBUART_5V_OPERATION);    //Start USBUART 5 V operation
      while(!USBUART_Init());                 //Wait for Device to initialize
    while(1)
    {
        Len = USBUART_bGetRxCount();       //Get count of ready data
        if (Len)
        {
            USBUART_ReadAll(pData);        //Read all data rom RX
            while (!USBUART_bTxIsReady()); //If TX is ready
            USBUART_Write(pData, Len);     //Echo
        }
    }
}
```

The equivalent code written in Assembly is:

```
;
;  This sample shows how to use The USBUART UM to emulate a COM port
;
; OVERVIEW:
;
;The following changes need to be made to the default settings in the Device Editor:
;
;   1. Select USBUART user module.
;   2. Rename User Module's instance name to USBUART.
;
; CONFIGURATION DETAILS:
;
;   The UM's instance name must be shortened to USBUART.
;
; USER MODULE PARAMETER SETTINGS:
;
;  ---------------------------------------------------------------------------------
;  UM         Parameter          Value                  Comments
;  ---------------------------------------------------------------------------------
;  USBUART    Name               USBUART                UM's instance name
;  ---------------------------------------------------------------------------------

;  Code begins here
include "m8c.inc"       ; part specific constants and macrosinclude "memory.inc"    ;
Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"   ; PSoC API definitions for all user modules

AREA bss (RAM, REL)

Len: blk 1
pData: blk 32

export _main

AREA text (ROM, REL)

_main:

    M8C_EnableGInt          ; Enable Global Interrupts

    mov   A, USBUART_5V_OPERATION
    lcall USBUART_Start      ; Start USBUART 5 V operation

deviceInit:                  ; Wait for Device to initialize
    lcall USBUART_Init
    cmp   A,0
    jz    deviceInit

mainLoop:
    lcall USBUART_bGetRxCount
    mov   [Len],A            ; Get count of ready data
    cmp   [Len],0            ; Check if Len is 0
    jz    mainLoop

    mov   A,>pData           ; Load MSB part of pointer to RAM buffer
    mov   X,<pData           ; Load LSB part of pointer to RAM buffer
    call  USBUART_ReadAll    ; Read all data rom RX

txReady:
    lcall USBUART_bTxIsReady ; Check to see if TX is ready
    cmp   A,0
    jz    txReady
                             ; Echo data
    mov   A,[Len]            ; Load array count
    push  A
    mov   A,>pData           ; Load MSB part of pointer to RAM string
    push  A
    mov   A,<pData           ; Load LSB part of pointer to RAM string
    push  A
    call  USBUART_Write
    add   SP,253             ; Reset stack pointer to original position
    jmp   mainLoop
```

Appendix 1: Configuration Registers (for Devices with Four Endpoints)

This section describes the PSoC Resource Registers used or modified by the USBUART Device User Module.

Table 2. Resource EP0_CNTL: Bank 0 reg[56] Endpoint0 Control Register

Bit	7	6	5	4	3	2	1	0
Value	Setup Received	IN Received	OUT Received	ACK'd Transaction	Mode			

Setup Received: When this bit is one, it indicates a valid SETUP packet has been received and ACKed. This bit is forced high from the start of the data packet phase of the SETUP transaction, until the start of the ACK packet returned by the SIE. The CPU is prevented from clearing this bit during this interval. After this interval, the bit remains set until cleared by firmware. While this bit is set to one, the CPU cannot write to the EP0_DRx registers. This prevents firmware from overwriting an incoming SETUP transaction before firmware has a chance to read the SETUP data. This bit is cleared by any nonlocked writes to the register.

IN Received: When this is one, it indicates a valid IN packet has been received. This bit is set to one after the host acknowledges an IN data packet. When zero, this bit indicates either that no IN has been received or that the host did not acknowledge the IN data by sending an ACK handshake. It is cleared to zero by any nonlocked writes to the register.

OUT Received: When this bit is one, it indicates a valid OUT packet has been received and ACKed. This bit is set to one after the last received packet in an OUT transaction. When zero this bit indicates no OUT packets have been received. It is cleared to zero by any nonlocked writes to the register.

ACK'd Transaction: This bit is one whenever the SIE engages in a transaction to the register's endpoint that completes with a ACK packet. This bit is zero by any nonlocked writes to the register.

Mode: The mode controls how the USB SIE responds to traffic and how the USB SIE changes the mode of that endpoint as a result of host packets to the endpoint.

Encoding	Mode	SETUP	IN	OUT	Comments
0h	Disable	Ignore	Ignore	Ignore	Ignore all USB traffic to this endpoint.
1h	NAK IN/OUT	Accept	NAK	NAK	NAK IN and OUT token.
2h	Status OUT Only	Accept	STALL	Check	For control endpoint, STALL IN and ACK zero byte OUT.
3h	Status IN/ OUT	Accept	STALL	STALL	For control endpoint, STALL IN and OUT token.
4h	Reserved	Ignore	Ignore	Ignore	
5h	ISO OUT	Ignore	Ignore	Always	Isochronous OUT.
6h	Status IN Only	Accept	TX 0 Byte	STALL	For control endpoint, STALL OUT and send zero byte data for IN token.
7h	ISO IN	Ignore	TX Count	Ignore	Isochronous IN.
8h	NAK OUT	Ignore	Ignore	NAK	Send NAK handshake to OUT token.
9h	ACK OUT (Stall = 0)	Ignore	Ignore	ACK	This mode is changed by the SIE to mode 8h on issuance of ACK handshake to an OUT.
9h	ACK OUT (Stall = 1)	Ignore	Ignore	STALL	STALL the OUT transfer.
Ah	Reserved	Ignore	Ignore	Ignore	
Bh	ACK OUT – Status IN	Accept	TX 0 Byte	ACK	ACK the OUT token or send zero byte data for IN token.
Ch	NAK IN	Ignore	NAK	Ignore	Send NAK handshake for IN token.
Dh	ACK IN (Stall = 0)	Ignore	TX Count	Ignore	The mode is changed by the SIE to mode Ch after receiving ACK handshake to an IN data.
Dh	ACK IN (Stall = 1)	Ignore	STALL	Ignore	STALL the IN transfer.
Eh	Reserved	Ignore	Ignore	Ignore	
Fh	ACK IN – Status OUT	Accept	TX Count	Check	Respond to IN data or Status OUT.

Table 3. Resource EPx_CNTL: Bank 1 reg[C4-C7] Endpoint1 – Endpoint4 Control Registers

Bit	7	6	5	4	3	2	1	0
Value	Stall0	Reserved	NakIntEnable	ACK'd Transaction	Mode			

Stall: When this bit is one, the SIE stalls an OUT packet if the Mode bits are set to ACK-OUT. The SIE stalls an IN packet if the mode bits are set to ACK-IN. This bit must be zero for all other modes.

NakIntEnable: When set to one, this bit causes an endpoint interrupt to be generated even when a transfer completes with a NAK.

ACK'd Transaction: This bit is one whenever the SIE engages in a transaction to the register's endpoint that completes with an ACK packet. This bit is zero after any writes to the register.

Mode: Same as EP0_CNTL.

Table 4. Resource EP0_CNT : Bank 0 reg[57] Enpoint0 Count Register

Bit	7	6	5	4	3	2	1	0
Value	Data Toggle	Data Valid	Reserved			Byte Count		

Data Toggle: This bit selects the DATA packet's toggle state. For IN transactions, firmware must set this bit. For OUT or SETUP transactions, the SIE hardware sets this bit to the state of the received Data Toggle bit.

Data Valid: This bit is used for OUT transactions only. It is cleared if CRC, bit stuff, or PID errors have occurred. This bit does not update for some endpoint mode settings. This bit may be cleared by writing a zero to it when the register is not locked.

Byte Count: These bits indicate the number of data bytes in a transaction. For IN transactions, firmware loads the count with the number of bytes to be transmitted to the host from the endpoint FIFO. Valid values are 0 to 8. For OUT or SETUP transactions, the count is updated by hardware to the number of data bytes received, plus two for the CRC bytes. Valid values are 2 to 10.

Table 5. Resource EPx_CNT0: Bank 0 reg[4F,51,53,55] Endpoint1 - Endpoint4 Count0 Registers

Bit	7	6	5	4	3	2	1	0
Value	Count LSb							

These bits are the eight LSb of a 9-bit counter. The MSb is the Count MSb of the EPx_CNT1 register.

The 9-bit count indicates the number of data bytes in a transaction. For IN transactions, firmware loads the count with the number of bytes to be transmitted to the host. Valid values are 0 to 256.

The lower eight bits of endpoint count also set the limit for the number of bytes that are received for an out transaction. Before an OUT transaction can be received for an endpoint, this count value must be set to the maximum number of bytes that can be received where 0x01 is 1 byte and 0xff is 255 bytes. If this count value is set to a value greater than the number of bytes received, both the data from the USB packet and the two-byte CRC are written to the USB's dedicated SRAM.

If the count value is less than the number of data bytes received, the SIE marks the packet as invalid and not generate an interrupt. For example, an eight byte data packet tries to write eight data bytes and two CRC bytes. A count value of eight or greater allows a good packet to generate an interrupt. A count value of seven or less causes the SIE to mark the packet as bad.

After the OUT transaction is complete, the full 9-bit count is updated by the SIE to the actual number of data bytes received by the SIE plus two for the packet's CRC. Valid values are 2 to 258.

Table 6. Resource EPx_CNT1: Bank 0 reg[4E,50,52,54] Endpoint1 - Endpoint4 Count1 Registers

Bit	7	6	5	4	3	2	1	0
Value	Data Toggle	Data Valid	Reserved					Count MSb

Data Toggle: This bit selects the DATA packet's toggle state. For IN transactions, firmware must set this bit to the expected state. For OUT transactions, the hardware sets this bit to the state of the received Data Toggle bit.

Data Valid: This bit is used for OUT transactions only and is read only. It is 0 if CRC, bit stuffing, or PID errors occur. This bit does not update for some endpoint mode settings.

Count MSb: This bit is the most significant bit of a 9-bit counter. The least significant bits are the EPx Count[7:0] bits of the EPx_CNT register. Refer to the EPx_CNTx register for more information.

Table 7. Resource EP0_DRx: Bank 0 reg[57-5F] Enpoint0 Data Register 0-7

Bit	7	6	5	4	3	2	1	0
Value	Data							

These registers are used to read and write data to the USB control endpoint. They are shared for both transmit and receive. The count in the EP0_CNT register determines the number of bytes received or to be transferred.

Table 8. Resource USB_CR0: Bank 0 reg[4A] USB Control Register 0

Bit	7	6	5	4	3	2	1	0
Value	USB Enable	Device Address						

USB Enable: When this bit is one it enables the SIE for USB traffic and the USB transceiver. The device does not respond to USB traffic if this bit is zero.

Device Address: The SIE responds to the USB device address specified by these bits. This address must be set by firmware and specified by the system with a SETUP command during USB enumeration.

Table 9. Resource USB_CR1: Bank 1 reg[C1] USB Control Register 1

Bit	7	6	5	4	3	2	1	0
Value	Reserved					Bus Activity	EnableLock	RegEnable

Bus Activity: This is a sticky bit that detects any nonidle USB event that has occurred on the USB bus. Once set to high by the SIE to indicate the bus activity, this bit retains its logical high value until firmware clears it. Writing a zero to this bit clears it; writing a one preserves its value.

EnableLock: Set this bit to one to turn on the automatic frequency locking of the internal oscillator for USB traffic. Unless an external clock is being given, this bit should remain set to one for proper USB operation.

RegEnable: This bit controls the operation of the internal USB regulator. For applications with PSoC supply voltages in the 5 V range, set this bit high to enable the internal regulator. For device supply voltages in the 3.3 V range, clear this bit to connect the transceiver directly to the supply.

Table 10. Resource USBIO_CR0: Bank 0 reg[4B] USB I/O Control Register 0

Bit	7	6	5	4	3	2	1	0
Value	TEN	TSE0	TD	Reserved				RD

TEN: Setting this bit allows the USB outputs to be driven manually. Normally, TEN is kept low so that the internal hardware can control traffic flow automatically. One application for manual USB mode is driving a resume signal (USB “K") to wake the system from USB suspend.

TSE0: This bit is used to manually transmit a single ended zero (both D+ and D- low) on the USB pins. This bit has no effect if TEN = 0.

TD: This bit is used to manually drive a USB J or K state onto the USB pins. There is no effect if TEN = 0, and TSE0 overrides this bit.

RD: This read only bit gives the state of USB Received Data from the differential receiver. The USB Enable bit in the USB_CR0 register must be set to receive data. If the USB Enable bit is not set, this bit reads zero.

Table 11. Resource USBIO_CR1: Bank 0 reg[4C] USB I/O Control Register 1

Bit	7	6	5	4	3	2	1	0
Value	I/O Mode	Drive Mode	DPI	DMI	PS2PUEN	USBPUEN	DPO	DMO

I/O Mode: This bit allows the D+ and D- pins to be configured for either USB mode or bit banged modes. If this bit is one, the DMI and DPI bits are used to drive the D- and D+ pins.

Drive Mode: If the IOMode bit is one, this bit configures the D- and D+ pins for either CMOS drive or open-drain drive. If IOMode is zero, this bit has no effect. Note that in open drain mode 5 k.

Appendix 2: Configuration Registers (for Devices with Eight Endpoints)

All used registers are the same as in the devices with four endpoints, except for the following differences:

1. Contents of EPx_CNT0 and EPx_CNT1 registers are swapped but all bit fields are the same.
2. An additional USB_MISC_CR register, which has the following bit fields, is used

Table 12. Resource USB_MISC_CR: Bank 1 reg[BD] USB Miscellaneous Control Register

Bit	7	6	5	4	3	2	1	0
Value	–	–	–	–	–	USB_SE_EN	USB_ON	USB_CLK_ON

USB_SE_EN: Enables or disables single ended outputs of USBIO. The bit should be set to "1"

USB_ON: This bit is used by the IMO DAC block to either work with better DNL consuming higher power, or with sacrificed DNL consuming lower power. The bit should be set to "1".

USB_CLK_ON: This bit either enables or disables the clocks to the USB block. The bit should be set to "1".

Version History

Version	Originator	Description
1.2	DHA	Added Version History.
1.30	DHA	Changed variable area location to allow code sublimation. Removed .Literal/.Endliteral directives around jmp instructions.
1.40	DHA	Added verification of the writing EP0_CR. Added verification of the SIE MODEs and ACK bit into the EP0 ISR. Updated the constant area definition in the user module.
1.50	DHA	1. Updated area declarations in boot.tpl to support Imagecraft optimization. 2. Added high level UART API description in this user module datasheet.
1.60	DHA	1. Fixed USBUART_bCheckUSBActivity API function to prevent missed activity. 2. Added CY8C20x66, CY7C643XX and CYRF89x35 device support.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.