TECHNICAL NOTE

TN213

Rabbit Serial Port Software

Rabbit Semiconductor supplies both stream- and frame-based drivers for the four serial ports of the Rabbit 2000 microprocessor. This technical note will discuss both types of drivers.

You must have a Rabbit 2000 or newer Rabbit chip to use the serial port driver libraries. Most Dynamic C libraries are forward compatible; to verify this for a particular library read the comments at the top of the library file.

Newer Rabbit processors have additional serial ports with additional features. Please see the appropriate chip manual for details.

Overview of Serial Communication

Serial interfaces, among the oldest and most widely used methods for machine communication, send and receive individual bits over a single digital line. Full-duplex communication (sending and receiving at the same time) is possible with only three wires: TX (transmit), RX (receive) and GND (ground).

Serial communication is either synchronous or asynchronous. Synchronous communication requires an additional wire for a clocking signal to coordinate the transmitting and receiving of each bit. Asynchronous communication does not need the additional wire, but requires that each byte that is transmitted be identified with stop and start bits. The sender transmits bits at precise time intervals and the receiver may then sample the transmission line at these same intervals to retrieve the bits.

The most common asynchronous serial interfaces are RS-232, RS-485, and RS-422. The serial ports on IBM PC-type computers are asynchronous RS232. Asynchronous interfaces use the same data format and differ only in electrical specifications. This data format consists of a start bit, followed by 7 or 8 data bits, followed by an optional bit, followed by one or two stop bits. The optional bit is the 8th bit when there are 7 data bits and it is the 9th bit when there are 8 data bits. This extra bit may be used as either a parity check bit or a frame-based driver may use it as a frame-signalling bit in a packet protocol. Our frame-based driver only supports 8 data bits.

The start bit is a logical zero and the stop bit is a logical one. A logical zero may be noted by: ***0***, ***low*** or ***space***. A logical one may be noted by: ***1***, ***high*** or ***mark***. An idle communication line is in the high state (a logical one). Since stop bits are also a logical one, they can be thought of as minimum idle time between transmitted bytes.

Asynchronous Serial Drivers

All four serial ports of the Rabbit 2000 microprocessor may be used for asynchronous communication; ports A and B may also be used for synchronous communication. The serial drivers that are provided with Dynamic C and described here are meant for asynchronous communication and would need modification to be used in the synchronous mode of either port A or B. (Synchronous drivers for an SPI interface are available starting with Dynamic C version 7.05.)

Data Transfer and Interrupt Times

The highest practical standard baud rate usable by the serial drivers is 115,200 bps. The Rabbit 2000 allows transmitting at much higher baud rates, but significant gaps in data begin to appear, limiting the actual transfer rate. Interrupt times for the serial driver interrupt service request (ISR) are approximately 500 cycles for transmitting a byte and 400 cycles for receiving a byte.

The Stream-Based Driver

The stream-based serial driver is implemented by the Dynamic C library `RS232.LIB`. This library consists of circular buffers, an interrupt service routine and user interface functions.

There are two circular buffers for each of the four serial ports, one for reading (the receive buffer) and one for writing (the transmit buffer). These buffers temporarily hold data that is ready to transmit and data that has been received but not processed. The default size of these buffers is set to 31 bytes. The sizes can be changed using the macros `XINBUFSIZE` and `XOUTBUFSIZE`, where `X` refers to serial port A, B, C, or D. Valid sizes for the buffers are a power of 2 minus 1 (e.g., 15, 31, 63, 127).

A buffer size of 2^n-1 enables masking for fast roll-over calculations.

Stream-Based Driver Transmit and Receive Routines

The standard transmit and receive routines in `RS232.LIB` are listed here. `X` is A, B, C or D, and designates the serial port. Complete function descriptions are in the *Dynamic C Function Reference Manual*.

- `serXgetc()` reads the next character in the receive buffer.
- `serXread()` reads a specified number of bytes in the receive buffer.
- `serXpeek()` looks at the next character in the receive buffer.
- `serXputc()` writes a character to the transmit buffer.
- `serXputs()` writes a null-terminated string to the transmit buffer.
- `serXwrite()` writes a specified number of bytes to the transmit buffer.

Except for `serXpeek()`, they all first lock the buffer they use and unlock it before returning. These functions rely on global data, making them non-reentrant. When using them with µC/OS-II, or another preemptive multitasker, only one process at a time will be able to use a particular serial port. The stream-based driver (a.k.a., the RS232 driver) is otherwise compatible with µC/OS-II.

The functions `serXputs()` and `serXwrite()` block. They do not return until the last of the data they are to write has been successfully placed into the circular buffer, at which point the functions return while the data transmission proceeds using the ISR.

> **TIP:** The data transmission is complete when the statement `(!serXwrUsed() && !BitRdPortI(SxSR, 2))` is TRUE. This means that the output buffer is empty and the status register reports idle. Since there are cases where the tranmitter is off for short periods of time even when there is more data in the buffer, it is safest to test for both conditions.

Stream-Based Driver Cofunction Routines

Cofunction versions of the standard send and receive routines are provided by RS232.LIB. The receive functions use time-outs to exit if no characters are received. These functions are also considered non-reentrant with respect to preemptive multitaskers, so only a single task has access to a particular port at any one time.

The cofunction routines yield to other tasks while waiting for an operation to complete, but do not return to execute the next statement within their own costatement block until they have completed their operation. The transmitting functions yield to other tasks whenever the output buffer becomes full, while transmission takes place using the ISR.

Here is a list of the cofunction send and receive routines. X is A, B, C or D, and designates the serial port. Complete function descriptions are in the *Dynamic C Function Reference Manual.*

- cof_serXgetc() yields to other tasks until a character is read from the receive buffer.
- cof_serXgets() reads characters from the receive buffer until one of several conditions is met. Yields to other tasks if the buffer is locked or empty.
- cof_serXread() reads the specified number of characters from the receive buffer unless a time-out occurs between characters. Yields to other tasks if the buffer is locked or empty.
- cof_serXputc() writes a character to the transmit buffer. Yields to other tasks if the buffer is locked or full.
- cof_serXputs() writes a null-terminated string to the transmit buffer. Yields to other tasks if the buffer is locked or full.
- cof_serXwrite() writes the specified number of bytes to the transmit buffer. Yields to other tasks if the buffer is locked or full.

Other Stream-Based Driver Routines

The following functions make up the rest of the RS232 driver API. X is A, B, C or D, and designates the serial port. Complete function descriptions are in the *Dynamic C Function Reference Manual.*

- serCheckParity() tests 8-bit character for correct parity.
- serXclose() disable serial port.
- serXdatabits() configures serial port to use 7 or 8 data bits.
- serXgetError() get byte of error flags set since last call to the function.
- serXopen() enable serial port.
- serXparity() set parity mode.
- serXrdFlush() flushes receive buffer.
- serXrdFree() returns the number of bytes unused in the receive buffer.
- serXrdUsed() returns the number of bytes currently in use in the receive buffer.
- serXWrFlush() flushes transmit buffer.
- serXWrFree() returns the number of bytes available for use in the transmit buffer.

Using Alternate Pins

All four serial ports share pins with parallel port C. Serial ports A and B can be configured to use the alternate pins of parallel port D instead. For those designing a board with the Rabbit 2000 microprocessor or a Z-World Core Module, to use the alternate pins of parallel port D for serial port A or B, include the appropriate macro definition in your application:

```
#define SERA_USEPORTD
```

or

```
#define SERB_USEPORTD
```

This is not necessary for the RCM2200, the RCM2300 or the RCM2250. The driver will define `SERB_USEPORTD` when any of those board types are recognized.

If you have a Z-World controller board the choice to use the alternate pins of parallel port C or D for serial port B has already been made by the designers of the board. Consult the hardware user's manual for your Z-World board for more information.

Using Flow Control with the RS232 Driver

Sometimes a system can not process incoming data at the rate it is being transmitted. Buffers may be used in these situations, but they will overflow if the receiver is unable to keep up with the transmitter. Flow control solves the problem by allowing the receiver to signal when the transmitter should pause.

Flow control may be implemented in software or hardware. The RS232 driver uses hardware flow control. The two functions `serXflowcontrolOn()` and `serXflowcontrolOff()` are used to enable or disable hardware flow control. `X` is A, B, C or D, and designates the serial port.

The Rabbit is configured as a DTE (Data Terminal Equipment), meaning that the flow control line RTS (Request To Send) is an output asserted by the Rabbit when it is ready for more data, and CTS (Clear To Send) is an input that monitors the ready state of the system that is connected to the Rabbit. RTS and 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 B.

```
#define SERB_RTS_PORT          PBDR         // use port B data register
#define SERB_RTS_SHADOW        PBDRShadow   // use port B shadow register
#define SERB_RTS_BIT           6            // output
#define SERB_CTS_PORT          PBDR         // input
#define SERB_CTS_BIT           5
```

The Frame-Based Driver

The frame-based driver is implemented by the Dynamic C library PACKET.LIB. Unlike the RS232 driver with its point-to-point interface, the packet driver was designed for the multipoint communications of an RS485 interface.

The packet driver handles transmitting and receiving data packet formats in half-duplex mode. The supported packet formats are:

- **Gap**—packets that are separated in transmission by gaps of a set length. No such gaps exist within the packet. Modbus uses this technique for delimiting frames; an end of frame is recognized if a gap of more than 3.5 character times exists between characters.
- **9th Bit**—packets that use the 9th bit to mark the first byte in a packet. This is used in the Opto 22 protocol and others. The Rabbit 2000 supports transmitting and receiving a low 9th bit. Transmitting and receiving a high 9th bit is simulated by the packet driver. Using the 9th bit to mark the first byte in a packet precludes the use of a parity bit or an extra stop bit.
- **Start Character**—packets that use a special byte to mark the beginning of a packet. This is used in protocols that use ASCII rather than binary data.

The gap packet and start character modes can be configured to use the 9th data bit for parity or as an extra stop bit. This is done by a call to pktXsetParity().

Recognizing Solitary Packets

With 9th bit and start character packet formats, the driver assumes a packet is completely received when it recognizes another 'start packet' signal. This is a problem if a solitary packet is received. The solution to this is a user-defined function that can determine whether or not a packet is complete. The packet driver will call this user-defined function when pktXreceive() is called and the only packet available is the one currently being received. The prototype for the function is:

```
int test_packet(char *packet_bytes, int count);
```

packet_bytes	The current packet contents to test.
count	The number of bytes in the current packet.

The function should return 1 if the packet is complete. A pointer to test_packet() is passed to pktXopen() when using 9th bit or start character packet formats. A null pointer is passed to pktXopen() when using the gap packet format.

Packet Driver API

Full descriptions for these functions are in the *Dynamic C Function Reference Manual*. The "X" in the function name designates the serial port: A, B, C, D, E or F.

Open, Close, and Configuration Functions

The open and close functions enable and disable serial communication over the specified port.

- `pktXinitBuffers()` allocates a block of extended memory for the packet driver.
- `pktXopen()` opens serial port, identifies baud rate, packet scheme and the user-supplied function that checks for packet completeness.
- `pktXsetParity()` configures 9th bit usage for gap and start character packet modes.
- `pktXclose()` disables the serial port ISR.

Send and Receive Functions

- `pktXsend()` initiates sending a packet.
- `cof_pktXsend()` cofunction version of `pktXsend()`.
- `pktXreceive()` gets a received packet if there is one.
- `cof_pktXreceive()` the cofunction version of `pktXreceive()`.

Status and Error Checking Functions

- `pktXsending()` returns true if a packet is currently being transmitted.
- `pktXgetErrors()` returns a byte of error flags.

User-Defined Functions

The packet driver is meant to be used with a variety of transceiver hardware, so some functions must be defined by the user. Each of these functions, listed below, take no arguments and return nothing.

- `pktXinit()` - Initializes the communication hardware. Called inside `pktXopen()`. This function may be written in C. It will only be called once each time the packet driver is opened, so speed is not a major concern. This is where I/O pins should be configured and any other setup should be performed.
- `pktXrx()` - Sets the hardware to receive data. This function must be written in assembly. Any registers besides the 8-bit accumulator A must be preserved first, and restored before returning. This function is called when the driver switches from transmit to receive mode once there are no packets to send. This function is necessary for half-duplex connections and other types of shared bus schemes so that the transmitter can be disabled, allowing other nodes to use the lines.
- `pktXtx()` - Sets the hardware to transmit data. This function must be written in assembly. The same rules for register usage as for `pktXrx()` apply. This function is called whenever the driver switches from receive to transmit mode in response to an additional packet or packets being available for sending. A typical use of this function is to enable any necessary transmitter hardware.

See the sample program `Samples/PKTDEMO.C` for an example of how to write these user-supplied functions.

Parity and Stop Bits

The RS232 driver and the packet driver can be configured to use most combinations of:

- 7 or 8 data bits (the packet driver only supports 8 data bits)
- even, odd, or no parity
- 1 or 2 stop bits

The only limitation for the RS232 driver is that parity bits and extra stop bits (more than one) cannot be combined due to limitations in the UART hardware; 11 bits is the upper limit. The packet driver shares this limitation and, in addition, disallows the 9th bit packet format to be used with parity or a second stop bit.

The default format is 1 start bit, 8 data bits, no parity, and one stop bit (8-N-1). This adds up to 10 bits.

Transmitting a 9th Data Bit

Special processing is required to transmit 9 data bits. A low 9th bit is handled in hardware by writing the byte to a special alternate port. For a high 9th bit, a special delay scheme is used by the serial drivers. For the stream-based driver, the normal stop bit is used as the high 9th 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. When using the RS232 driver, this delay scheme slows down the data throughput rate and can cause problems with hardware that is sensitive to gaps in the data stream.

The packet driver resolves the issue of an additional byte time delay by speeding up the baud rate. The result is an idle state of a few bit times instead of a full byte time. Because the transmitter and receiver logic for a serial port both use the same baud rate counter this scheme can only be used in half-duplex mode. The packet driver ISR is able to raise the baud rate temporarily with the knowledge that nothing will be received during that period.

Here are the mode configuration functions. Complete descriptions for them can be found in the *Dynamic C Function Reference Manual*.

- `serXparity()` establishes parity check or extra stop bit in RS232 driver.
- `serXdatabits()` establishes number of data bits in RS232 driver.
- `pktXsetParity()` establishes parity check or extra stop bit in packet driver.

Summary

The serial port drivers for the Rabbit microprocessors are provided in source code format with Dynamic C. The stream-based driver is suitable for full-duplex point-to-point communication. The frame-based driver is suitable in a half-duplex multidrop system and has been designed to be easily adaptable for many types of transceiver hardware.