Technical Note

Z-WORLD

AN213

Rabbit 2000 Serial Port Software

Introduction

Serial interfaces are among the oldest and most widely used methods for machine communication. The basic concept is to take the individual bits of a set of data and transmit them over a single digital channel one after the other. Many serial interfaces rely on clocking channels to coordinate the sending of each individual bit. These interfaces are referred to as synchronous interfaces. However, the majority of serial interfaces simply transmit bits at precise time intervals. The receiver can then sample the transmission line at these same intervals and retrieve the bits. These interfaces are known as an asynchronous serial interface. The most common asynchronous serial interfaces are RS-232 and RS-485. All of these interfaces use the same data format and differ only in their electrical specifications. This data format, known as SPI (serial peripheral interface) consists of a start bit (low, 0) followed by 7 to 9 data bits, and 1 or 2 stop bits (high, 1). When the communication line is idle, it is in the high (1) state. Therefore, the stop bits can be thought of as minimum idle time between transmitted bytes. In most cases only 7 or 8 of the data bits contain actual data. The extra bit, when present, is used as either a parity check bit or a frame signalling bit in a packet protocol. The Rabbit serial driver can be configured to use most of these options.

Two flavors of serial port support software are available for the Rabbit: circular-buffer routines and packet drivers. Circular-buffer routines for all four Rabbit serial ports are included in all Dynamic C releases. At the time of this writing, the packet driver routines are only implemented for Serial Port D and are only available as beta versions, but should be available soon for all ports in regular Dynamic C releases. The packet driver library and a sample program using are available in the zip file accompanying this Technical Note.

Circular Buffer Implementation

The circular-buffer serial routines are best used with RS-232. The interrupt times for the serial-driver interrupt service request are approximately 500 cycles for transmitting a byte and 400 cycles for receiving a byte. The serial driver uses circular buffers to temporarily hold data that are ready to be transmitted and data that have been received but not processed. The default size of these buffers is set to 31 bytes. These sizes can be changed using the macros **`XINBUFSIZE`** and **`XOUTBUFSIZE`**, where **`X`** refers to the serial port (A, B, C or D). Valid sizes for the buffers are $\mathbf{2^n - 1}$ (e.g., 15, 31, 63, 127).

Using Non-Cofunction Routines

The standard send and receive routines in the serial library will not return until finished, or when a timeout occurs in the case of receive routines. These functions rely on global data and are non-reentrant. Therefore, when using them with uC/OS-II or another preemptive multitasker, only one process at a time will be able to use a particular serial port. The serial port library is otherwise compatible with uC/OS-II. Here is a listing of the non-cofunction send and receive routines. Complete descriptions for them can be found in the ***Dynamic C Premier User's Manual***.

- **`serXgetc`**
- **`serXread`**
- **`serXpeek`**
- **`serXputc`**
- **`serXputs`**
- **`serXwrite`**

(In each of the above, **`X`** is one of A, B, C or D, corresponding to the desired serial port.)

Using Cofunction Routines

A full set a cofunction versions exists for the serial send and receive routines that will yield to other tasks while waiting for an operation to complete. The receive functions use timeouts to exit if no characters are received after a set amount of time. These functions are also considered non-reentrant with respect to preemptive multitaskers, and so only a single task is able to have access to a particular port. Here is a listing of the cofunction send and receive routines. Complete descriptions for them can be found in the ***Dynamic C Premier User's Manual***.

- **`cof_serXgetc`**
- **`cof_serXgets`**
- **`cof_serXread`**
- **`cof_serXputc`**
- **`cof_serXputs`**
- **`cof_serXwrite`**

(where **`X`** is A, B, C or D)

Parity and Stop Bits

The serial drivers can be configured to use any combination of:

- 7 or 8 bit data
- even, odd, or no parity
- 1 or 2 stop bits

The default format is 8 data bits, no parity, and one stop bit. This adds up to 9 bits. If a 10th bit is required, some special processing is done to transmit it. If the 9th bit is low, the 10th bit is handled in hardware by writing the byte to a special alternate port. If the 9th bit is high, a special delay scheme is used. The normal stop bit is used as the 9th high data bit, and the transmitter is disabled immediately after the byte is sent to create an idle state for one additional byte time. This creates a high 9th bit followed by a long stop bit. Unfortunately, this slows down the data throughput rate and can cause problems with hardware that is sensitive to gaps in the data stream. The serial packet library described later provides a solution to this for half-duplex communication.

Here is a listing of the mode configuration functions. Complete descriptions for them can be found in the ***Dynamic C Premier User's Manual***.

- **`serXparity`**
- **`serXdatabits`**

(where **`X`** is A, B, C or D)

Flow Control

There are often times when a system is unable to process incoming data at the rate it is being received. Buffers can handle short lapses in processing capability, but they will fill up if the receiver is consistently unable to keep up. Methods for a receiver to signal when it is able to receive data are know as flow control. The two main methods for flow control are XON/XOFF and a hardware-based method. The XON/XOFF method designates two byte values to be the XON and XOFF signals respectively. When the receiver is unable to process any more data, it transmits an XOFF control byte. When it is once again able to receive data, it sends an XON byte. Obviously, the data being sent must not include these control bytes. Hardware flow control relies on additional signal lines (RTS and CTS) between the two systems to indicate when data can be sent.

Flow Control Implementation

The Rabbit serial driver implements hardware flow control. The driver is configured as a DTE (data terminal equipment), meaning RTS is an output asserted by the Rabbit when it is ready for more data, and CTS is an input that monitors the ready state of the system the Rabbit is connected to. Flow control lines (RTS/CTS) are currently configured using **`#define`** macros to specify which port and bit a particular line will use. Here is an example of configuring RTS/CTS for Serial Port D.

```
#define SERD_RTS_PORT PDBR
#define SERD_RTS_SHADOW PBDRShadow
#define SERD_RTS_BIT 6
#define SERD_CTS_PORT PBDR
#define SERD_CTS_BIT 5
```

The two functions **`serXflowcontrolOn( )`** and **`serXflowcontrolOff( )`** are used to enable or disable hardware flow control.

Achievable Baud Rates

Appendix Appendix:, “Baud Rate Speed Tests,” presents the results of some speed tests with two different speed Rabbit boards under different conditions. The highest practical standard baud rates depend on factors such as the interrupt-latency effects of other interrupt-driven processes.

Packet Library Implementation

As of Dynamic C 7.03, the packet driver functions are available only for Serial Port D, and only as a beta version. The code is in the zip file accompanying this Technical Note. To use the packet driver, **`PACKET.LIB`** should be listed in the file **`LIB.DIR`**.

The Rabbit packet driver handles transmitting and receiving a variety of data packet formats over a half-duplex or a full-duplex communication channel. The driver can be easily adapted to use any type of transceiver hardware by writing custom routines to handle switching between transmit and receive modes. There are three basic types of packets that the library handles.

- Gap packets—packets that are separated by gaps in transmission of a set length
- 9th bit packets—packets that use the 9th bit to mark the first byte in a packet
- Start character packets—packets that use a special byte to mark the beginning of a packet

The gap packet and special character modes can also be configured to use the 9th bit for parity or as an extra stop bit. A benefit of the half-duplex restriction is that the driver interrupt service request is able to raise the baud rate temporarily when simulating a high 9th data bit. This allows for the stop bits to be closer to a true bit time rather than the length of a whole byte. Since the transmitter and receiver logic for a port both use the same baud rate counter, bytes cannot be received properly while this is occurring, therefore only half-duplex communication is allowed.

Spillover Buffer

The spillover buffer is a byte array used internally by the packet driver to store bytes when there is no user-supplied receive buffer available. Once a complete packet has been received, no further bytes will be written into the current receive buffer. The system will instead put them into the spillover buffer. When the next call to **`pktXreceive`** is made, the spillover buffer contents will be copied into the new receive buffer. The size of the spill buffer is set at compile time with the macro **`XPKTBUFSIZE`**, where X is replaced with the serial port designation (A,B,C, or D).

User-Defined Functions

Since the packet library is meant to be used with a variety of transceiver hardware, certain functions must be defined by the user: **`pktXinit()`**, **`pktXrx()`**, and **`pktXtx()`** (where **`X`** is A,B,C, or D). Each takes no arguments and returns nothing.

The sample code accompanying this Technical Note implements **`pktDinit()`**, **`pktDXrx()`**, and **`pktDtx()`**, and will work on a Jackrabbit board or a RabbitCore RCM2000 module.

- **`pktXinit()`**—Initializes the communication hardware. Called inside **`pktXopen()`**. This function may be written in C.
- **`pktXrx()`**—Sets the hardware to receive data. This must be written in assembly. Any registers besides the 8-bit accumulator A must be preserved first, and restored before returning.
- **`pktXtx()`**—Sets the hardware to transmit data. This must be written in assembly. The rules for register usage are the same as for **`pktXrx()`**.

Function Listing

For these functions, **`X`** is one of A, B, C, or D. As of Dynamic C 7.03, these functions are available only for Serial Port D, and only as a beta version. The code is in the zip file accompanying this Technical Note.

```
int pktXopen (long baud, int mode, char options,
   int (*test_packet)());
```

The open function sets up the isr vector and the baud rate. The minimum baud rate is PCLK / 32 / 256 / 256, and the maximum baud rate is PCLK / 32. Baud rates below PCLK / 32 / 256 (below 1800 if the crystal is 7.372 MHz with clock doubled for PCLK = 14.744 MHz) modify the A1 timer, affecting the B timer.

Parameters

`baud`—desired baud rate in bits per second

`mode`—type of packet scheme used. Valid options are:

`PKT_GAPMODE`
`PKT_9BITMODE`
`PKT_CHARMODE`

`options`—Further specification for the packet scheme. The value depends on the mode used:

gap mode - minimum gap size (in byte times)

9bit mode - type of 9-bit protocol

`PKT_RABBITSTARTBYTE`

The first byte in a packet has a low 9th bit, all other bytes are 8 bit length.

`PKT_LOWSTARTBYTE`

The first byte in a packet has a low 9th bit, all other bytes have a high 9th bit.

`PKT_HIGHSTARTBYTE`

The first byte in a packet has a high 9th bit, all other bytes have a low 9th bit.

char mode - character marking start of packet

`test_packet`—pointer to function that tests for completeness of a packet. The function should return 1 if the packet is complete, or 0 if more data should be read in. For gap mode the test function is not used and should be set to **`NULL`**.

Return Value

1—The baud rate set on the Rabbit 2000 is the same as the input baud rate.
0—The baud rate set on the Rabbit 2000 does not match the input baud rate.

`void pktXsetParity(char mode);`

Configures the driver to use the 9th bit as either a parity bit or a second stop bit. This can only be used with “gap” packets and “start character” packets.

Parameters

`mode`—the specific use of the 9th bit:

`PKT_NOPARITY`—don’t use the 9th bit for anything special (8N1 format)

`PKT_OPARITY`—Odd-parity bit (8O1 format)

`PKT_EPARITY`—Even-parity bit (8E1 format)

`PKT_TWOSTOP`—Use the ninth bit as an extra stop (8N2 format)

Return Value

None

`int pktXclose ();`

Disables the serial port interrupt service routine.

Parameters

None.

Return Value

1

`void pktXsend (void *send_buffer, int buffer_length, char delay);`

Sends a packet. If there is already a packet being transmitted, the function will block until that packet is done. Otherwise, the function will return immediately

Parameters

`send_buffer`—buffer containing the packet to be sent

`buffer_length`—the length of the send buffer

`delay`—the number of byte times to wait before sending the packet.

`void cof_pktXsend (void *send_buffer, int buffer_length, char delay);`

The cofunction version of **`pktXsend()`**. The function will return when the given packet is done transmitting.

Parameters

`send_buffer`—buffer containing the packet to be sent

`buffer_length`—the length of the send buffer

`delay`—the number of byte times to wait before sending the packet.

`void pktXreceive (void *buffer, int buffer_length);`

Prepares the driver to receive a packet. This function will always return immediately, however the buffer will be in use by the system until **`pktXreceiveDone()`** (see below) returns true.

Parameters

`buffer`—a byte array to store the incoming packet

`buffer_length`—the length of the buffer

`int cof_pktXreceive (void *buffer, int buffer_length);`

The cofunction version of **`pktXreceive()`**. The function will return when the given packet is done transmitting. Prepares the driver to receive a packet. This function will return once a complete packet has been read into the buffer.

Parameters

`buffer`—a byte array to store the incoming packet

`buffer_length`—the length of the buffer

Return Value

The number of bytes in the received packet

`int pktXsending ();`

Returns true is a packet is currently being transmitted

Parameters

None

Return Value

1—a packet is being transmitted
0—nothing is being transmitted

`int pktXreceiveDone ();`

Returns true if a complete packet is in the current receive buffer.

Parameters

None

Return Value

1—A complete packet has been received
0—The receive buffer is not ready

`char pktXgetErrors ();`

Returns a byte of error flags, with bits set for any errors that occurred since the last time this function was called. Any bits set will be automatically cleared when this function is called so that a particular error will only be reported once. The error flags are checked with error flag masks to determine which errors occurred.

Error flag masks:

`PKT_BUFFEROVERFLOW`—the received packet is too large for the given buffer

`PKT_RXOVERRUN`—the receive register could not be read before another byte was received

`PKT_PARITYERROR`—a byte was received with incorrect parity

`PKT_SPILLOVERFLOW`—the spillover buffer filled up before another receive buffer became available

Parameters

None

Return Value

The error flags byte.

Appendix: Baud Rate Speed Tests

Four different tests using the RS-232 circular buffer functions were run on both 14 MHz and 29 MHz Jackrabbit boards, with four variations on each test. The first test was to transmit blocks of data continuously and without any gaps in the outgoing data stream. The baud rates listed in the tables below are the maximum rates at which this is possible.

The second test was like the first, except that characters were transmitted one at a time by the test program using **`serXputc()`**. The baud rates listed are the highest speeds that do not produce gaps between bytes.

The third test received a continuous stream of data in blocks. The baud rate in the table is the highest at which overruns of the hardware receive buffer did not occur.

The fourth test was identical to the third except that data was received a character at a time.

Each test was performed under four different sets of conditions. In each, hardware flow control (FC) was either on or off, and the test was performed in either debug mode or no debug mode (run mode).

Table 1. Baud Rate Limits for Serial Tests (14 MHz Rabbit 2000)

	no FC, no debug	**no FC, debug**	**with FC, no debug**	**with FC, debug**
Block Transmit Test	230,400 bps	230,400 bps	115,200 bps	115,200 bps
Character Transmit Test	115,200 bps	57,600 bps	115,200 bps	57,600 bps
Block Receive Test	230,400 bps	57,600 bps	230,400 bps	57,600 bps
Character Receive Test	230,400 bps	57,600 bps	230,400 bps	57,600 bps

Table 2. Baud Rate Limits for Serial Tests (29 MHz Rabbit 2000)

	no FC, no debug	**no FC, debug**	**with FC, no debug**	**with FC, debug**
Block Transmit Test	460,800 bps	230,400 bps	230,400 bps	230,400 bps
Character Transmit Test	230,400 bps	115,200 bps	115,200 bps	115,200 bps
Block Receive Test	460,800 bps	57,600 bps	460,800 bps	57,600 bps
Character Receive Test	460,800 bps	57,600 bps	460,800 bps	57,600 bps

Z·WORLD

Z-World
2900 Spafford Street
Davis, California 95616-6800
USA

Tel. (530)757-3737
FAX (530)753-5141

Web site: http://www.zworld.com