PUMPKIN™
REAL-TIME SOFTWARE

AN-8
Application Note

750 Naples Street • San Francisco, CA 94112 • (415) 584-6360 • http://www.pumpkininc.com

Implementing a Quad 1200 baud Full-duplex Software UART with Salvo

Introduction

A Software-driven Universal Asynchronous Receiver Transmitter (UART)[1] is a low-cost solution to the problem of not enough hardware UARTs in a microcontroller. This Application Note illustrates how to implement up to four 1200 baud full-duplex UARTs in software by using the Salvo™ RTOS on various Microchip® PICmicro® devices, including the PIC16F877.

The only dedicated resources required are general-purpose I/O pins for the receivers and transmitters, ROM and RAM, a single periodic interrupt at four times the baud rate and a free-running timer to count instruction cycles. The application is written entirely in C.

The software is configurable for one to four software receivers and one to four software transmitters. The processor clock speed and baud rate can also be specified. It can be used with all versions of Salvo, including the freeware version, Salvo Lite™.[2] It can easily be incorporated into a larger multitasking application, or users can add conventional "superloop" code to it.

A detailed analysis of the system behavior is presented, using Logic Analyzer screen dumps to illustrate the run-time behavior of the system's tasks, interrupt handler, main loop and serial I/O.

Simple Serial Communications

Figure 1 illustrates the serial bitstream for the letter 't' (ASCII 116, 0x74, 0b01110100) as it would appear on an I/O pin.

created by Andrew E. Kalman on Aug 8, 2001 updated on Aug 20, 2001

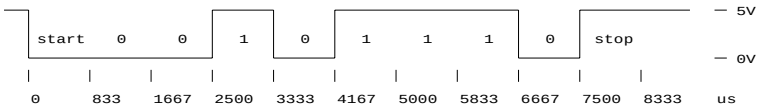

Figure 1: 1200 baud Serial Bitstream on Microcontroller I/O Pin

This is the sort of signal you would see between a microcontroller's hardware UART output pin and an RS-232 transmitter, for instance. Note the following:

- when *idle*, the bitstream is logic high (+5V),[3]
- the start of transmission is marked by a transition from high to low (the *start bit*),
- there are eight *data bits*, transmitted least-significant bit (LSB) first and
- the end of transmission is marked by a transition from low to high (the *stop bit*)

At 1200 baud, bit transitions may occur every 833µs, and one byte of data can be transmitted and received every 8.333ms, or ten *bit periods*. *Baud rate errors* of less than 5% are generally acceptable at low baud rates. *Full-duplex* means that serial data can be received and transmitted at the same time. In this case, separate signals are used for receive and transmit bitstreams.

Note This Application Note's software UARTs are set to 1200 baud, with no parity, eight data bits, and one stop bit. These settings – commonly referred to as *1200,N,8,1* – can be modified to suit other requirements by editing the source code.

Resources Required

Each full-duplex software UART, consisting of a software receiver and a software transmitter, will require two I/O pins – one for the receiver and one for the transmitter. The code to implement them will require ROM, as well as some RAM. Lastly, some form of timing and/or periodic timing signal will be required. This is often accomplished via a *periodic interrupt* driven by a hardware timer.

Design Challenges

Salvo is a priority-based event-driven cooperative multitasking RTOS. Tasks may be delayed for a specified number of *system ticks*, but *time-slicing* is not explicitly supported. Thus it might seem difficult to implement the timing-sensitive receiver and transmitter algorithms in software. However, Salvo's *tasks*, *task*

priorities, *events* and means of *intertask communications* can be combined with a multi-purpose *interrupt service routine* (*ISR*) to implement the desired functionality.

While effort was made to minimize ROM and RAM usage, the greatest challenge – not surprisingly – was to guarantee proper serial bitstream timing.

Implementation

Theory of Operation

Serial Reception in Software

Implementing a serial receiver in software requires that the start bit be detected as quickly as possible, and that each data bit be sampled thereafter in the middle of its bit period.[4] Finally, a valid stop bit must also be detected. Listing 1 contains pseudocode for a simple software receiver.

```
while ( no start bit detected )
  do nothing;
if ( start bit valid )
  wait one and a half bit periods;
  data bit #1 (LSB) = receiver pin;
  wait one bit period;
  data bit #2 = receiver pin;
  wait one bit period;
  …
  data bit #8 (MSB) = receiver pin;
  wait one bit period;
  stop bit = receiver pin;
if ( not stop bit valid )
  discard data;
wait half bit period;
```

Listing 1: Pseudocode for Software Receiver

To implement this software receiver successfully, the application must have sufficiently fast response time to detect the start bit soon after it occurs and begin sampling incoming data, and it must have accurate internal timing so that each bit is sampled in the middle of its bit period, so that false readings are not taken during the signal's logic level transitions.

Serial Transmission in Software

Implementing a serial transmitter in software is relatively straightforward. Each bit of the data being sent must be shifted out the transmitter pin serially, preceded by a start bit and followed by a stop bit. An accurate shift clock is required and synchronization must be maintained so that the serial bitstream represents valid data to a connected receiver. Listing 2 contains pseudocode for a software transmitter, where bits and pins each have values of either 0 (logic low, GND) or 1 (logic high, often +5V).

```
transmitter pin = start bit;
wait one bit period;
transmitter pin = data bit #1 (LSB);
wait one bit period;
transmitter pin = data bit #2;
wait one bit period;
…
transmitter pin = data bit #8 (MSB);
wait one bit period;
transmitter pin = stop bit;
wait one bit period;
```

Listing 2: Pseudocode for Software Transmitter

The final wait of one bit period is required to ensure that the stop bit is valid for at least one bit period before the next transmission occurs.

Buffers and Buffer Control

In order for this implementation to be as general-purpose as possible, each software receiver and transmitter has a dedicated *data buffer* (`buff[]`). Each buffer is implemented as a *ring buffer* holding byte-sized data. Functions are provided to *put* and *get* data from each buffer. Each buffer also has a dedicated *buffer control block* (`bcb`), which is used by these functions to access and manipulate the buffer. Each buffer's *size* can be specified independently. By using this standardized buffer architecture, just two buffer routines[5] service all eight buffers, thus conserving ROM.

Upon receipt of valid data, a software receiver places that data into its buffer via a call to `PutRxNBuff()`.[6] The data can then be extracted by calling `GetRxNBuff()`. To send data via a software transmitter, the data is placed into its buffer via `PutTxNBuff()`. It is extracted from the buffer via `GetTxNBuff()` when the software transmitter is active.

Note The software transmitters and their output pins are labeled TX1 through TX4. The software receivers and their input pins are labeled RX1 through RX4. The hardware UART (used in testing) and its output pins are labeled TX5/RX5. Please see *Source Code Listings* below for more information.

Software Receivers

In order to simplify the application, it was decided to run the software receivers as tasks, with an interrupt-driven *start-bit detector* for each one. The start bit detector is a simple code snippet that watches for the receiver pin to go logic low, and resamples it again to be sure. At this point, one could signal a binary semaphore upon which a receiver task was waiting. But to conserve RAM (each semaphore requires three bytes in this application), a different approach was taken. Each software receiver task is started by its start-bit-detector via a call to the RTOS service `OSStartTask()`, and once running, samples the incoming bitstream at the appropriate times.

In addition to starting the software receiver task, each start-bit-detector also records a *timestamp* when detection occurs. The timestamp is in units of instruction cycles,[7] and will be used by the receiver task to maintain synchronization while sampling the bitstream.

Lastly, once the start bit is detected, the start-bit detector must disable itself until the receiver task is done sampling the bitstream and placing valid data into its buffer.

Since maximum responsiveness from the software receiver tasks is desirable, in order for each task to run entirely independent of the others (since incoming data will not be synchronized across the four receivers), each software receiver requires its own start-bit detector and receiver task.

Receiver Task Operation

When multitasking begins, each software receiver task initializes its buffer and then stops via the RTOS service `OS_Stop()`. It will remain in the stopped state until its start-bit detector starts it up again.

Upon restarting, `TaskRxN()` should find itself running in the first or even second bit periods (i.e. start bit or first data bit) of the serial reception. It then computes (via `BitDelay()`) the number of

system ticks it should delay itself by calling the RTOS service `OS_Delay()` so that when its delay expires in the future, it will find itself in the middle of the bit period, and can sample the data bit. By setting the *system tick period* to be a quarter-bit period, software receiver tasks can delay themselves with a resolution of one quarter-bit period. To achieve this, the RTOS service `OSTimer()` is called every quarter-bit period via a periodic interrupt trigger by timer TMR2.

Note The delay portion of the algorithm in Listing 1 will not work in this application, because of accumulated error from one bit sampling to the next. When a task is delayed, it will only be made *eligible* to run when the delay expires – it will not necessarily run immediately thereafter. That's because there may be other, higher-priority tasks eligible to run, and so the task in question will be delayed by an additional number of processor cycles.

One solution to this problem is to delay each software receiver task by a computed number of system ticks after the start bit was detected. In other words, each task is re-synchronized with the start bit when it calculates what its next delay should be. Accumulated delay error is thus avoided.

This is repeated until all of the data bits and the stop bit are sampled. If any error occurs (e.g. the stop bit was not logic high, or an appropriate delay could not be implemented), the data is discarded and an error is flagged. Otherwise the data is inserted into the software receiver's buffer.

Finally, the software receiver's start-bit detector in `ISR()` is re-enabled.

Note The astute reader will recognize that since there is no disabling and re-enabling of interrupts around this final operation of the software receiver task, it is possible for the receiver's start-bit detector to be triggered – and hence `OSStartTask()` to be called – while the task is still running. However, this does not cause problems because `OSStartTask()` can only start a task that is stopped, which will only be the case after the software receiver task executes `OS_Stop()`.

Software Transmitters

Interrupt-driven Core for Accurate Timing

In order to avoid transmission errors[8], it is imperative that the shift clock for each software transmitter have minimal error. While it is possible to implement one or more software transmitters as independent tasks, it was found that timing errors were unacceptably high. Therefore, a *state machine* with different states representing the different stages in transmitting data, as shown in Table 1, is used to serially shift out the start, data and stop bits.

`TXSTATE_IDLE`	Transmitter is idle
`TXSTATE_START`	Transmit start bit(s)
`TXSTATE_DATA0-DATA7`	Transmit data bit(s)
`TXSTATE_STOP`	Transmit stop bit(s)
`TXSTATE_DONE`	Wait for stop bit to finish

Table 1: Transmitter State Machine States

The software transmitter's state machine is one of several independent parts of the interrupt service routine `ISR()`, which occurs via a periodic timer interrupt every *quarter-bit* period, i.e. every 208µs. Once transmitting, the state machine transitions from one state to the next every bit period (833µs).

Since there is no benefit from operating each software transmitter asynchronously, all four operate together, i.e. bit transitions for all four software transmitters occur at the same time. Only active software transmitters – i.e. those that currently have non-empty buffers – send data while the state machine is not idle.

Transmit Task Empties Buffers, Improves Performance

Running a software transmitter via a state machine from inside an interrupt handler is commonplace. In order to minimize the size of `ISR()`, it was decided to remove data from each transmit buffer and ready it for transmission via a task. This is accomplished via `TaskTx()`.

`TaskTx()` initializes the software transmitter buffers, and then waits via the RTOS service `OS_WaitBinSem()` for the binary semaphore `BINSEM_TXBUFF` to be signaled. When this occurs elsewhere in the application, it means that data in one or more software transmitter buffers is ready for transmission. Then, for each software transmitter, `TaskTx()` strips a data byte from a non-

empty buffer, places it in a globally accessible variable (`txNData`) and makes the corresponding transmitter (see `ISR()`) active. It then starts the transmitter state machine (see *Transmit Task and ISR Interaction*, below). When transmission is complete, it signals `BINSEM_TXBUFF` if any of the software transmitter buffers remain non-empty.

Since independent software transmitter tasks would consume more RAM as well as ROM, and offer no other benefit given the construction of the state machine in `ISR()`, only a single software transmitter task is required.

Transmit Task Handles Software Transmitters

When the application wants to send data via a particular software transmitter, it simply places the data in the corresponding buffer and signals a binary semaphore that indicates that data is ready for transmission. This is done via calls to `PutTxNBuff()` and the RTOS service `OSSignalBinSem(BINSEM_TXBUFF_P)`. This binary semaphore represents the flow of information from other parts of the program to `TaskTx()`.

Transmit Task and ISR Interaction

With the software transmitter state machine idling in `ISR()`, and `TaskTx()` having readied data for transmission, some means of communicating between `TaskTx()` and `ISR()` are required. Again, binary semaphores are used – one to indicate that the state machine should begin transmitting `(BINSEM_TXSTART)`, and one to indicate when it has finished `(BINSEM_TXSTOP)`. The former represents the flow of information from `TaskTx()` to `ISR()`, the latter from `ISR()` back to `TaskTx()`.

Note Of special interest is the use of the binary semaphore `BINSEM_TXSTART`. ISRs cannot wait events in the conventional sense – i.e. `OS_WaitBinSem()` cannot be called from within `ISR()`. However, a related function, `OSTryBinSem()` can be called from the foreground (i.e. interrupt level). `ISR()` uses this RTOS service to detect when transmission should begin.

Choosing Task Priorities

Salvo's tasks are assigned priorities. In this application, the timing-sensitive receiver tasks run in the background. Therefore they should be given the highest task priorities to ensure that they get

processor cycles when they need them. The transmitter task is not time-critical, and so it can run at a lower priority. Additional tasks should run at even lower priorities.

Timing Issues

Two issues dictate the minimum interrupt rate – the rate at which the start-bit detectors must be called, and the minimum system tick rate for adequate delay resolution (via `OS_Delay()`) for use in the receiver tasks. An interrupt rate of four times the baud rate – giving minimum delays of 208µs –works well.

Test Code

To verify the quad UART's proper operation, test code was written to take 8-bit data received via the hardware UART (RX5), permute it four different ways, and pass the permuted data on to the four software transmitters TX1-TX4. Each receiver RX1-RX4, which is connected externally to its respective transmitter (see Figure 2, below) reverses the permutation and passes the result to the hardware UART (TX5). The permutations are listed in Table 2.

`TX1`	none
`TX2`	data is complemented
`TX3`	data's nibbles are swapped
`TX4`	data's nibbles are swapped and complemented

Table 2: Test Code Permutations on Incoming Data

Any errors in the software transmitters or receivers will be reflected in the output of TX5 not matching the input of RX5. A simple terminal program is used to send data to RX5 and receive it from RX5. The terminal program's serial output comes from both individual keystrokes and text file dumps. The test code was implemented as an additional task `TaskTestCode()` with low priority.

Hardware UARTs

The test code requires the use of a hardware UART at 9600 baud (N,8,1) to send data to and receive data from the system under test. The target processor has its own hardware UART, which supports interrupt-driven or polled operation. It was found that the application could not tolerate these additional two[9] sources of interrupts, due to the frequency at which `ISR()` is called coupled with its context-saving and restoring times.

Therefore polled operation for the hardware UART was selected. This has no discernable deleterious effect on the system, as the receiver is at least double-buffered, and its receiver full register (`RCIF`) is polled five times faster than 8-bit data is received. Similarly, the transmitter also benefits from the rapid rate at which its transmitter empty register (`TRMT`) is polled.

Each hardware receiver and transmitter has its own buffer (`rx5Buff[]` and `tx5Buff[]`). Discrete buffer control variables are used (as opposed to the software receivers' and transmitters' buffer control blocks) for added speed and slightly different functional requirements.

Test Setup

The test setup consists of a Microchip PIC16F877 PICmicro® MCU with a few external components, running at 20MHz (200ns instruction cycle). A schematic diagram is presented in Figure 2. A PC running a terminal program at 9600,N,8,1 is connected via a null-modem cable to connector H1.

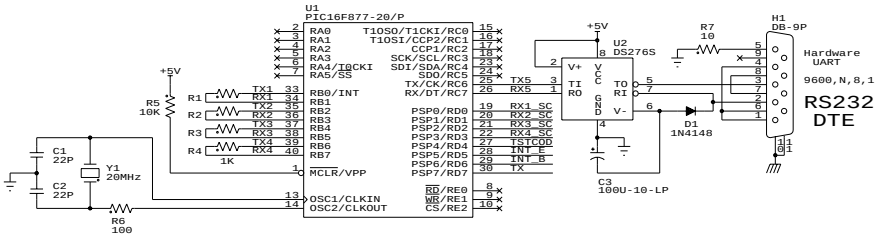

Figure 2: Test Setup Schematic Diagram

A functionally identical test setup can be derived from the Microchip PICDEM-2 Demonstration board by placing a quad-resistor network [10] in U2's pins 21-28.[11]

Nomenclature

The signals in the Test Setup and displayed in the logic analyzer screens (below) are described in Table 3.

RX5	data received from PC terminal program at 9600 baud
TX5	data transmitted to PC terminal program at 9600 baud
INT_B	beginning of ISR() (after context save)
INT_E	end of ISR() (before context restore)
TX	toggles when TaskTx() runs, tx1Data1-tx4Data are ready for transmission
TX/RX1	RX1 input pin (connected to TX1), 1200 baud
TX/RX2	RX2 input pin (connected to TX2), 1200 baud
TX/RX3	RX3 input pin (connected to TX3), 1200 baud
TX/RX4	RX4 input pin (connected to TX4), 1200 baud
TSTCOD	set high at the beginning of TaskTestCode() and reset low at the end
RX1_SC	toggles when TaskRx1() samples RX1
RX2_SC	toggles when TaskRx2() samples RX2
RX3_SC	toggles when TaskRx3() samples RX3
RX4_SC	toggles when TaskRx4() samples RX4

Table 3: Signal Names for Analysis

Performance

The minimum clock speeds[12] for a PIC16F877 to echo 9600 baud data received on RX5 back to TX5 without error are shown in Table 4. These results were obtained with the test code running as a task.[13]

1 software receiver/transmitter pair	10MHz
2 software receiver/transmitter pairs	12.5MHz
3 software receiver/transmitter pairs	16MHz
4 software receiver/transmitter pairs	20MHz

Table 4: Speed Requirements for UART Configurations

The non-uniform rise in minimum clock speed as the number of software receivers and transmitters increases is not surprising. Each additional receiver/transmitter pair requires one additional task. As the number of tasks increases, more cycles are expended by the RTOS in managing the priority-based scheduling of tasks.

Analysis

A logic analyzer[14] was connected to RX1, RX2, RX3, RX4, RX5, TX5 and PORTD[0..7] of the test setup. Waveform capture was triggered on TX5's start bit.

Single Byte Received

Figure 3 shows a single byte (ASCII '5', 0x35, 0b00110101) received via RX5.[15]

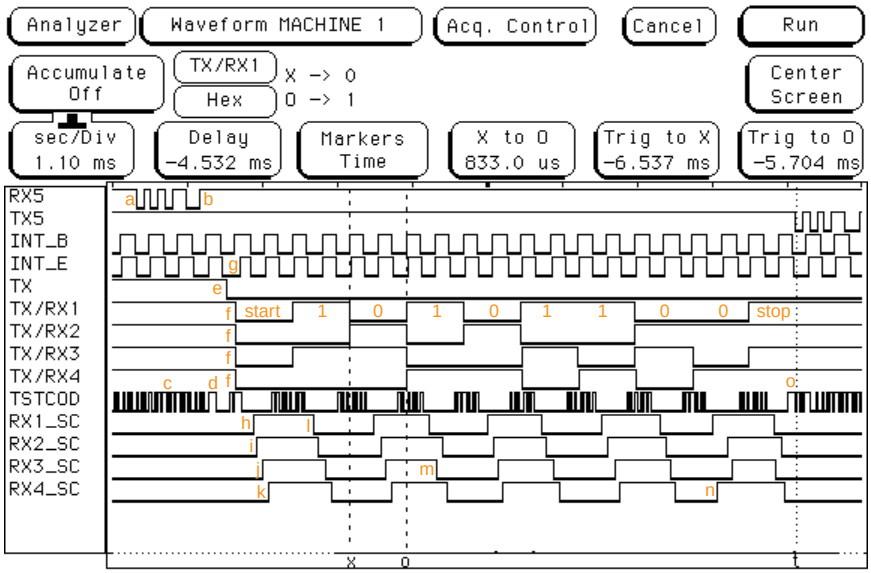

Figure 3: Single Byte Reception and Re-Transmission

Description of Events

The terminal program begins transmitting the start bit at 9600 baud at *a*. After the stop bit is received, the byte is placed into RX5's buffer at *b* in `ISR()`. During this time (*c*), `TaskTestCode()` has been polling RX5's buffer for an incoming byte. In `TaskTestCode()`, the '5' is stripped from RX5's buffer and put into TX1-TX4's buffers at *d*, and binary semaphore `BINSEM_TXBUFF` is signaled to indicate that data is waiting to be transmitted. This launches the first significant RTOS activity outside of `OSTimer()`, and changes `TaskTx()` from waiting to eligible.

At *e*, `TaskTx()` has stripped a byte from each TX1-TX4's buffers and has signaled `BINSEM_TXSTART` and is now waiting `BINSEM_TXDONE`. `TaskTx()` will not run again until another byte is received on RX5. At *f*, `ISR()` (see signal INT_B) begins sending data with the start bit for each transmitter TX1-TX4. Note the long time spent in `ISR()` (approx 400µs) between *f* and *g* – this is when

`ISR()` detects start bits on RX1-RX4 and starts tasks `TaskRx1()-TaskRx4()` via calls to `OSStartTask()`. Prior to *g*, time spent in `ISR()` was around 120µs. From this point forward, `ISR()` will update TX1-TX4 every four system ticks (i.e. one bit time, 833µs) with the required bit values representing the data being transmitted.

The time between *g* and *h* is spent in the scheduler, making `TaskRx1()-TaskRx4()` eligible,[16] followed by `TaskRx1()` running, since it has the highest priority of the four receiver tasks. `TaskRx2()`, `TaskRx3()` and `TaskRx4()` follow in quick succession at *i*, *j* and *k*, respectively. They follow in this order because each task has a lower task priority than the previous one. Each of the receiver tasks delays itself for right amount of system ticks so that it will run again in the middle of the next data bit, i.e. 1.5 bit times, or 1250µs, after the falling edge of the start bit (*f*).

From *k* to *l*, `TaskTx()` remains waiting for transmission on TX1-TX4 to complete, and `TaskRx1()-TaskRx4()` are delayed. Therefore `TaskTestCode()` is able to run despite its having the lowest priority. At *l*, `TaskRx1()`'s computed delay has expired, and it samples RX1 for a 1. Similarly, `TaskRx2()-TaskRx4()` sample RX2-RX4, respectively, all within a 250µs window centered around the middle of the bit period. Then `TaskRx1()-TaskRx4()` again delay themselves for a computed number of system ticks for the next data bit.

This process continues until each data bit has been sampled by `TaskRx1()-TaskRx4()`. Note at *m* and *n* how the order in which the receiver tasks run has been changed – this is not due to priorities changing, but rather due to differences in the number of system ticks each receiver task is computing as being required for the proper inter-data-bit delay. This also illustrates how each receiver `TaskRx1()-TaskRx4()`synchronizes itself to the incoming serial bitstream independently and with a resolution of a single system tick, or one quarter of a bit period. When the delays of multiple receiver tasks expire in the same system tick, then the tasks will run in priority order, and that is why the predominant order in Figure 3 is one based on task priority.

At *o*, `TaskRx1()-TaskRx4()` have each received valid data and put it into TX5's buffer. Transmission begins on TX5 at the next system tick, and sure enough, an ASCII '5' is being transmitted.

Processing Power In Reserve

It's instructive to note that between data bit samplings by `TaskRx1()-TaskRx4()`, `TaskTestCode()` is still able to run ten or

more times per bit period, over roughly half the bit period (400µs). This illustrates that there is still considerable processing power available for other tasks while `TaskRx1()-TaskRx4()` are active and receiving serial data.

As a test, three additional tasks with the same priority as `TaskTestCode()` were added to the application. Each task repeatedly delayed itself for one system tick. All communications continued without error. The main effect of the three additional tasks was that `TaskTestCode()` only ran two times per bit period, on average.

Multiple Bytes Received

Figure 4 shows multiple bytes received via RX5.

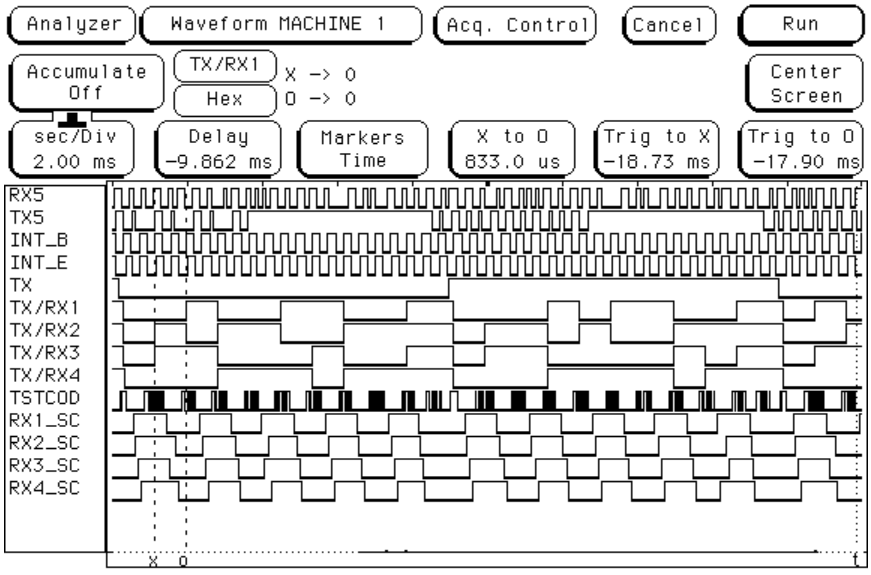

Figure 4: Multi-Byte Reception and Re-Transmission

System Responsiveness

Figure 5[17] can be used to observe the response times of the software transmitters and receivers when running the test code.

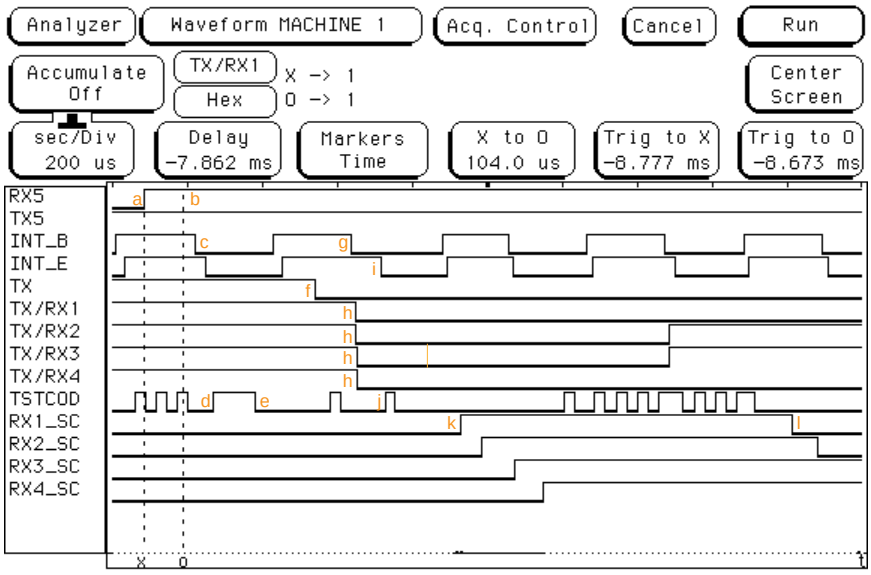

Figure 5: Response of Software Transmitters and Receivers under Test

At *a*, the stop bit of the 9600 baud transmission into RX5 has begun, and by *b*, it has ended and a valid test byte has been received into the hardware UART's receive buffer. `ISR()` begins at *c*, detects the incoming data, and places it into the hardware receiver's ring buffer `rx5Buff[]`. At *d*, `TaskTestCode()` strips the byte from `rx5Buff[]` and places the permuted copies into all four software transmitter buffers. By *e* it has finished, taking approximately 112µs, or 560 cycles at 20MHz, to put data into four buffers and call the RTOS service `OSSignalBinSem()` four times.

Between *e* and *f* the scheduler dispatches `TaskTx()`, and by *f* `TaskTx()` has stripped a byte from each software transmitter's buffer, activated all four transmitters in `ISR()`, and signaled the binary semaphore `BINSEM_TXSTART`. Almost immediately after the start of the next system tick, at *g*, the transmitters begin transmitting by sending out their start bits (*h*). Thus, it took roughly 430µs – or roughly one-half of a 1200 baud bit period – between the receipt of data on RX5 and its transmission on TX1-TX4.

By *i*, the start-bit detectors in `ISR()` have all detected activity on TX1-TX4. A timestamp for each software receiver is recorded, and `TaskRx1()-TaskRx4()` are started. Since the receiver tasks have higher priorities than `TaskTestCode()`, they would normally run next. However, the interrupt at *g* happened immediately prior to the scheduler dispatching `TaskTestCode()` – hence it runs one last time (*j*) before `TaskRx1()-TaskRx4()` start running. By *k*, `TaskRx1()` – the highest-priority software receiver – is already

running and is ready to delay itself until the middle of the first data bit on TX1 (*l*). Again, it has taken roughly one-half of a 1200 baud bit period between the start-bit detectors in `ISR()` starting a receiver task and the tasks `TaskRx1()-TaskRx4()` actually running.

Timing Specifications

Timing specifications for the quad software UART application were obtained from the above analysis and through empirical testing. The results are shown in Table 5.

Maximum delay between putting data into software transmitter and transmission beginning	< 500µs, less than one bit period
Maximum delay between detecting receive data's start bit and software receiver task starting	< 500µs, less than one bit period
Software transmitter timebase error	< 0.2%, assuming 20.000000MHz.
Software transmitter clock error	< 5%
Software receiver maximum sampling time error	±175µs, less than one-quarter bit period

Table 5: Timing Specifications for 1200 baud Quad Full-duplex Software UART

Building Your Own Application

This application is intended to be the framework for up to four full-duplex 1200 baud software UARTs, and leaves plenty of ROM and RAM for additional functionality. You can add your own code, whether it's called from `main()` or from one or more tasks, and use as few or as many of the software UARTs as you need.

If you add your own code in `main()`, you must ensure that it does not interfere with task execution. This usually means that it should never run for more than, say, one eighth of one bit period:

```
…
for (;;) {
  OSSched();
  /* your code here – make it short! */
  …
}
```

Listing 3: Adding User Code to main()

This approach is not recommended because the system's responsiveness suffers from the extra code that delays the operations of the scheduler. A better solution is to place your code in tasks with priorities lower than those of the software receivers and transmitters. This way, your application will run your code only when it is not busy with the software UARTs. Thus, the system will dynamically allocate its processing power – when there is no UART activity, your tasks will run flat-out, and when there is UART activity, Salvo's scheduler will ensure that the software UARTs are able to function without being held off by other parts of your application.[18]

The header file `d5.h` contains various symbols you can redefine to suit your application. For example, you change the crystal speed (`XTAL`) and baud rate (`BAUD`), the buffer sizes (`RX1_BUFF_SIZE`), the pin assignments (`TX1`, etc.) and where variables are located in RAM (`LOC_BCB`). You can also disable the test and debugging code (`ENABLE_TEST_CODE`, `ENABLE_TEST_PINS` and `RUN_TEST_CODE_TASK`).

Enhancements

Several avenues exist for reducing code size, reducing the minimum clock speed required, or porting to smaller PICmicro® devices.

The best opportunity for code size reduction, and with it greater performance, lies with `ISR()`. The compiler generates fairly large context save and restore code because the RTOS services called from within `ISR()` are not included in `int.c`. A source-code build that includes the relevant Salvo source files in `int.c` will result in a smaller – and therefore faster – `ISR()`.

The *available stack depth* (which is reduced by the two levels required by `ISR()`) can be increased in a source code build by in-lining `OSSched()`.

Using a *fixed buffer size* will reduce RAM and ROM requirements.

A source-code build can be used with task priorities *disabled*. This will reduce ROM somewhat, and may be adequate for certain applications with only a few tasks.

Using *your own binary semaphores* instead of Salvo's is not recommended. While this approach may decrease RAM usage by up to 8 bytes,[19] it's unlikely to reduce ROM usage, and performance will probably suffer. That's because these binary semaphores will have to be polled by the transmitter and receiver tasks on a regular basis (e.g. every system tick) in order to respond quickly to software receive and transmit activity. This major burden on the system is avoided by using the event-driven RTOS approach.

Conclusion

A robust timing-critical serial communications application can be built with the Salvo RTOS. Careful structuring of the interrupt service routine may be required. Through the use of task priorities, the Salvo scheduler dynamically allocates processing power where it's needed most. With adequate system timer resolution, task delay services can be used to synchronize receivers with the incoming bitstream. The application is highly scalable, with plenty of ROM and RAM left for additional functionality.

Build Results

Listing 4 displays the linker output for project `salvo\demo\d5\sysa\d5lib.pjt` when the quad UART application is built using a Salvo standard library for the PIC16F877 with debugging and test code included. Library `spl42Caa.lib` supports multitasking and delays with event signaling from foreground and the background levels. This build was done with full 20-byte receive buffers for RX1-RX4, 4-byte transmit buffers for TX1-4, and 10-byte buffers for RX5 and TX5. RAM bank1 contains Salvo variables, RAM bank2 contains buffer control blocks, RX5 and TX5 buffers, receiver timestamps, and other variables used in the application, and RAM bank3 contains TX1-4 and RX1-4 buffers.

```
Linking:
Command line: "C:\HT-PIC\BIN\PICC.EXE -G -INTEL -Md5lib.map -16F877
-oD5LIB.HEX -fakelocal -I\salvo\include D:\SALVO\DEMO\D5\MAIN.OBJ
D:\SALVO\SOURCE\MEM.OBJ D:\SALVO\DEMO\D5\INT.OBJ
D:\SALVO\DEMO\D5\SWRXUART.OBJ D:\SALVO\DEMO\D5\SWTXUART.OBJ
D:\SALVO\DEMO\D5\BUFF.OBJ D:\SALVO\LIBRARY\SLP42CAA.LIB "
Enter PICC -HELP for help

Memory Usage Map:

Program ROM   $0000 - $0812  $0813 (  2067) words
Program ROM   $0E33 - $0FFF  $01CD (   461) words
Program ROM   $2007 - $2007  $0001 (     1) words
                             $09E1 (  2529) words total Program ROM

Bank 0 RAM    $0020 - $003A  $001B (    27) bytes
Bank 0 RAM    $0070 - $0072  $0003 (     3) bytes
                             $001E (    30) bytes total Bank 0 RAM

Bank 1 RAM    $00A0 - $00D0  $0031 (    49) bytes total Bank 1 RAM
Bank 2 RAM    $0110 - $0161  $0052 (    82) bytes total Bank 2 RAM
Bank 3 RAM    $0190 - $01EF  $0060 (    96) bytes total Bank 3 RAM
```

Listing 4: Build Results for Library Build

Table 6 illustrates the build results for different configurations with the symbols `ENABLE_TEST_CODE`, `ENABLE_TEST_PINS` and `RUN_TEST_CODE_TASK` all defined as non-zero in `d5.h`.

UART		ROM	RAM			
receivers	transmitters	words	Bank 0 bytes	Bank 1 bytes	Bank 2 bytes	Bank 3 bytes
1	0	1434	25	20	39	20
2	0	1648	25	25	47	40
3	0	1862	25	30	55	60
4	0	2077	25	35	63	80
0	1	1264	26	29	33	4
0	2	1331	27	29	37	8
0	3	1399	28	29	41	12
0	4	1468	29	29	45	16
1	1	1682	27	34	46	24
2	2	1963	28	39	58	48
3	3	2245	29	44	70	72
4	4	2529	30	49	82	96

Table 6: Build Results for Different UART Configurations

Table 7 illustrates the results of a library build with one receiver and one transmitter. All the Salvo variables are in RAM Bank 1 and the UART buffers, control blocks, etc. have all been placed in RAM Bank 3:

UART		ROM	RAM			
receivers	transmitters	words	Bank 0 bytes	Bank 1 bytes	Bank 2 bytes	Bank 3 bytes
1	1	1735	27	34	0	70

Table 7: Build Results for all UART Variables in RAM Bank 3

For this implementation of a single full-duplex UART, 237 bytes of RAM remain free, representing 64% of available RAM. Table 8 illustrates results of another configuration when all of the non-Salvo variables are placed in RAM bank 0. Of note is the reduction in ROM because of the reduced need for bank switching.

UART		ROM	RAM			
receivers	transmitters	words	Bank 0 bytes	Bank 1 bytes	Bank 2 bytes	Bank 3 bytes
0	1	1214	63	29	0	0

Table 8: Build Results for all UART Variables in RAM Bank 0

Source Code Listings

salvocfg.h

```
/*****************************************************
Copyright (C) 1995-2001 Pumpkin, Inc. and its
Licensor(s). Freely distributable.

$Source: C:\\RCS\\D\\salvo\\demo\\d5\\sysa\\salvocfg.h,v $
$Author: aek $
$Revision: 1.1 $
$Date: 2001-08-09 21:46:47-07 $

Configuration options for demo program.

*****************************************************/

#define TEST_SYSTEM_A                   TRUE

#if defined(MAKE_WITH_STD_LIB) || defined(MAKE_WITH_FREE_LIB)

#define OSUSE_LIBRARY                   TRUE
#if   defined(MAKE_WITH_FREE_LIB)
#define OSLIBRARY_TYPE                  OSF
#define OSTASKS                         3
#elif defined(MAKE_WITH_STD_LIB)
#define OSLIBRARY_TYPE                  OSL
#define OSTASKS                         6
#endif
#define OSLIBRARY_CONFIG                OSA
#define OSLIBRARY_VARIANT               OSA

#define OSEVENTS                        3
#define OSEVENT_FLAGS                   0
#define OSMESSAGE_QUEUES                0

#endif
```

d5.h

```
/*********************************************************
Copyright (C) 1995-2001 Pumpkin, Inc. and its
Licensor(s). Freely distributable.

$Source: C:\\RCS\\D\\salvo\\demo\\d5\\d5.h,v $
$Author: aek $
$Revision: 1.5 $
$Date: 2001-08-19 21:31:57-07 $

Header file for demo program.

*********************************************************/

/*********************************************************
****                                                 ****
**                                                     **
User-definable symbols and settings. Each and every symbol
in this block can be changed to suit the user's preferences
and/or requirements.

Tested at 10, 12.5, 16, and 20MHz for 1, 2, 3 and 4 1200bps
rx/tx pairs, respectively.

Also works with other baud rates, e.g. 1 2400bps rx/tx pair
at 20MHz.

**                                                     **
****                                                 ****
*********************************************************/
#define XTAL                20000000    /* Xtal freq    */
#define BAUD                1200        /* baud rate    */

#define RECEIVERS           4           /* # s/w rcvrs  */
#define TRANSMITTERS        4           /* # s/w xmtrs  */

#define RX1_BUFF_SIZE       20          /* size of s/w  */
#define RX2_BUFF_SIZE       20          /*  receiver    */
#define RX3_BUFF_SIZE       20          /*  buffers     */
#define RX4_BUFF_SIZE       20          /*  ""          */
#define TX1_BUFF_SIZE       4           /* size of s/w  */
#define TX2_BUFF_SIZE       4           /*  xmitter     */
#define TX3_BUFF_SIZE       4           /*  buffers     */
#define TX4_BUFF_SIZE       4           /*  ""          */

#define RX5_BUFF_SIZE       10          /* size of UART */
                                        /*  receiver    */
                                        /*    buffer    */
#define TX5_BUFF_SIZE       10          /* size of UART */
                                        /*  xmitter     */
                                        /*    buffer    */

#define RX1                 RB1         /* receiver and */
#define RX2                 RB3         /*  xmitter pin */
#define RX3                 RB5         /*  assignments */
#define RX4                 RB7         /*  ""          */
#define TX1                 RB0         /* Normally all */
#define TX2                 RB2         /*  are on same */
#define TX3                 RB4         /*  port.       */
#define TX4                 RB6         /*  ""          */

#define InitUARTTris()      { TRISB = 0xAA; } /* must   */
#define InitUARTPort()      { PORTB = 0xFF; } /*  match */
                                        /*  port bits   */
                                        /*  above.      */

#define LOC_BUFF            bank3       /* RAM bank for */
                                        /*  s/w receiver*/
                                        /*  and xmitter */
                                        /*  buffers.    */
#define LOC_BCB             bank2       /* RAM bank for */
                                        /*  s/w receiver*/
                                        /*  and xmitter */
                                        /*  buffer ctrl */
                                        /*  blocks.     */

#define LOC_RX5             bank2       /* RAM bank for */
                                        /*  UART's      */
```

```
                                        /*  receiver    */
                                        /*  buffer, etc.*/
#define LOC_TX5             bank2       /* RAM bank for */
                                        /*  UART's      */
                                        /*  transmitter */
                                        /*  buffer, etc.*/

#define LOC_RX_LOCALS       bank2       /* RAM bank for */
                                        /*  Rx tasks'   */
                                        /*  static local*/
                                        /*  variables.  */
#define LOC_INT_LOCALS      bank2       /* RAM bank for */
                                        /*  ISR's       */
                                        /*  static local*/
                                        /*  variables.  */

#define LOC_TEST_VARS       bank2       /* RAM bank for */
                                        /*  test code   */
                                        /*  variables   */

#define ENABLE_TEST_CODE    1
#define ENABLE_TEST_PINS    1
#define RUN_TEST_CODE_TASK  1

/*************************************************************
****                                                      ****
**                                                          **

**                                                          **
****                                                      ****
*************************************************************/
#if ( TRANSMITTERS < 0 ) || ( TRANSMITTERS > 4 )
#error This configuration not supported.
#endif

#if ( RECEIVERS < 0 ) || ( RECEIVERS > 4 )
#error This configuration not supported.
#endif

/*************************************************************
****                                                      ****
**                                                          **
Normally a user wouldn't change any of these ...

**                                                          **
****                                                      ****
*************************************************************/
/* taskID's for each sw UART task.                          */
#define TASKTESTCODE_TASKID 1
#define TASKTX_TASKID       2
#define TASKRX1_TASKID      3
#define TASKRX2_TASKID      4
#define TASKRX3_TASKID      5
#define TASKRX4_TASKID      6

/* task priorities for each sw UART task. Rx tasks need     */
/*  to be highest. A higher priority could be used for      */
/*  a dramatic error-handling task.                         */
#define TASKRX1_PRIO        1
#define TASKRX2_PRIO        2
#define TASKRX3_PRIO        3
#define TASKRX4_PRIO        4
#define TASKTX_PRIO         5
#define TASKTESTCODE_PRIO   6


/*************************************************************
****                                                      ****
**                                                          **
Test code.

**                                                          **
****                                                      ****
*************************************************************/
#if ENABLE_TEST_PINS
#define InitTestPins()      { TRISD &= 0x00; PORTD = 0x00; }
#define ToggleTPRx1SC()     { RD0 ^= 1; }
#define ToggleTPRx2SC()     { RD1 ^= 1; }
#define ToggleTPRx3SC()     { RD2 ^= 1; }
#define ToggleTPRx4SC()     { RD3 ^= 1; }
#define ToggleTPMain()      { RD4 ^= 1; }
#define ToggleTPIntDone()   { RD5 ^= 1; }
#define ToggleTPInt()       { RD6 ^= 1; }
#define ToggleTPTx()        { RD7 ^= 1; }
```

```
#else

#define InitTestPins()
#define ToggleTPRx1SC()
#define ToggleTPRx2SC()
#define ToggleTPRx3SC()
#define ToggleTPRx4SC()
#define ToggleTPMain()
#define ToggleTPIntDone()
#define ToggleTPInt()
#define ToggleTPTx()

#endif


/*************************************************************
****                                                     ****
**                                                         **
Target-specific #defines.

**                                                         **
****                                                     ****
*************************************************************/
#define DisableTMR1()       { TMR1ON = 0; }
#define EnableTMR1()        { TMR1ON = 1; }
static volatile unsigned int TMR1 @ 0x0E;

#define PORTBIT(adr, bit)   ((unsigned)(&adr)*8+(bit))

/* for use with 1:8 postscalar with 1/4-bit resolution.    */
#define TMR2_RELOAD         (XTAL/BAUD)/(4*8*4)



#define INT_TIME            QUARTER_BIT_TIME
#define ONE_SEC             1000000  /* in microseconds    */
#define FULL_BIT_TIME       ONE_SEC/BAUD
#define QUARTER_BIT_TIME    ONE_SEC/BAUD/4
#define FULL_BIT_CYCLES     XTAL/BAUD/(4*1)
#define FULL_BIT_TICKS      (FULL_BIT_TIME)/(INT_TIME)

#define TASKTESTCODE_P      OSTCBP(TASKTESTCODE_TASKID)
#define TASKTX_P            OSTCBP(TASKTX_TASKID)
#define TASKRX1_P           OSTCBP(TASKRX1_TASKID)
#define TASKRX2_P           OSTCBP(TASKRX2_TASKID)
#define TASKRX3_P           OSTCBP(TASKRX3_TASKID)
#define TASKRX4_P           OSTCBP(TASKRX4_TASKID)

#define BINSEM_TXBUFF_P     OSECBP(1)
#define BINSEM_TXDONE_P     OSECBP(2)
#define BINSEM_TXSTART_P    OSECBP(3)

#define TXSTATE_IDLE        0
#define TXSTATE_START       1
#define TXSTATE_DATA0       2
#define TXSTATE_DATA1       3
#define TXSTATE_DATA2       4
#define TXSTATE_DATA3       5
#define TXSTATE_DATA4       6
#define TXSTATE_DATA5       7
#define TXSTATE_DATA6       8
#define TXSTATE_DATA7       9
#define TXSTATE_STOP        10
#define TXSTATE_DONE        11

#define RX1BCB              rxBcbArray[0]
#define RX2BCB              rxBcbArray[1]
#define RX3BCB              rxBcbArray[2]
#define RX4BCB              rxBcbArray[3]
#define TX1BCB              txBcbArray[0]
#define TX2BCB              txBcbArray[1]
#define TX3BCB              txBcbArray[2]
#define TX4BCB              txBcbArray[3]

#define RX1BCBP             &rxBcbArray[0]
#define RX2BCBP             &rxBcbArray[1]
#define RX3BCBP             &rxBcbArray[2]
#define RX4BCBP             &rxBcbArray[3]
#define TX1BCBP             &txBcbArray[0]
#define TX2BCBP             &txBcbArray[1]
#define TX3BCBP             &txBcbArray[2]
#define TX4BCBP             &txBcbArray[3]
```

```
/* buffer control block */
struct bcb {
    unsigned char count;
    unsigned char inP;
    unsigned char outP;
    unsigned char size;
}
typedef struct bcb typeBcb;


#define typeSize unsigned char
#define typeBuff unsigned char
#define typeBuffP LOC_BUFF typeBuff *
#define typeBcbP LOC_BCB typeBcb *


void TaskRx1(void);
void TaskRx2(void);
void TaskRx3(void);
void TaskRx4(void);
void TaskTx(void);

void InitUART(void);
void InitBcb(typeBcbP txBcbP, typeSize size);
unsigned char GetBuff(unsigned char * dataP,
                      typeBcbP        bcbP,
                      typeBuffP       buffP );
unsigned char GetRx5Buff(unsigned char * dataP);
unsigned char PutBuff(unsigned char data,
                      typeBcbP      bcbP,
                      typeBuffP     buffP );
unsigned char PutTx5Buff(unsigned char data);

#define GetRx1Buff(c) GetBuff(c, RX1BCBP, rx1Buff)
#define GetRx2Buff(c) GetBuff(c, RX2BCBP, rx2Buff)
#define GetRx3Buff(c) GetBuff(c, RX3BCBP, rx3Buff)
#define GetRx4Buff(c) GetBuff(c, RX4BCBP, rx4Buff)
#define GetTx1Buff(c) GetBuff(c, TX1BCBP, tx1Buff)
#define GetTx2Buff(c) GetBuff(c, TX2BCBP, tx2Buff)
#define GetTx3Buff(c) GetBuff(c, TX3BCBP, tx3Buff)
#define GetTx4Buff(c) GetBuff(c, TX4BCBP, tx4Buff)
#define PutRx1Buff(c) PutBuff(c, RX1BCBP, rx1Buff)
#define PutRx2Buff(c) PutBuff(c, RX2BCBP, rx2Buff)
#define PutRx3Buff(c) PutBuff(c, RX3BCBP, rx3Buff)
#define PutRx4Buff(c) PutBuff(c, RX4BCBP, rx4Buff)
#define PutTx1Buff(c) PutBuff(c, TX1BCBP, tx1Buff)
#define PutTx2Buff(c) PutBuff(c, TX2BCBP, tx2Buff)
#define PutTx3Buff(c) PutBuff(c, TX3BCBP, tx3Buff)
#define PutTx4Buff(c) PutBuff(c, TX4BCBP, tx4Buff)

#define putch(a) putchar(a)


#ifndef MAIN_C_INCLUDES

extern LOC_INT_LOCALS txState;
extern unsigned char tx1Data;
extern unsigned char tx2Data;
extern unsigned char tx3Data;
extern unsigned char tx4Data;

extern LOC_INT_LOCALS unsigned int rx1Timestamp;
extern LOC_INT_LOCALS unsigned int rx2Timestamp;
extern LOC_INT_LOCALS unsigned int rx3Timestamp;
extern LOC_INT_LOCALS unsigned int rx4Timestamp;

extern LOC_RX5 unsigned char rx5Count;
extern LOC_RX5 unsigned char rx5InP, rx5OutP;
extern LOC_RX5 unsigned char rx5Buff[];
extern LOC_TX5 unsigned char tx5Count;
extern LOC_TX5 unsigned char tx5InP, tx5OutP;
extern LOC_TX5 unsigned char tx5Buff[];

extern LOC_INT_LOCALS struct {
    unsigned char tx1Active:1;
    unsigned char tx2Active:1;
    unsigned char tx3Active:1;
    unsigned char tx4Active:1;
    } txStatus;

extern LOC_INT_LOCALS struct {
    unsigned char rx1Error:1;
    unsigned char rx2Error:1;
```

```
    unsigned char rx3Error:1;
    unsigned char rx4Error:1;
    unsigned char rx1Active:1;
    unsigned char rx2Active:1;
    unsigned char rx3Active:1;
    unsigned char rx4Active:1;
    } rxStatus;

extern LOC_BUFF typeBuff rx1Buff[RX1_BUFF_SIZE];
extern LOC_BUFF typeBuff rx2Buff[RX2_BUFF_SIZE];
extern LOC_BUFF typeBuff rx3Buff[RX3_BUFF_SIZE];
extern LOC_BUFF typeBuff rx4Buff[RX4_BUFF_SIZE];
extern LOC_BUFF typeBuff tx1Buff[TX1_BUFF_SIZE];
extern LOC_BUFF typeBuff tx2Buff[TX2_BUFF_SIZE];
extern LOC_BUFF typeBuff tx3Buff[TX3_BUFF_SIZE];
extern LOC_BUFF typeBuff tx4Buff[TX4_BUFF_SIZE];


extern LOC_BCB typeBcb rxBcbArray[4];
extern LOC_BCB typeBcb txBcbArray[4];

extern LOC_RX5 unsigned char rx5Count;
extern LOC_RX5 unsigned char rx5InP, rx5OutP;
extern LOC_RX5 unsigned char rx5Buff[];
extern LOC_RX5 unsigned char tx5Count;
extern LOC_RX5 unsigned char tx5InP, tx5OutP;
extern LOC_RX5 unsigned char tx5Buff[];

#endif
```

main.c

```
/*********************************************************
Copyright (C) 1995-2001 Pumpkin, Inc. and its
Licensor(s). Freely distributable.

$Source: C:\\RCS\\D\\salvo\\demo\\d5\\main.c,v $
$Author: aek $
$Revision: 1.11 $
$Date: 2001-08-19 21:31:57-07 $

Demo program. Runs on Microchip PICDEM-2 (PIC16C77,
PIC16F877, PIC18C452, etc.) and MPLAB-ICD (PIC16F877).

Implements quad full-duplex 1200bps UARTs in software.
Requires Salvo v2.3.0 or higher.

*********************************************************/

#ifndef MAIN_C_INCLUDES
#define MAIN_C_INCLUDES

#include "d5.h"
#include "salvo.h"

/*********************************************************
****                                                 ****
**                                                     **
Test-system-specific configuration.

**                                                     **
****                                                 ****
*********************************************************/
#if   defined(TEST_SYSTEM_A)
#ifdef _16F877
__CONFIG(FOSC0 | UNPROTECT);
#else
__CONFIG(XT | UNPROTECT);
#endif
#elif defined(TEST_SYSTEM_F)
__CONFIG(1, FOSC0 | UNPROTECT);
#elif defined(TEST_SYSTEM_H)
__CONFIG(FOSC0 | UNPROTECT);
#endif


/*********************************************************
****                                                 ****
**                                                     **
Global variable declarations.
**                                                     **
****                                                 ****
*********************************************************/
/* current value of character being shifted out by s/w  */
/*  UART. Cannot be banked because PutTxNChar() expects */
/*  non-banked parameter.                               */
/* state of transmitter state machine (runs in ISR).    */
/* xmitter status is used to control which xmitters are */
/*  currently running.                                  */
/* buffer control blocks for transmitters.              */
/* ring buffers for transmitters.                       */
#if TRANSMITTERS >= 1
unsigned char tx1Data;
LOC_INT_LOCALS txState;
LOC_INT_LOCALS struct {
    unsigned char tx1Active:1;
    unsigned char tx2Active:1;
    unsigned char tx3Active:1;
    unsigned char tx4Active:1;
    } txStatus;
LOC_BCB typeBcb txBcbArray[TRANSMITTERS];
LOC_BUFF typeBuff tx1Buff[TX1_BUFF_SIZE];
#endif
#if TRANSMITTERS >= 2
unsigned char tx2Data;
LOC_BUFF typeBuff tx2Buff[TX2_BUFF_SIZE];
#endif
#if TRANSMITTERS >= 3
unsigned char tx3Data;
```

```
LOC_BUFF typeBuff tx3Buff[TX3_BUFF_SIZE];
#endif
#if TRANSMITTERS >= 4
unsigned char tx4Data;
LOC_BUFF typeBuff tx4Buff[TX4_BUFF_SIZE];
#endif


/* receiver status is used to control which receivers   */
/*  are running and also to flag rx errors.             */
/* rx timestamps are used to mark when the bitstream's  */
/*  start bit goes low. In current value of TMR1.       */
/* buffer control blocks for receivers.                 */
/* ring buffers for receivers.                          */
#if RECEIVERS >= 1
LOC_INT_LOCALS unsigned int rx1Timestamp;
LOC_INT_LOCALS struct {
    unsigned char rx1Error:1;
    unsigned char rx2Error:1;
    unsigned char rx3Error:1;
    unsigned char rx4Error:1;
    unsigned char rx1Active:1;
    unsigned char rx2Active:1;
    unsigned char rx3Active:1;
    unsigned char rx4Active:1;
    } rxStatus;
LOC_BCB typeBcb rxBcbArray[RECEIVERS];
LOC_BUFF typeBuff rx1Buff[RX1_BUFF_SIZE];
#endif
#if RECEIVERS >= 2
LOC_INT_LOCALS unsigned int rx2Timestamp;
LOC_BUFF typeBuff rx2Buff[RX2_BUFF_SIZE];
#endif
#if RECEIVERS >= 3
LOC_INT_LOCALS unsigned int rx3Timestamp;
LOC_BUFF typeBuff rx3Buff[RX3_BUFF_SIZE];
#endif
#if RECEIVERS >= 4
LOC_INT_LOCALS unsigned int rx4Timestamp;
LOC_BUFF typeBuff rx4Buff[RX4_BUFF_SIZE];
#endif


/* control vars and ring buffers for RX5/TX5: built-in  */
/*  UART. We use explicit vars instead of control       */
/*  blocks because GetRx5Buff() and PutRx5Buff() are    */
/*  different from GetBuff() and PutBuff() due          */
/*  to interrupts.                                      */
LOC_RX5 unsigned char rx5Count;
LOC_RX5 unsigned char rx5InP, rx5OutP;
LOC_RX5 unsigned char rx5Buff[RX5_BUFF_SIZE];
LOC_TX5 unsigned char tx5Count;
LOC_TX5 unsigned char tx5InP, tx5OutP;
LOC_TX5 unsigned char tx5Buff[TX5_BUFF_SIZE];

/* local function declarations.                         */
#if ENABLE_TEST_CODE
void TestCode(void);
#if RUN_TEST_CODE_TASK
void TaskTestCode(void);
#endif
#endif

/* context-switching labels.                            */
_OSLabel(TaskTestCode1)


/*********************************************************
****                                                 ****
**                                                     **
main()

Initialize relevant hardware registers, initialize Salvo and
create tasks (idle task is created automatically), enable
interrupts and start multitasking.

IMPORTANT NOTE: Since PICC (stackless) compiler is used,
background calls to OSSignalBinSem() must be protected,
since it's also called in the foreground.

**                                                     **
****                                                 ****
*********************************************************/
int main( void )
```

```
{
    #if ENABLE_TEST_PINS
    /* setup test port.                                    */
    InitTestPins();
    #endif

    /* initialize the port we'll be bit-banging.          */
    InitUARTTris();
    InitUARTPort();

    /* configure TMR1 for 1:1 prescale, use internal      */
    /*  clock. TMR1 is used to count instruction          */
    /*  cycles. Note that TMR1CS is 0, and will remain    */
    /*  unchanged.                                        */
    T1CON = 0b00000001;

    /* initialize RS-232 transmitter software.            */
    InitUART();

    /* set TMR2 for 1:8 postcale. TMR2 is used to call    */
    /*  the Salvo system timer every quarter bit          */
    /*  period.                                           */
    PR2 = TMR2_RELOAD;
    T2CON = 0b00111100;

    /* initialize global flags.                           */
    #if RECEIVERS >= 1
    rxStatus.rx1Error  = 0;
    rxStatus.rx2Error  = 0;
    rxStatus.rx3Error  = 0;
    rxStatus.rx4Error  = 0;
    rxStatus.rx1Active = 0;
    rxStatus.rx2Active = 0;
    rxStatus.rx3Active = 0;
    rxStatus.rx4Active = 0;
    #endif
    #if TRANSMITTERS >= 1
    txState = TXSTATE_IDLE;
    txStatus.tx1Active = 0;
    txStatus.tx2Active = 0;
    txStatus.tx3Active = 0;
    txStatus.tx4Active = 0;
    #endif

    /* initialize Salvo.                                  */
    OSInit();

    /* create tasks.                                      */
    #if TRANSMITTERS >= 1
    OSCreateTask(TaskTx,   TASKTX_P,  TASKTX_PRIO);
    #endif
    #if RECEIVERS >= 1
    OSCreateTask(TaskRx1,  TASKRX1_P, TASKRX1_PRIO);
    #endif
    #if ENABLE_TEST_CODE && RUN_TEST_CODE_TASK
    OSCreateTask(TaskTestCode, TASKTESTCODE_P, TASKTESTCODE_PRIO);
    #endif

    /* these tasks will work only with the standard     */
    /*  libraries, since the freeware libraries support */
    /*  only 3 tasks.                                   */
    #if RECEIVERS >= 2
    OSCreateTask(TaskRx2,  TASKRX2_P, TASKRX2_PRIO);
    #endif
    #if RECEIVERS >= 3
    OSCreateTask(TaskRx3,  TASKRX3_P, TASKRX3_PRIO);
    #endif
    #if RECEIVERS >= 4
    OSCreateTask(TaskRx4,  TASKRX4_P, TASKRX4_PRIO);
    #endif

    /* transmit buffers are initially empty.              */
    #if TRANSMITTERS >= 1
    OSCreateBinSem(BINSEM_TXBUFF_P,  0);
    OSCreateBinSem(BINSEM_TXDONE_P,  0);
    OSCreateBinSem(BINSEM_TXSTART_P, 0);
    #endif

    /* enable TMR2 interrupts, enable peripheral          */
    /*  interrupts, enable (global) interrupts.           */
    TMR2IE = 1;
    PEIE   = 1;
    OSEnableInts();
```

```
    /* start multitasking.                                  */
    for (;;) {
        OSSched();

        #if ENABLE_TEST_CODE && !RUN_TEST_CODE_TASK
        TestCode();
        #endif
    }
}

/*********************************************************
****                                                 ****
**                                                     **
TestCode()

Test code -- feed each transmit buffer with permuted data.
Since each receiver is listening to its associated
transmitter, undo the permutation and send the data back
out the hardware UART. Test pin marks how long this op takes
and when we're in here.

OSSignalBinSem() must be protected for use with stackless
compilers because it's also called from within ISR().

**                                                     **
****                                                 ****
*********************************************************/
#if ENABLE_TEST_CODE
void TestCode( void )
{
    unsigned char data;


    ToggleTPMain();

    #if TRANSMITTERS >= 1
    if ( GetRx5Buff(&data) == TRUE ) {

        if ( PutTx1Buff(data)) {
            OSProtect();
            OSSignalBinSem(BINSEM_TXBUFF_P);
            OSUnprotect();
        }

        #if TRANSMITTERS >= 2
        if ( PutTx2Buff(~data) ) {
            OSProtect();
            OSSignalBinSem(BINSEM_TXBUFF_P);
            OSUnprotect();
        }
        #endif

        #if TRANSMITTERS >= 3
        if ( PutTx3Buff((data << 4) | (data >> 4)) ) {
            OSProtect();
            OSSignalBinSem(BINSEM_TXBUFF_P);
            OSUnprotect();
        }
        #endif

        #if TRANSMITTERS >= 4
        if ( PutTx4Buff(~((data << 4) | (data >> 4))) ) {
            OSProtect();
            OSSignalBinSem(BINSEM_TXBUFF_P);
            OSUnprotect();
        }
        #endif
    }
    #endif /* #if TRANSMITTERS >= 1 */


    #if RECEIVERS >= 1
    if ( GetRx1Buff(&data) == TRUE )
        PutTx5Buff(data);
    #endif

    #if RECEIVERS >= 2
    if ( GetRx2Buff(&data) == TRUE )
        PutTx5Buff(~data);
    #endif

    #if RECEIVERS >= 3
    if ( GetRx3Buff(&data) == TRUE )
        PutTx5Buff((data << 4) | (data >> 4));
```

```
    #endif

    #if RECEIVERS >= 4
    if ( GetRx4Buff(&data) == TRUE )
        PutTx5Buff(~((data << 4) | (data >> 4)));
    #endif

    ToggleTPMain();
}
#endif

/*************************************************************
****                                                     ****
**                                                       **
TaskTestCode()

Run test code as a task. Return to scheduler when done.

**                                                       **
****                                                     ****
*************************************************************/
#if ENABLE_TEST_CODE && RUN_TEST_CODE_TASK
void TaskTestCode( void )
{
    for (;;) {
        TestCode();

        OS_Yield(TaskTestCode1);
    }
}
#endif

/*************************************************************
****                                                     ****
**                                                       **
InitUART()

Initialize the hardware UART as a 9600bps UART. This UART
has two ring buffers for receive and transmit.

**                                                       **
****                                                     ****
*************************************************************/
void InitUART( void )
{
    /* reset Tx ring buffer pointers.                   */
    tx5Count = 0;
    tx5InP   = 0;
    tx5OutP  = 0;
    rx5Count = 0;
    rx5InP   = 0;
    rx5OutP  = 0;

    /* PORTC is output on pin 6, rest is input.         */
    /* Set Tx out high to avoid bad chars.              */
    PORTC |= 0x40;     /* 01000000 */
    TRISC |= 0x80;    /* 10000000 */
    TRISC &= 0xBF;    /* 10111111 */

    /* Serial Port is ON, 8-bit data reception,         */
    /*  continuous receive.                             */
    RCSTA =  0x90;    /* 10x1xxxx */

    /* force Tx ints off now to avoid spurious          */
    /*  outgoing chars when transmitter is enabled.     */
    TXIE = 0;

    /* 8-bit transmit, Tx enabled, Async, BRGH=1        */
    TXSTA =  0x24;    /* x010x100 */

    /* 9600 baud.                                       */
    SPBRG = XTAL/16/9600-1;
}

/*************************************************************
****                                                     ****
**                                                       **
GetRx5Buff(dataP)

Removes a character (if present) from the hardware UART's
receiver ring buffer and copies it to variable specified.

Interrupt control is required because PutRx5Buff() is in
ISR.
```

```
Returns:    TRUE if a character was present.
            FALSE if buffer was empty.

**                                                          **
****                                                      ****
************************************************************/
unsigned char GetRx5Buff( unsigned char * dataP )
{
    if ( rx5Count ) {
        *dataP = rx5Buff[rx5OutP++];

        if ( rx5OutP > RX5_BUFF_SIZE-1 )
            rx5OutP = 0;

        OSDisableInts();
        rx5Count--;
        OSEnableInts();

        return TRUE;
    }
    else
        return FALSE;
}

/************************************************************
****                                                      ****
**                                                          **
PutTx5Buff(data)

Puts the character into the hardware UART's xmitter
ring buffer if room is available.

Interrupt control is required because GetTx5Buff() is in
ISR.

Returns:    TRUE on if there was room in buffer.
            FALSE if buffer was full.

**                                                          **
****                                                      ****
************************************************************/
unsigned char PutTx5Buff(unsigned char data)
{
    if ( tx5Count < TX5_BUFF_SIZE )
    {
        tx5Buff[tx5InP++] = data;

        if ( tx5InP > TX5_BUFF_SIZE-1 )
            tx5InP = 0;

        OSDisableInts();
        tx5Count++;
        OSEnableInts();

        return TRUE;
    }
    else
        return FALSE;
}

/************************************************************
****                                                      ****
**                                                          **
OSIdleFnHook()

Used primarily for debug. User can add idle task code in
here or create a new task in its place.

Test pin marks when idle function runs.

**                                                          **
****                                                      ****
************************************************************/
void OSIdleFnHook( void )
{
    ;
}

#endif
```

buff.c

```
/*********************************************************
Copyright (C) 1995-2001 Pumpkin, Inc. and its
Licensor(s). Freely distributable.

$Source: C:\\RCS\\D\\salvo\\demo\\d5\\buff.c,v $
$Author: aek $
$Revision: 1.10 $
$Date: 2001-08-19 21:17:09-07 $

Support file.

*********************************************************/

#ifndef BUFF_C_INCLUDES
#define BUFF_C_INCLUDES

#include "d5.h"
#include "salvo.h"

#if ( RECEIVERS >= 1 ) || ( TRANSMITTERS >= 1 )
/*********************************************************
****                                                 ****
**                                                     **
InitBcb()

Initialize the specified buffer by setting the buffer's
control parameters, which reside in the buffer control block.

Not compatible with buffer functions called from ISR().

**                                                     **
****                                                 ****
*********************************************************/
void InitBcb( typeBcbP bcbP, typeSize size )
{
    bcbP->count = 0;
    bcbP->inP   = 0;
    bcbP->outP  = 0;
    bcbP->size  = size;
}


/*********************************************************
****                                                 ****
**                                                     **
GetBuff()

Get char from buffer and wrap pointer if necessary.

**                                                     **
****                                                 ****
*********************************************************/
unsigned char GetBuff( unsigned char * dataP,
                       typeBcbP        bcbP,
                       typeBuffP       buffP )
{
    if ( bcbP->count ) {
        *dataP = buffP[bcbP->outP++];

        if ( bcbP->outP >= bcbP->size )
            bcbP->outP = 0;

        bcbP->count--;

        return TRUE;
    }
    else
        return FALSE;
}


/*********************************************************
****                                                 ****
**                                                     **
PutBuff()

Insert char into buffer and wrap pointer if necessary.
```

```
**                                                            **
****                                                        ****
************************************************************/
unsigned char PutBuff( unsigned char data,
                       typeBcbP      bcbP,
                       typeBuffP     buffP )
{
    if ( bcbP->count < bcbP->size )
    {
        buffP[bcbP->inP++] = data;

        if ( bcbP->inP >= bcbP->size )
            bcbP->inP = 0;

        bcbP->count++;

        return TRUE;
    }
    else
        return FALSE;
}
#endif


#endif
```

int.c

```
/*********************************************************
Copyright (C) 1995-2001 Pumpkin, Inc. and its
Licensor(s). Freely distributable.

$Source: C:\\RCS\\D\\salvo\\demo\\d5\\int.c,v $
$Author: aek $
$Revision: 1.5 $
$Date: 2001-08-19 21:31:56-07 $

Support file.

*********************************************************/

#ifndef INT_C_INCLUDES
#define INT_C_INCLUDES

#include "d5.h"
#include "salvo.h"

#if TRANSMITTERS >= 1
unsigned char intPS;
#endif



/*********************************************************
****                                                 ****
**                                                     **
intVector()

Periodic interrupt via TMR2 occurs every quarter bit time.
TMR2 is reloaded automatically from period register PR2. This
is the system tick period. Every system tick, Rx lines are
sampled and flagged if we detect a start bit. If detected,
a timestamp is recorded, and the associated receiver task is
started. This only happens if the associated receiver task
is inactive. OSTimer() is called at system tick rate.

Hardware UART activity is handled on an interrupt basis.

**                                                     **
****                                                 ****
*********************************************************/
#pragma interrupt_level 0
void interrupt ISR( void )
{
    ToggleTPInt();

    #if TRANSMITTERS >= 1
    /* when idling, check to see if we've been          */
    /*  signaled by TaskTx().                           */
    if ( txState == TXSTATE_IDLE ) {
        if ( OSTryBinSem(BINSEM_TXSTART_P) ) {
            /* if so, start up state machine and        */
            /*  set PS to minimize delay.               */
            txState++;
            intPS = 1;
        }
    }

    /* we're now transmitting data every fourth         */
    /*  system tick. By using if() if() instead of      */
    /*  if() else we reduce startup delay by 1          */
    /*  system tick.                                    */
    if ( txState != TXSTATE_IDLE ) {
        if ( --intPS == 0 ) {
            intPS = 4;

            /* first, set start bit for each active     */
            /*  transmitter.                            */
            if ( txState == TXSTATE_START ) {

                if ( txStatus.tx1Active )
                    TX1 = 0;

                #if TRANSMITTERS >= 2
                if ( txStatus.tx2Active )
```

```
        TX2 = 0;
    #endif

    #if TRANSMITTERS >= 3
    if ( txStatus.tx3Active )
        TX3 = 0;
    #endif

    #if TRANSMITTERS >= 4
    if ( txStatus.tx4Active )
        TX4 = 0;
    #endif


    ++txState;
}

/* now pump the data out, bit by bit.       */
/*  Do nothing if transmitter is            */
/*  disabled, i.e. has no data to xmit.     */
else if ( txState < TXSTATE_STOP ) {

    if ( txStatus.tx1Active ) {
        if ( tx1Data & 0x01 )
            TX1 = 1;
        else
            TX1 = 0;
        tx1Data >>= 1;
    }

    #if TRANSMITTERS >= 2
    if ( txStatus.tx2Active ) {
        if ( tx2Data & 0x01 )
            TX2 = 1;
        else
            TX2 = 0;
        tx2Data >>= 1;
    }
    #endif

    #if TRANSMITTERS >= 3
    if ( txStatus.tx3Active ) {
        if ( tx3Data & 0x01 )
            TX3 = 1;
        else
            TX3 = 0;
        tx3Data >>= 1;
    }
    #endif

    #if TRANSMITTERS >= 4
    if ( txStatus.tx4Active ) {
        if ( tx4Data & 0x01 )
            TX4 = 1;
        else
            TX4 = 0;
        tx4Data >>= 1;
    }
    #endif

    ++txState;
}

/* now send the stop bit. Can set it        */
/*  regardless of txStatus.txNActive.       */
else if ( txState == TXSTATE_STOP ) {

    TX1 = 1;

    #if TRANSMITTERS >= 2
    TX2 = 1;
    #endif

    #if TRANSMITTERS >= 3
    TX3 = 1;
    #endif

    #if TRANSMITTERS >= 4
    TX4 = 1;
    #endif

    ++txState;
}
```

```
        /* we got here after a full bit delay,      */
        /*  so stop bit is valid -- done.           */
        else if ( txState == TXSTATE_DONE ) {
            txState = TXSTATE_IDLE;
            OSSignalBinSem(BINSEM_TXDONE_P);
        }
    }
}
#endif /* #if TRANSMITTERS >= 1 */


if ( TMR2IE && TMR2IF )    {

    /* clear flag -- req'd.                          */
    TMR2IF = 0;

    /* any start bit activity on RX1? If we haven't */
    /*  detected any already, capture free-running  */
    /*  instruction cycle counter and start task.   */
    /*  If successful (i.e. task was stopped when   */
    /*  OSStartTask() was called) then disable this */
    /*  start-bit detector. It will be re-enabled   */
    /*  when TaskRx1() is finished receiving this   */
    /*  character. If not successful (i.e. task     */
    /*  hadn't stopped yet), then we'll be back in  */
    /*  one system tick to check again. Test RX1    */
    /*  twice to ensure it wasn't noise.            */
    #if RECEIVERS >= 1
    if ( !rxStatus.rx1Active )
        if ( !RX1 && !RX1 ) {
            DisableTMR1();
            rx1Timestamp = TMR1;
            EnableTMR1();
            if ( OSStartTask(TASKRX1_P) == OSNOERR )
                rxStatus.rx1Active = 1;
        }
    #endif

    #if RECEIVERS >= 2
    if ( !rxStatus.rx2Active )
        if ( !RX2 && !RX2 ) {
            DisableTMR1();
            rx2Timestamp = TMR1;
            EnableTMR1();
            if ( OSStartTask(TASKRX2_P) == OSNOERR )
                rxStatus.rx2Active = 1;
        }
    #endif

    #if RECEIVERS >= 3
    if ( !rxStatus.rx3Active )
        if ( !RX3 && !RX3 ) {
            DisableTMR1();
            rx3Timestamp = TMR1;
            EnableTMR1();
            if ( OSStartTask(TASKRX3_P) == OSNOERR )
                rxStatus.rx3Active = 1;
        }
    #endif

    #if RECEIVERS >= 4
    if ( !rxStatus.rx4Active )
        if ( !RX4 && !RX4 ) {
            DisableTMR1();
            rx4Timestamp = TMR1;
            EnableTMR1();
            if ( OSStartTask(TASKRX4_P) == OSNOERR )
                rxStatus.rx4Active = 1;
        }
    #endif

    /* finally, call Salvo's timer since this is    */
    /*  all happening at the system tick rate.      */
    OSTimer();
}

/*                                                    */
if ( RCIF ) {
    rx5Buff[rx5InP++] = RCREG;

    if ( rx5InP > RX5_BUFF_SIZE-1 )
        rx5InP = 0;

    rx5Count++;
```

```
    }

    /*                                                          */
    if ( tx5Count && TRMT ) {
        /* send the valid char, and advance the         */
        /*  pointer.                                    */
        {
            TXREG = tx5Buff[tx5OutP++];

            if ( tx5OutP > TX5_BUFF_SIZE-1 )
                tx5OutP = 0;

            tx5Count--;
        }
    }


    /* this will NEVER occur. It's here because the     */
    /*  Salvo library in use groups many services       */
    /*  together as callable from interrupts, and all   */
    /*  such services are #pragma interrupt_level 0.    */
    /*  This isn't too costly, ROM-wise, so don't       */
    /*  worry about it.                                 */
    if ( TMR1CS ) {
        OSCreateBinSem(BINSEM_TXBUFF_P, 0);
        OSSignalBinSem(BINSEM_TXBUFF_P);
        OSTryBinSem(BINSEM_TXSTART_P);
    }

    ToggleTPIntDone();
}

#endif
```

swrxuart.c

```
/*************************************************************
Copyright (C) 1995-2001 Pumpkin, Inc. and its
Licensor(s). Freely distributable.

$Source: C:\\RCS\\D\\salvo\\demo\\d5\\swrxuart.c,v $
$Author: aek $
$Revision: 1.4 $
$Date: 2001-08-19 21:31:57-07 $

Support file.

*************************************************************/

#ifndef SWRXUART_C_INCLUDES
#define SWRXUART_C_INCLUDES

#include "d5.h"
#include "salvo.h"

/* measured instructions between obtaining desired  */
/*  delay in cycles and when task has been enqueued */
/*  into delay queue. This is an average of the     */
/*  non-interrupted and interrupted case.           */
#define    OVERHEAD            200

/* these are the desired times, in number of        */
/*  instruction cycles from the start bit going low,*/
/*  when we want to sample data.                    */
/* see discussion in BitDelay() below for timing    */
/*  issues.                                         */
#if RECEIVERS >= 1
const unsigned int samplePts[9] = {
    (unsigned int) (1.625*FULL_BIT_CYCLES),
    (unsigned int) (2.625*FULL_BIT_CYCLES),
    (unsigned int) (3.625*FULL_BIT_CYCLES),
    (unsigned int) (4.625*FULL_BIT_CYCLES),
    (unsigned int) (5.625*FULL_BIT_CYCLES),
    (unsigned int) (6.625*FULL_BIT_CYCLES),
    (unsigned int) (7.625*FULL_BIT_CYCLES),
    (unsigned int) (8.625*FULL_BIT_CYCLES),
    (unsigned int) (9.625*FULL_BIT_CYCLES) };


#if ENABLE_TEST_CODE
LOC_TEST_VARS long unsigned int rxErrors = 0;
#endif
#endif

#if RECEIVERS >= 1
_OSLabel(TaskRx1a);
_OSLabel(TaskRx1b);
_OSLabel(TaskRx1c);
_OSLabel(TaskRx1d);
_OSLabel(TaskRx1e);
#endif

#if RECEIVERS >= 2
_OSLabel(TaskRx2a);
_OSLabel(TaskRx2b);
_OSLabel(TaskRx2c);
_OSLabel(TaskRx2d);
_OSLabel(TaskRx2e);
#endif

#if RECEIVERS >= 3
_OSLabel(TaskRx3a);
_OSLabel(TaskRx3b);
_OSLabel(TaskRx3c);
_OSLabel(TaskRx3d);
_OSLabel(TaskRx3e);
#endif

#if RECEIVERS >= 4
_OSLabel(TaskRx4a);
_OSLabel(TaskRx4b);
_OSLabel(TaskRx4c);
_OSLabel(TaskRx4d);
```

```
_OSLabel(TaskRx4e);
#endif


/*********************************************************
****                                                 ****
**                                                     **
BitDelay(timestamp, period)

This is an interesting function. It returns its best-fit
estimate for how many ticks we should delay the receiver
task before sampling the incoming bitstream.

There are several source of error to contend with: 1) system
timer jitter (-1, +0 system ticks), 2) errors resulting from
the quantization of the delay in cycles into system ticks,
3) the number of cycles from when the delay is calculated
to when the task is actually delayed, 4) interrupts and 5)
delays from when the task is made eligible to when it can
actually run.

1) and 2) result in delays that can be too short. 3), 4) and
5) result in delays that can be too long. 1) and 2) are up to
one system tick in size. 3), 4) and 5) are smaller.

By attempting to sample the bitstream one half of one
quarter-cycle after the theoretical mid-time of the data
bit, we hope to increase our chances of sampling between
3/8 and 5/8 into the data bit, i.e. during a one-quarter
bit-time period in the middle of the data bit. Empirical
data suggests this works quite well.


**                                                     **
****                                                 ****
*********************************************************/
#if RECEIVERS >= 1
OStypeDelay BitDelay( unsigned int  timestamp,
                      unsigned char period )
{
    unsigned int tmr;
    unsigned int elapsed;
    unsigned int desired;


    /* capture current instruction cycle count.         */
    DisableTMR1();
    tmr = TMR1;
    EnableTMR1();

    /* calculate instruction cycles elapsed since time- */
    /*  stamp. Deal with wrap as necessary.             */
    /* OVERHEAD is derived from the average number of   */
    /*  instruction cycles from here to exiting the     */
    /*  function, below.                                */
    if ( tmr < timestamp )
        elapsed = tmr + 65536 - timestamp;
    else
        elapsed = tmr - timestamp;

    /* obtain the desired total cycle count (in the     */
    /*  future) for when we want to sample the data     */
    /*  again.  Adjust slightly for the time we're      */
    /*  spending in here.                               */
    desired = samplePts[period] - elapsed - OVERHEAD;

    /* now we know when (in terms of instruction cycles */
    /*  from now) we want to sample data again. Figure  */
    /*  out how many ticks into the future that is.     */
    /* this could use some optimization ... but         */
    /*                                                  */
    /*      return (OStypeDelay) \                      */
    /*     ((4 * desired)/FULL_BIT_CYCLES)              */
    /*                                                  */
    /*  takes more ROM and is likely to be slower.      */
    if      ( desired < (unsigned int) (0.25*FULL_BIT_CYCLES) )
        return 0;
    else if ( desired < (unsigned int) (0.50*FULL_BIT_CYCLES) )
        return 1;
    else if ( desired < (unsigned int) (0.75*FULL_BIT_CYCLES) )
        return 2;
    else if ( desired < (unsigned int) (1.00*FULL_BIT_CYCLES) )
        return 3;
    else if ( desired < (unsigned int) (1.25*FULL_BIT_CYCLES) )
```

```
        return 4;
    else if ( desired < (unsigned int) (1.50*FULL_BIT_CYCLES) )
        return 5;
    else if ( desired < (unsigned int) (1.75*FULL_BIT_CYCLES) )
        return 6;
    else
        /* this catches case of (desired < 0), i.e.     */
        /*  we're too late, as well as (desired > 1 3/4 */
        /*  bit times.                                  */
        return 0xFF;
}
#endif


/*************************************************************
****                                                     ****
**                                                         **
TaskRx1(timestamp, period)

Receiver task. Is normally stopped, wakes up when interrupt-
driven start bit detector sees start bit go low.

Uses BitDelay() to re-synchronize itself to the start bit.
O/wise looks like a normal software UART, with delays
implemented via OS_Delay() instead of looping.


**                                                         **
****                                                     ****
*************************************************************/
#if RECEIVERS >= 1
void TaskRx1( void )
{
    LOC_RX_LOCALS static char c;
    LOC_RX_LOCALS static char i;
    OStypeDelay delay;


    InitBcb(RX1BCBP, RX1_BUFF_SIZE);

    for (;;) {
        /* sample incoming data as soon as the start-bit*/
        /*  detector in the ISR starts this task. We    */
        /*  remain stopped until this happens.          */
        OS_Stop(TaskRx1a);

        /* now that we're here, calculate the right     */
        /*  delay so that when we return we're ready    */
        /*  to sample the next data bit (in this case,  */
        /*  the LSB -- data bit 0).                     */
        delay = BitDelay(rx1Timestamp, 0);

        /* if we're too late then we have to give up.   */
        /*  Re-activate start-bit detector one full     */
        /*  character time from now.                    */
        if ( delay == 0xFF ) {
            OS_Delay(8*FULL_BIT_TICKS, TaskRx1e);
            rxStatus.rx1Active = 0;
            continue;
        }
        /* else we seem to have enough time to delay    */
        /*  and return to sample the data bit.          */
        else {
            ToggleTPRx1SC();

            /* if time is really short, don't delay,    */
            /*  o/wise go ahead and delay.              */
            if ( delay )
                OS_Delay(delay, TaskRx1d);

            /* we're back, ready to begin sampling data */
            /*  bits. c need not be initialized since   */
            /*  all 8 bits will be shifted out.         */
            i = 8;
            rxStatus.rx1Error = FALSE;
            do {
                ToggleTPRx1SC();

                c = (c >> 1) | (RX1 << 7);
                delay = BitDelay(rx1Timestamp, 9-i);
                if ( delay == 0xFF ) {
                    rxStatus.rx1Error = TRUE;
                    break;
                }
```

```
                else if ( delay )
                    OS_Delay(delay, TaskRx1b);
            } while (--i);

            ToggleTPRx1SC();

            /* if the stop bit is still high, we got    */
            /*  valid data.                             */
            if ( RX1 && !rxStatus.rx1Error )
                PutRx1Buff(c);
            #if ENABLE_TEST_CODE
            else
                rxErrors++;
            #endif

            /* done receiving char -- re-enable start   */
            /*  bit detection. No need to control       */
            /*  interrupts, as OSStartTask() can only   */
            /*  start a stopped task, and right now     */
            /*  we're still running.                    */
            rxStatus.rx1Active = 0;
        }

        /* now we're done receiving a char. Return to   */
        /*  top of loop and remain in the stopped state */
        /*  until the start-bit detector wakes us up.   */
    }
}
#endif

#endif
```

N.B. `TaskRx2()` through `TaskRx4()` omitted for brevity.

swtxuart.c

```
/*************************************************************
Copyright (C) 1995-2001 Pumpkin, Inc. and its
Licensor(s). Freely distributable.

$Source: C:\\RCS\\D\\salvo\\demo\\d5\\swtxuart.c,v $
$Author: aek $
$Revision: 1.5 $
$Date: 2001-08-19 21:31:56-07 $

Support file.

*************************************************************/

#ifndef SWTXUART_C_INCLUDES
#define SWTXUART_C_INCLUDES

#include "d5.h"
#include "salvo.h"

#if TRANSMITTERS >= 1
_OSLabel(TaskTxa);
_OSLabel(TaskTxb);
_OSLabel(TaskTxc);
_OSLabel(TaskTxd);
#endif

/*************************************************************
****                                                     ****
**                                                         **
TaskTx()

This is bit-banging transmit task that handles up to four
transmitters at once.

It explicitly takes its data from txNBuff[] and places it in
txNData. It context-switches whenever it has nothing to do.

There's little point in having independent transmitter tasks,
since having the transmitters run synchronously results in a
maximum of one character delay. By placing all four
transmitters in the same task, a considerable RAM and ROM
savings is realized.

**                                                         **
****                                                     ****
*************************************************************/
#if TRANSMITTERS >= 1
void TaskTx( void )
{
    /* Start with all Tx outputs inactive (i.e. idling).*/
    TX1 = 1;

    #if TRANSMITTERS >= 2
    TX2 = 1;
    #endif

    #if TRANSMITTERS >= 3
    TX3 = 1;
    #endif

    #if TRANSMITTERS >= 4
    TX4 = 1;
    #endif

    /* initialize transmitters.                          */
    InitBcb(TX1BCBP, TX1_BUFF_SIZE);

    #if TRANSMITTERS >= 2
    InitBcb(TX2BCBP, TX2_BUFF_SIZE);
    #endif

     #if TRANSMITTERS >= 3
    InitBcb(TX3BCBP, TX3_BUFF_SIZE);
    #endif

     #if TRANSMITTERS >= 4
    InitBcb(TX4BCBP, TX4_BUFF_SIZE);
```

```
    #endif

    txState = 0;

    for (;;) {

        /* nothing happens until the semaphore is       */
        /*  signaled.                                   */
        OS_WaitBinSem(BINSEM_TXBUFF_P, TaskTxa)

        /* get local copy of data to be sent. Activate  */
        /*  transmitter if there's data to send. O/wise */
        /*   deactivate transmitter.                    */
        if ( GetTx1Buff(&tx1Data) )
            txStatus.tx1Active = 1;
        else
            txStatus.tx1Active = 0;

        #if TRANSMITTERS >= 2
        if ( GetTx2Buff(&tx2Data) )
            txStatus.tx2Active = 1;
        else
            txStatus.tx2Active = 0;
        #endif

        #if TRANSMITTERS >= 3
        if ( GetTx3Buff(&tx3Data) )
            txStatus.tx3Active = 1;
        else
            txStatus.tx3Active = 0;
        #endif

        #if TRANSMITTERS >= 4
        if ( GetTx4Buff(&tx4Data) )
            txStatus.tx4Active = 1;
        else
            txStatus.tx4Active = 0;
        #endif

        /* tell ISR data is ready to be shifted out.    */
        OSProtect();
        OSSignalBinSem(BINSEM_TXSTART_P);
        OSUnprotect();

        ToggleTPTx();

        /* wait till ISR is done.                       */
        OS_WaitBinSem(BINSEM_TXDONE_P, TaskTxb);

        /* since we're using only a binsem to indicate  */
        /*  data being available in _any_ buffer, and   */
        /*  since more than a single byte may have been */
        /*  put into any buffer, and since we just got  */
        /*  (i.e. stripped) a maximum of one byte from  */
        /*  each buffer, then buffers may still be non- */
        /*  empty, and we must stay alive until they're */
        /*  completely empty -- all four of 'em.        */
        #if   TRANSMITTERS == 1
        if   ( TX1BCB.count != 0 )
        #elif TRANSMITTERS == 2
        if ( ( TX1BCB.count != 0 ) || ( TX2BCB.count != 0 ) )
        #elif TRANSMITTERS == 3
        if ( ( TX1BCB.count != 0 ) || ( TX2BCB.count != 0 ) \
         ||  ( TX3BCB.count != 0 ) )
        #elif TRANSMITTERS == 4
        if ( ( TX1BCB.count != 0 ) || ( TX2BCB.count != 0 ) \
         ||  ( TX3BCB.count != 0 ) || ( TX4BCB.count != 0 ) )
        #endif
            OSSignalBinSem(BINSEM_TXBUFF_P);
    }
}
#endif

#endif
```

[1] Also called *bit-banged serial comms*.

[2] Applications compiled with Salvo Lite support up to four software transmitters and a maximum of two software receivers.
[3] Do not confuse this with the state of the RS-232 signal line itself, which will typically be between –3 and –12V when idling.
[4] Enhanced software receiver algorithms can look for bit transitions at the expected times, etc. These are not covered here.
[5] E.g. `PutRxNBuff(data)` is a macro invoking `PutBuff(data, bcbP, buffP)`.
[6] Where `N` is 1, 2, 3 or 4.
[7] The timestamp is derived from the free-running 16-bit counter `TMR1`.
[8] E.g. framing errors in the attached receivers.
[9] The hardware receiver and the hardware transmitter.
[10] Isolated resistors, single inline package (SIP), e.g. Panasonic P/N EXB-F8V152G (1.5kΩ each).
[11] U2 on the PICDEM-2 board is a PIC16C73 or equivalent.
[12] The clock speed is divided by four internally to obtain the instruction cycle time, e.g. a 20MHz clock results in 200ns instruction cycles.
[13] The test code can also be configured to run in the scheduler's main loop via a compile-time option.
[14] Hewlett-Packard® 1633A.
[15] LSB first.
[16] It's time that is not spent running tasks, since `TaskTestCode()` – which is always eligible to run – does not run during this time.
[17] Note the different time scale.
[18] Of course, user tasks must also be relatively quick between context switches – a maximum of a few hundred cycles at 20MHz should be observed.
[19] Each binary semaphore can be represented by a single bit. Each Salvo binary semaphore requires three bytes (two for source-code builds with `OSUSE_EVENT_TYPES` set to `FALSE`) for its event control block.