

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

Designing a Low-Cost Multifunction PIC12C509A-based Remote Fan Controller with Salvo

Introduction

SalvoTM, The RTOS that runs in tiny placesTM, is small enough to fit inside the 8-pin Microchip PIC12C509A¹ PICmicro® MCU with its 41 bytes of RAM and up to 1024 instructions in ROM and still leave plenty of room for a full-featured application.

This Application Note provides a detailed explanation of the entire hardware and software design for a multitasking Fan Controller application that

controls and displays fan speed and provides user and remote interfaces, all on a PIC12C509A running Salvo. limited The resources available required creative some solutions while staying within a conventional multitasking



AN-6

Application Note

Figure 1: 8-pin PIC12C509A Fan Controller running Salvo RTOS

framework. How and why they were implemented is presented in detail, with particular attention paid to timing issues.

The Fan Controller's application software (see Listing 6 below) was written entirely in C in under two hours. A portable, battery-powered demonstration version of the Fan Controller is shown in Figure 1 above.

Functional Description

The Fan Controller runs from a DC power source or, optionally, three AA batteries for demonstration purposes. It accepts user input in the form of up- and down-button keypresses. To increase or decrease the fan speed, press the Up or Down keys, respectively. Nine possible fan speed settings are available, indicated on an LED bargraph as no segments lit (fan is OFF) to all eight segments² are lit (fan is at full speed). A beep tone is generated with each keypress for user feedback.

Remote communications are provided over an RS-232 link.³ The Fan Controller reports speed changes due to keypresses via the RS-232 link. It also accepts various commands (see *Command Set* below) and supports software flow control (XON/XOFF). The Fan Controller is connected to a PC / terminal via a null-modem cable.

Several measures are taken to ensure long battery life. After thirty seconds of inactivity the bargraph display is converted to a 1Hz metronome with tick sound. After a total of two minutes of inactivity the Fan Controller automatically shuts off the fan and bargraph and goes to sleep. Either keypress will wake the fan controller, as will activity on the RS-232 receive line. When it wakes up, the Fan Controller restores the fan speed and bargraph display.

Software Organization

The Fan Controller's software must perform the actions listed below, some repetitively. The minimum number of I/O pins required for each action is listed in ().

- Initialization and startup code (0)
- Sleep and wake-up properly (0)
- Detect and debounce keypresses (2)
- Change and maintain fan speed accordingly (1)
- Display fan speed on bargraph (up to 8)
- Beep on keypress (1)
- Transmit characters over RS-232 (1)
- Receive characters over RS-232 (1)

Listing 1: Fan Controller Actions

The ability to create separate tasks in Salvo to perform many of the items above greatly facilitates writing the Fan Controller's software. Normally one would create six tasks to handle the last six actions above and also use Salvo's event services for semaphores and intertask communications⁴ to control program flow. However,

the severely limited RAM of the PIC12C509A requires us to group these items to fit into just three tasks and do without events altogether. Despite these restrictions, it's quite easy to create a fullfeatured application that uses multitasking to its advantage.

Each software component of the Fan Controller application is described in detail below. We will take advantage of Salvo's multitasking abilities as well as its delay services in writing this application.

Note Listing 6 contains the source code described in the sections below. Please see *Design Challenges* below for application-specific information relating to the software and/or hardware.

Variables

Most compilers will automatically initialize variables to 0 at the start of program execution with a small piece of code that executes immediately after reset but before main(). Since we would like the Fan Controller to remember the fan speed setting (and a few other items) when it sleeps, it's necessary to qualify those variables as persistent. Since some variables (e.g. sysStat.xmitOK) require non-zero initial values anyway, we might as well initialize all of them explicitly, and thereby do away with the variable initialization routine that is transparently provided by the compiler – this will save us a few instructions in ROM.

The lack of event services requires that various global flags and semaphores be used. Semaphores like sysStat.beep are set in one place (TaskReadKeys()) and cleared in another (TaskBeep()). The lack of interrupts makes semaphore management very simple. Flags like sysStat.xmitOK are set or cleared in one place (RcvCmds()) and are read in other places (e.g. OutRS232()). speed and sleepTimer are globally visible and are used in various places.

Task Priorities

This application is not hard-real-time, and so task priorities are more a luxury than a necessity. Since there are some ROM savings to be realized when running without priorities, the OSDISABLE_TASK_PRIORITIES configuration options was used. See *Source Code Listings* below for the complete set of configuration options used in this application.

main()

The application's main() is straightforward, with reset detection and initialization happening before a main loop that calls the scheduler and manages sleep. Of note are the means by which a timer is employed, despite the lack of interrupts (see *What? No Interrupts!* below), and the fact that Salvo is initialized and tasks are created only once, immediately before power-on reset (POR). Subsequent wake-on-pin-change resets bypass the initialization and task-creation code. This was done to minimize the system's startup time from sleep.

Locating the Software UART

Since the PIC12C509A has no hardware UART, RS-232 communications must be implemented in software. The lack of interrupts means that the RS-232 receiver must poll the receive data line on a regular basis and analyze the bitstream to decide if it contains valid incoming data. The more often the line can be sampled, and the slower the baud rate, the better the odds of catching the complete transmission. Therefore InRS232() should be called as often as possible. By ensuring that all of the tasks are normally delayed, the odds at any time of one or more tasks needing to be dispatched by the Salvo scheduler OSSched() are very low. By calling InRS232() from within the same loop that holds OSSched(), the time between successive calls to InRS232() will usually be very short, maximizing its chances of "catching" an incoming RS-232 transmission.

The software transmitter OutRS232() does not require any special considerations.

TaskReadKeys()

Since key-reading is a periodic event, this task will use $OS_Delay()$. TaskReadKeys() performs three major functions – i) detecting keypresses and changing the speed setting accordingly, ii) running the metronome when the Fan Controller naps, and iii) implementing the fan speed changes by updating the bargraph and transmitting a character via RS-232.

Each of the three functions runs, one after the other, every SAMPLE_PERIOD ticks, unless a keypress has been detected, in which case the delay will be DEBOUNCE_PERIOD. The metronome runs independently of the other two tasks, i.e. it is not affected by keypresses, etc. Placing it in TaskReadKeys() enables it to run periodically. By reading sysStat.change outside of the keypress-

detecting algorithm, it's possible to force an update to the bargraph and a beep by simply setting this semaphore elsewhere in the program.

TaskSpinFan()

TaskSpinFan() implements a command-driven 8-step software PWM in just 17 lines of C.⁵ Since the fan must spin continuously, TaskSpinFan() monitors the global variable speed and sets the PWM output pin accordingly. Changes to the fan speed take effect in either one tick (if the fan is OFF or fully ON), or as soon as the current PWM period ends if the fan is currently at an intermediate speed.

TaskBeep()

TaskBeep() checks the semaphore sysStat.beep periodically and if set, clears the semaphore and generates a beep tone.

Other Functions

The remaining functions are straightforward. Care was taken to construct program statements (e.g. while() loops) so as to yield the smallest possible code size. InRS232() and OutRS232() have bit-period delays that are optimized for the chosen baud rate.

A Snapshot in Time

Since all three tasks are usually delayed by some number of system ticks, and run for short periods when they're eligible, at any particular instant in time all three tasks are likely to be delayed. This leaves the scheduler with nothing to do. The timer counts down the tasks' delays and makes each one eligible when its delay expires. Therefore most of the time spent in main()'s loop is spent in some very quick trips through OSTimer(), RcvCmd() and OSSched(), with the additional overhead imposed by our Timer0 scheme and the need to monitor for going to sleep.

Command Set

The Fan Controller's commands, and its response to each one, are shown in Listing 2 below.

'?': return current fan speed setting

'0':	turn fan off
'1'-'8':	set fan speed (8 is maximum speed)
'S', 's':	sleep
'T', 't':	sleep in two minutes
'W', 'w':	stay awake indefinitely
XOFF:	stops sending characters
XON:	resumes sending characters

Listing 2: Command Set

Design Challenges

Packing all of the Fan Controller's desired functionality into an 8pin, 1K ROM PICmicro can be challenging. The sections below highlight some of the difficulties (mainly hardware issues) encountered and explain how they were solved through hardware and/or software.

Just Six I/O Pins

The PIC12C509A has only five general-purpose I/O pins and one general-purpose input pin. In order to use all six, the internal oscillator (INTRC) with its fixed 4MHz frequency must be used. Additionally, the six-GPIO-pins configuration precludes the use of an external RESET signal, as well as the Timer0 counter mode.

Six I/O pins are not enough to satisfy the Fan Controller requirement for an eight-segment LED bargraph display. One solution is to add an external serial-to-parallel interface to accommodate the bargraph. Serial-to-parallel interfaces require a minimum of two or three output lines. Thus the Fan Controller requires five output pins (two for bargraph, one for beeper, one for fan and one for RS-232 Tx data) and three input pins (two for keys and one for RS-232 Rx data). Clearly some pins will have to do double-duty as both inputs and outputs.

Not Much Stack

Salvo requires a minimum of four levels of call...return stack. The PIC12C509A has only a two-level call...return stack, which would normally be a problem. Happily, the compiler used⁶ is able to circumvent this limitation and support Salvo by intelligent use of jump tables. This is transparent to the user and does not require any special application coding.

What? No Interrupts!

While it does have a single timer resource – Timer0 (TMR0) – the PIC12C509A has no interrupts. Since it would be nice to take advantage of Salvo's time-based services (delays specified in system ticks, in this case), this means that Salvo's timer needs to be called in an unusual manner. Recall that OSTimer() is normally called every system tick interval – a range of 2-20ms is typical for a 4MHz PICmicro application. Since a periodic interrupt is not available, Timer0 must be used in such a way to call OSTimer() at something approaching a constant system tick rate.

One possible solution would be to have the PIC12C509A sleep, and wake up every system tick. With all six GPIO pins dedicated to I/O and with no external clock source, the only way to achieve this is to use the watchdog timer (WDT) to wake us from sleep. Unfortunately, the WDT period can vary wildly, and perhaps more importantly, is too long (nominally 18ms) to be of use to us – more on why below.

An alternative is to let Timer0 free-run in timer mode, monitor it for rollover and call OSTimer() when this occurs. Therefore the rollover period becomes the system tick period. As long as this period is relatively long compared to the time spent in any other single part of the application, the system tick rate – i.e. the rate at which the application calls OSTimer() – will be relatively constant. This is achieved with the following code snippet, called repeatedly in the infinite for() loop at the end of the Salvo application's main():

```
tmpTMR0 = TMR0;
if ( tmpTMR0 < oldTMR0 )
    OSTimer();
oldTMR0 = tmpTMR0;
Listing 3: Calling OSTimer() Periodically without
    Interrupts
```

With the internal 4MHz oscillator and a 1:16 prescalar assigned to TMR0, OSTimer() is called approximately every 256 * 16 = 4096 instruction cycles. So we have a system tick period of approximately 4ms, or a rate of approximately 250Hz.

As long as successive calls to the snippet in Listing 3 are not more than 4ms apart, the system timer will never lose a tick. Looking through the code, the longest operations (once the application is running) are sending and receiving RS-232 characters. See *Impact of Software UART* below for more information.

Effective and Inexpensive Fan Speed Control

Variable Voltage Drive

At first glance it might seem reasonable to control a 5Vdc fan's speed by varying the voltage applied across it. Since the PIC12C509A has no analog outputs, some sort of digital-to-analog conversion must be employed. The simplest form of D-to-A is a pulse-width modulated (PWM) pulsetrain followed by a low-pass filter. High-frequency PWM signals are preferred as the passive components values (resistors and capacitors) in the low-pass filter are small and therefore inexpensive.

In order for this to work properly, the low-pass filter's cutoff frequency must be sufficiently far below the PWM's frequency to remove high-frequency ripple. Also, we must be able to vary the PWM's duty cycle with sufficient resolution for the number of fan speeds desired. Finally, driving the fan with closed-loop control ensures that the voltage applied across the fan does not vary with power supply fluctuations. Figure 2 depicts an analog fan control circuit with a second-order low-pass filter⁷ first stage followed by a closed-loop fan control output stage. The first stage's output is the weighted the sum of a DC setpoint voltage (3/4 x 5Vdc) and the PWM signal converted to DC (1/4 x 0-5V).



Figure 2: PWM-driven Analog Fan Speed Control Circuit

DC fans usually operate over a range of 75-100% of the design voltage. At lower voltages, the fan may still turn, but it is unlikely to start turning from a dead stop. And below 50% of the design voltage it's unlikely to turn at all. Therefore the useful voltage range is relatively small, and can only control the fan over a limited speed range. Table 1 lists the filtered PWM output of the first stage in Figure 2 for a 30Hz PWM signal as it varies from 0 to 100% duty cycle in 10% steps.

duty cycle	V _{filtered}
(%)	(Vuc)
0	3.80
10	3.89
20	4.00
30	4.12
40	4.20
50	4.27
60	4.38
70	4.51
80	4.59
90	4.63
100	4.73

Table 1 : First Stage Output Voltages for Figure 2

The complexity of the driving circuitry, the extra cost of the required passive and active components and the limited voltage range over which the fan speed can be controlled suggest that this is not the ideal way to build a fan controller.

Direct PWM Drive

Controlling fan speed via direct PWM has many advantages. First, the design voltage is always applied in full across the fan. Second, drive circuitry is considerably simplified. Third, as PWM frequencies of around 30-100Hz are preferred,⁸ high-frequency or hardware PWM is not required. Figure 3 illustrates how to drive a fan directly via PWM.



Figure 3: Direct-PWM-driven Fan Speed Control Circuit

No PWM Output Either?

Since the PIC12C509A has no hardware PWM, let's investigate what it would take to drive a fan via direct PWM in software. Recall that PWM frequencies of 30-100Hz are preferred. For ten steps at 50Hz, we would need a timing resolution of 2ms in order to be able to toggle the PWM output bit at any point (10%, 20%, ..., 90%) of the PWM's waveform, as shown in Figure 4.⁹



Figure 4: 10-step PWM Waveforms at 50Hz

By calling OSTIMER() every 4.096ms and implementing an eightstep PWM, we can achieve similar fan speed resolution at a PWM frequency of 30.5Hz. This means that we can drive the fan speed from within a Salvo task that delays itself by one or more system ticks between writes to the PWM pin. The PWM waveform for a fan speed setting of 6 (75% of full speed) is shown in Figure 5.



It was determined that the fan did not turn reliably at the two lowest fan speed settings (12% and 25% duty cycle). Since the resolution is the system tick period of 4.096ms, the only solution is to increase the minimum ON time by adding extra ON cycles at the beginning of the PWM period. 1 extra cycle (22% duty cycle) was found to be insufficient, but two extra cycles (30% duty cycle) was found to work reliably. The lower PWM frequency of 24.4Hz does not appear to be a problem. The PWM task's output for a fan speed setting of 6 (80% of full speed) is shown in Figure 6 below.



The duty cycles for particular fan speed settings are shown in Table 2 below. The duty cycle can be changed in 10% steps over its useful range.

speed	duty cycle (%)
0	0
1	30%
2	40%
3	50%
4	60%
5	70%
б	80%
7	90%
8	100%

Table 2 : Fan Speed Settings vs. Duty Cycle

The PWM signal's jitter depends on the accuracy of TaskSpinFan()'s delays. Salvo specifies that the accuracy of delays as +/- one system tick. This was observed to be true by sending a continuous stream of '1' commands and observing changes in the PWM period.

Driving the Beeper

Single-tone transducers (beepers) with built-in drive circuits are very simple to use – just take the drive signal high and then take it low the desired time period later. With a system tick every 4ms, a task can easily drive the beeper by driving an output pin high, delaying the desired time, and then driving the output pin low, as shown in Figure 7.



Figure 7: Drive Signal for Beeper with Integrated Driver

Beepers with integrated drivers are noticeably more expensive¹⁰ than those without, so it behooves us to design-in the less expensive variant. Single-tone transducers without integrated drive electronics should be driven with a 50% duty-cycle square wave at the specified frequency. The unit chosen has a frequency of 2048Hz, and the driving signal is shown in Figure 8.



Figure 8: Square Wave for Beeper without Integrated Driver

As with the fan speed control, driving the beeper with PWM in hardware is trivial. But since there is none available on the PIC12C509A, let's investigate how we can drive the beeper in software. Looking at Figure 8, we would have to toggle an output pin every 244 μ s, or every 244 instructions, to drive the beeper at its design frequency. Since the system timer ticks every 4.096ms, using a task to generate this signal via Salvo's delay services is out of the question.

As there are no interrupts available, we would have to code this waveform into the application's main loop. The overhead associated with detecting the passage of 244µs via Timer0 and whether or not the beeper should be beeping (based on the desired duration of the beep after a key is pressed) would consume 20-30 instruction cycles in this 244µs period. With this scheme about ten percent of the PIC12C509A's processing power will be spent managing a relatively unimportant beeper and would prevent us doing any operations that lasted longer than 244µs.

Perhaps more importantly, any scheme that introduces jitter into the beeper's drive waveform around the beeper's design frequency will result in a varying duty cycle. This produces undesirable frequency components and a raspy and unpleasant audio output.

One interesting possibility is to drive the beeper with a single pulse or group of pulses at the desired drive frequency and repeating at the system tick interval of 4ms. This would completely remove high-frequency jitter. It would allow us to place control of the beeper in a task, and enable or disable beeping via a global flag or semaphore. Since the pulse duration is much shorter than a system tick, a conventional delay loop inside the task would be necessary. The driving waveform for the beeper is shown in Figure 9.



Even with additional drive current to make up for the lost energy of this scheme, the resultant beeper output was deemed unacceptable.

Ultimately we're left with no choice but to generate exactly the waveform the beeper requires. In order to avoid jitter we must remain in a tight loop within TaskBeep() while generating a group of equally-spaced 122µs pulses. As we increase the number of pulses, we increase the jitter of the fan PWM, since pulse generation might start just before the Timer0 overflow that marks the expiration of TaskSpinFan()'s delay. A waveform with ten pulses was found to give a pleasant and recognizable "key click"

sound. The final output of the beeper task is shown in Figure 10 below.



Figure 10 : Beeper Waveform as Implemented

Conveniently, the Fan Controller's time to write a new value to the beeper (see *Two Outputs and Two Inputs, One Three-Pin Interface*) is approximately 100 μ s. Therefore back-to-back writes of 1 and 0 result in a waveform that's very close to ideal, but consumes slightly less time.¹¹

Detecting Keypresses

Properly debouncing tact switch keypresses involves sampling and resampling the state of each switch over a time period (e.g. 20ms) that is large relative to a single instruction cycle. Naturally we'd like to do other things while waiting to (re-)sample each switch.

Fortunately Salvo's ability to delay a task by a number of system ticks makes this operation very easy to implement. In pseudocode, the sample-and-debounce algorithm is shown in Listing 4.

```
test for keypress periodically
if key is pressed
delay for the debounce period
if key is still pressed
do key action
```

Listing 4: Key Sample and Debounce Pseudocode

To implement this into a Salvo application the system timer tick duration must be compatible with the delay(s) employed by the algorithm. Testing for keypresses every 20ms works well. Stretching the debounce period to a relatively long 75ms works to our favor as it creates a pleasant key-repeat rate as a side effect. Our choice of 4ms for the system tick fits nicely, as both of these periods are near-integer multiples of the system tick.

Two Outputs and Two Inputs, One Three-Pin Interface

In order to prevent any flickering on the LED bargraph while a new value is serially clocked into it, a latching shift register ('595) was chosen to drive the bargraph. This requires three outputs from the PIC – serial data out, serial clock and strobe. The beeper can be

connected to the shift register's serial output so as not to use another PIC output pin.



Figure 11 : Parallel Expansion Circuit

Of this three-pin serial interface, the only pin that should be dedicated to the interface at all times is the strobe pin. Therefore the serial data out and serial clock pins on the PICCan also function as inputs. After unwanted LOW-to-HIGH transitions on the shift register's serial clock input, new data should be shifted through the shift register in order to maintain the desired level (LOW or HIGH) driving the beeper.

By choosing a latched '595 shift register over an unlatched one (e.g. '164) we can use two PIC12C509A pins as inputs without affecting the bargraph at any time.

Taking Advantage of the Beeper's Slow Response

With a 1 μ s instruction cycle, a byte can be clocked through the shift register in less than 100 μ s. Single updates to the bargraph have little audible effect on the beeper. The beeper can also be set LOW or HIGH without updating the bargraph. Data sent to the beeper due to intentional (e.g. bargraph updates) or unintentional (e.g. serial clock pin on PIC12C509A configured as input and changing) clocking of data through the shift register can be overwritten with a quick blast of serial data out.

Impact of Software UART

Because it is not interrupt-driven, the software UART can affect system timing. Faster baud rates require slower bit-time delay loops, and thereby have less adverse impact. But slower baud rates enable the receiver to better detecting incoming characters. The application is configured for a default baud rate of 2400bps, which was chosen because of its performance in detecting incoming commands.

Optional Baud Rates

The Fan Controller's default baud rate of 2400bps can be overridden to 4800bps or 9600bps by holding the Down or Up keys, respectively, when the unit is first powered on. The selected baud rate will remain in effect until the power is removed.

Timing Issues

At this point it's instructive to take a closer look at some of the timing issues involved, especially because of their impact on the software UART's performance. The PIC12C509A's instruction cycle is 1µs when running from its internal oscillator – this cannot be changed. Most of the time - i.e. when no task delays have expired and there are no eligible tasks to run – the incoming RS-232 data is sampled (via RCvCmd()) every 123µs – this figure was obtained through the MPLAB simulator. In these 123 instructions Salvo manages time services and task scheduling (via OSTimer()) and OSSched(), respectively) and the application calls RcvCmd() and handles the overhead associated with our Timer0 usage and the need to monitor for going to sleep. One might refer to this as the "idle condition". When delays expire and / or eligible tasks need to run, the loop timing will increase significantly as tasks are dispatched, run, and context-switch back to the scheduler. The loop period will return to 123µs as soon as all tasks are again in the delayed state.

This 123µs cycle time defines the upper limit of the responsiveness of the system to incoming RS-232 characters, and its ability to detect the start bit of an incoming RS-232 bitstream. 123µs is well below the bit periods for 1200bps (833µs) and 2400bps (416µs) communications. Since it's less than half their bit periods, RcvCmd() should be able to detect incoming RS-232 data without any difficulties when the system is idling. For 9600bps (104µs bit period) odds are that RcvCmd() will have difficulty picking up the start bit. The software transmitter is of course unaffected by these issues.

While 2400bps and slower baud rates may be advantageous for RS-232 reception, they have the opposite effect on the quality of the PWM signal. Since bit period delays must be generated inside of InRS232() and OutRS232() without the use of interrupts, these delays may affect overall system timing by lasting longer than the

Timer0 overflow period. When this happens, system ticks are lost, and task delays no longer meet Salvo's +/- 1 system tick timer accuracy. Decreasing the system tick period (by changing Timer0's prescalar) only makes this worse, but lengthening the period (say, to 8.192ms) would halve the PWM frequency.

Note that a tight delay loop also occurs within TaskBeep(). It has been configured to be substantially less than a system tick, and hence will not affect system timing.

An instruction cycle of much less than 1μ s used in conjunction with the existing baud rates and PWM frequency would be an ideal solution to the problems above. Since we're stuck with it, we must make choose timing-related parameters wisely.

Circuit Description

The schematic diagram for the PIC12C509A Demo Board is shown in Figure 12. The circuitry is described below.

Battery B1 with three AA cells (nominally +4.5V) supply battery power to the Demo Board. Diode D1 prevents excessive quiescent current draw by regulator U4 when running under battery power. Plugging a DC power source into J1 disconnects B1 and feeds the demo board with regulated DC at one diode drop below +5V, or roughly +4.5V. The Demo Board's can function with the positive supply as high as +5.5V or as low as +3.5V, ensuring long life from a set of three alkaline batteries.

U1 PIC12C509A runs from its internal 4MHz oscillator, thus making all six pins GP[5..0] available for input and/or output. U1 uses its internal reset circuitry (INTERNAL MLCR).

GP[2..0], when configured as outputs, function as serial data, serial clock and strobe to 74HC595 shift register U2. U2's latched outputs feed the upper eight segments of ten-segment LED bargraph D2. U2's serial output (BEEP) is used to directly drive inexpensive single-tone magnetic transducer (beeper) SP1. Diode D3 reduces ground bounce on SP1.

GP4, always configured as an output, drives 5V fan M1 via PNP transistor Q1, necessary because M1's current requirements (130mA) far exceed the output drive of the PICmicro. GP4 is forced LOW (0V) to turn on M1. Since Q1 functions as a saturated transistor switch, nearly the entire +5V supply is available to drive

the fan. Diode D4 protects Q1 from inductive load M1. To turn the fan completely off, GP4 must be taken HIGH (+5V).

GP5, always configured as an output, drives level transceiver U3 with RS-232 transmit data.

GP0, when configured as an input, has weak pull-ups enabled and can wake U1 from sleep when a change occurs. The application can read incoming RS-232 data on this pin. Resistor R2 is provided to isolate U3.1 from GP0 when GP0 is configured as an output.

GP1, when as an input, has weak pull-ups enabled and can wake U1 from sleep when a change occurs. The application can poll the Up key SW1 on this pin. Resistor N2A is provided to isolate SW1 from GP1 when GP1 is configured as an output and SW1 is pressed.

GP3, always configured as an input, has weak pull-ups enabled and can wake U1 from sleep when a change occurs. The application can poll the Down key SW2 on this pin.

RS-232 driver U3 gets logic-side power from +5V, and "steals" – 12V power from the RS-232 receive data line, which is normally idle at -12V. Capacitor C3 serves as a reservoir for U3's V-supply. R4 and C4 form a noise filter to shunt noise on the RS-232 cable's shield to local ground.

5Vdc fan M1 is mounted on standoffs above regulator U4 and other components.

Decoupling capacitors C5-C7, test points TP1-TP11 and mounting holes ZH1-ZH4 are provided. Additional pads Z19-Z20 are provided in case one wants to hook up the remaining LEDs to an on-board signal (e.g. Tx and Rx).

Performance

The Fan Controller software, with various extra bells and whistles like the metronome and the ability to accept upper- and lower-case commands, fits in the PIC12C509A with little room to spare – see *Build Results* below. The Fan Controller itself fits on a 2.400" x 2.400" printed circuit board (PCB) with mixed through-hole and surface-mount components – see *Assembly Drawing (Component Side)* below. It draws a maximum of 160mA¹² (fan at full speed, all bargraph segments lit, RS-232 active), and a minimum of less than 1µA when sleeping.

Commands are received, echoed and processed without errors.¹³ Fan speed can be varied over the entire range in less than a second via the Up and Down keys. Beeper volume is adequate, approximating the "click" sound of a tactile keyboard.

Despite being initialized only at Power-On Reset, Salvo manages its three tasks through sleep and wake-on-pin-change without difficulty.

Enhancements

The size of the application can be reduced somewhat by in-lining Salvo's scheduler and timer – this may also reduce RAM usage. Additionally, Salvo can be configured to use priority arrays instead of priority queues at a substantial savings in ROM size. With priority arrays, the tasks will be prioritized as per the arguments to OSCreateTask() (see main() in Listing 6).

Conclusion

The Fan Controller is a bulletproof, relatively sophisticated application with several time-critical operations. PWM drive, RS-232 transmission and reception, keypress scanning and beeping all occur essentially independent of one another. This is demonstrated by noting that the disabling of any one of these activities has no affect on the others.

Additional embedded programming issue like eliminating unnecessary startup overhead, maximizing RAM utility, using a system timer, software-driven serial communications and PWM, minimizing power consumption and performing multiple functions with single I/O pins are all easily accomplished within Salvo without having to resort to source code changes, assembly language coding or processor-specific extensions.

The clearly-defined behavior of multiple tasks running under the Salvo RTOS makes writing the Fan Controller and similar applications easy. Salvo's scalability enables it to be used in a microcontroller with only 41 bytes of RAM and 1024 program instructions.

Source Code Listings

salvocfg.h

Listing 5 below displays the contents of the salvocfg.h configuration file used to build this project. OSDISABLE TASK PRIORITIES is set to TRUE to reduce ROM requirements, and OSLOC_ALL has the added persistent type qualifier in order to avoid re-initializing the Salvo variables after each reset. The other configuration options are set to typical values.

#define	OSBYTES_OF_DELAYS	1
#define	OSCOMPILER	OSHT_PICC
#define	OSDISABLE_TASK_PRIORITIES	TRUE
#define	OSEVENTS	0
#define	OSLOC_ALL	bank1 persistent
#define	OSTARGET	OSPIC12
#define	OSTASKS	3

Listing 5 : salvocfg.h Configuration File

main.c

Listing 6 below displays the entire Fan Controller application's C source code. Comments have been added to aid in understanding.

```
Copyright (C) 1995-2001 Pumpkin, Inc. and its
Licensor(s). Freely distributable.
$Source: C:\\RCS\\D\\salvo\\demo\\d3\\main.c,v $
$Author: aek $
$Revision: 1.3 $
$Date: 2001-07-28 16:53:16-07 $
Multitasking Salvo-based application using delay
services but no event services due to the limited amount of
ROM and RAM in the host Microchip PIC12C509A PICmicro.
For use on Pumpkin's Salvo PIC12 Demo Board, assembly P/N
710-00197.
See "AN-6 Multitasking PIC12C509A-based Remote Fan
Controller" for more information.
       v1.1aek
                  updated to reflect v2.2 library scheme
   v1.2 aek
               minor edits
   v1.3
          aek
                 v2.3 used less ROM, now all builds
                  have optional baud rates.
*****
#include "salvo.h"
/* detect which target we're compiling for.
#if defined (_12C509) || defined (_12C509A) || defined (_12CR509A)
#define USING_12C50X TRUE
#define BATTERY_OPERATION TRUE
___CONFIG(INTRC | UNPROTECT);
#else
```



#undef USING_12C50X #define BATTERY_OPERATION FALSE #endif /* port pin defs for different targets. Alternate target */ /* is PIC16C77 or equivalent. #ifdef USING_12C50X #define PORT GPIO #define outDATA GP0 #define inRX GP0 #define outCLK GP1 #define keyUP GP1 #define outSTB GP2 #define keyDN GP3 #define outPWM GP4 #define outTX GP5 #define OPTION_CONFIG 0x03 #else #define PORT PORTB #define TRIS TRISB #define outDATA RB0 #define inRX RB0 #define outCLK RB1 #define keyUP RB1 #define outSTB RB2 #define keyDN RB3 #define outPWM RB4 #define outTX RB5 #define OPTION_CONFIG 0x03 #endif /* I/O port configurations for different modes. * / #define GPIO_NORMAL_CONFIG 0x0B /* normal operation */ /* serial writes #define GPIO_SERIAL_CONFIG 0x08 */ /* I/O port default values. */ #define GP0_NO_SERIAL_DATA 0x00 #define GP1_NO_SERIAL_CLK 0x00 #define GP2_NO_SERIAL_STB 0x00 #define GP3_NO_KEY_DOWN 0×08 #define GP4_NO_PWM 0x10 #define GP5 RS232 IDLE 0x20 */ /* Salvo task pointers. OSTCBP(1) #define TASK_READ_KEYS_P #define TASK_SPIN_FAN_P OSTCBP(2) #define TASK BEEP P OSTCBP(3) /* delays based on system tick of 4.096ms. Times are */ /* approximate. #define FOUR_MS
#define TWENTY_MS 1 5 #define FIFTY_MS 12 #define SEVENTY_FIVE_MS 18 #define HUNDRED_MS 24 #define HUNDRED_FIFTY_MS 36 #define ONE_S #define TWENTY_S 2.44 4883 #define THIRTY_S 7324 #define ONE_MIN 14648 #define TWO_MIN 29297 FOUR_MS #define ONE TICK #define TIME_TO_NAP THIRTY S #define TIME TO SLEEP TWO MIN TWENTY MS #define SAMPLE PERIOD #define DEBOUNCE_PERIOD SEVENTY_FIVE_MS #define MINIMUM PERIOD FOUR MS ONE_S / SAMPLE_PERIOD #define NAP_TOCK_PERIOD /* PWM output is active-low. */ #define PWM_ON 0 #define PWM_OFF 1 /* PWM steps and extra periods to ensure that fan spins at */ /* all settings. Extra periods are determined empirically.*/ #define PWM STEPS 8

#define PWM_SIEPS 8 #define PWM_EXTRA_PERIODS 2



PWM_STEPS + PWM_EXTRA_PERIODS

/* duration of pulse train to beeper. */ #define BEEP_PULSES /* pulses * / 10 /* values for the serial data stream to be shifted through */ /* U2 in order to disable or enable the beeper. Since each*/ /* bit affects the beeper, it's imperative to use full-0's*/ /* and full-1's as the patterns, as other patterns will */ /* result in non-50%-duty-cycle waveforms. #define BEEPER_OFF 0x00 #define BEEPER_ON 0xFF /* for OutShiftRegister()'s second argument -- either just */ /* shift the data through U2, or shift and then latch it. */ #define SHIFT_ONLY FALSE #define SHIFT AND LATCH TRUE /* bit times in lus instructions for RS-232 baud rates. * / /* Note that low baud rates will have an adverse effect on*/ the PWM period when there's RS-232 activity, and high */ /* baud rates lead to poor command receiption due to the */ /* narrow sampling window. 2400 baud is default because */ /* it responds best to commands. 4800 is OK, 9600 barely */ /* works. * / #ifndef BAUD #elif BAUD < 1310 /* limit w/char delays */ #error RS-232 baud rate too low. #elif BAUD > 19200
#error RS-232 baud rate too high. #endif #define DLY /* cycles per delay loop */ 3 8 #define OHEAD /* overhead in Tx and Rx */ #define XTAL 4000000 #define ONE_BIT ((XTAL/4/BAUD)-(OHEAD))/DLY #define ONE_BIT_1200 ((XTAL/4/1200) - (OHEAD))/DLY#define ONE_BIT_1200
#define ONE_BIT_2400
#define ONE_BIT_4800 ((XTAL/4/2400)-(OHEAD))/DLY ((XTAL/4/4800)-(OHEAD))/DLY #define ONE_BIT_9600 ((XTAL/4/9600)-(OHEAD))/DLY #define BAUD DEFAULT ONE BIT #define BAUD_KEY_UP ONE_BIT_9600 #define BAUD_KEY_DN ONE_BIT_4800 /* software handshaking / flow control characters. Used to */ /* indicate when we're ready to receive a command. */ /* Ctrl-Q #define XON */ 17 /* Ctrl-S */ #define XOFF 19 /* return codes for InRS232 * / #define NO_RX_CHAR 1 /* none detected
#define BAD_RX_CHAR 2 /* bad stop bit * / /* bit patterns for bargraph LEDs when napping. */ #define TICK_PATTERN 0x7F
#define TOCK_PATTERN 0xBF /* speed of fan at POR start -- show some life. */ б #define FAN_START_SPEED /* function-calling overhead on PIC12 is greater (by 2 */ */ instructions) than in-lining delay functionality, so /* use this macro instead. Downside is that a char /* variable declaration for delay must accompany it ... */ */ /* with full optimizations, each loop iteration takes three*/ /* cycles. * / $#define ShortDelay(a, b) { b = a; while (--b); }$ /* sleep instruction. Don't sleep when debugging via ICE. */ #define Sleep() asm("SLEEP"); #if BATTERY OPERATION #define ClrWakeFlag() { GPWUF = 0; } #define GoToSleep() { Sleep(); }

GPWIIF

#define WokeFromSleep()

#define PWM_PERIOD

*/

*/

*/

* /

*/

* /

* /

*/

*/

*/

*/

* /

*/

* /

*/

*/

*/

*/

* /

* * * *

* *

```
#else
#define ClrWakeFlag()
#define GoToSleep()
                          { SleepHere: goto SleepHere; }
#define WokeFromSleep()
#endif
/* function prototypes.
char InRS232 ( void );
void OutBargraph ( char pattern );
void OutRS232 ( char byte );
void OutShiftRegister ( char byte, char useStrobe );
void RcvCmd ( void );
void TaskBeep ( void );
void TaskReadKeys ( void );
void TaskSpinFan ( void );
/* context-switching labels.
_OSLabel(TaskBeep1)
_OSLabel(TaskReadKeys1)
_OSLabel(TaskReadKeys2)
_OSLabel(TaskSpinFan1)
/* global system status and sysStat byte.
typedef struct {
                :1;
                          /* keypress beep required (sem)*/
  char beep
              :1;
                          /* speed changed (flag)
  char change
                          /* suppress sleeping
 char dontSleep :1;
 char onPWM :1;
                          /* PWM out active, not dc
                          /* OK to transmit to remote
 char xmitOK
               :1;
                         /* for nap display on bargraph */
 char napBit
               :1;
} typeSysStat;
persistent typeSysStat sysStat;
/* fan speed, 0-8. Too expensive (ROM-wise) to have this
/* nibble in sysStat.
persistent char speed;
/* system ticks counter. By declaring it persistent (OK,
/*
  since we always reset it on startup) we're able to rid */
/*
   ourselves of all the startup variable initialization */
/*
  code.
persistent unsigned int sleepTimer;
/* baud-rate-specific delay counter. Used by ShortDelay(). */
persistent char oneBitDelay;
/* REV B has odd pinout, hence the bit juggling.
                                        /* 0 / OFF
const char LEDs[PWM_STEPS+1] = { 0xFF,
                                         /* 1
                               OXEF.
                                         /* 2
                               0xCF,
                                         /* 3
                               0x8F.
                                         /* 4
                               0x0F,
                                         /* 5
                               0x07,
                                         /* 6
                               0x03,
                                          /* 7
                               0x01,
                                          /* 8
                               0x00 };
       *******
****
* *
main()
Typical Salvo main(), with extra code to accomodate
differences between resets, \ensuremath{\mathsf{OSTimer}}(\xspace) and receipt of RS-232
in main loop due to lack of interrupts, and entering sleep.
```

* * * * * * * * ****** void main (void) {

unsigned char oldTMR0, tmpTMR0;

/* Power-on reset (POR) and wake-up-from-sleep (WUF) */

pumpkin



/* bring us here after startup code. No other resets */ /* */ in use. /* setting TRIS and OPTION is always required. */ /* port outputs in normal configuration. */ /* Timer0 in timer mode at fosc/4 w/1:16 prescalar, * / /* free-running. Rolls over every 256 * 16 = 4096 * / /* instruction cycles. * / TRIS = GPIO_NORMAL_CONFIG; OPTION = OPTION_CONFIG; /* wake-on-pin-change can be due to either a keypress or RS-232 Rx activity. The Device Reset Timer (DRT)*/ /* /* period is around 300us for non-POR resets and * / /* varies over operating conditions. This, coupled * / /* with the number of instructions it takes to get * / /* from reset to RcvCmd() actually sampling incoming */ /* RS-232 Rx data, makes processing the command that */ /* woke us up from sleep simply infeasible. Hence * / /* there's nothing to do on wake-on-pin-change. */ $/\,{}^{\star}$ we get here through POR. All these things need be * / /* initialized only once. */ if (!WokeFromSleep()) { $/\,\star\,$ initialize port pin default values and modes. */ /* */ GPO: input Rx data /* GP1: input kev up * / . /* GP2: * / output serial strobe /* GP3: input kev down */ /* GP4: PWM */ output /* GP5: output Tx data * / (GP0_NO_SERIAL_DATA) PORT = (GP1_NO_SERIAL_CLK) (GP2_NO_SERIAL_STB) (GP3_NO_KEY_DOWN) (GP4_NO_PWM) (GP5_RS232_IDLE); /* these variables are initialized only on power-/* up and persist until the system loses power. */ /* .dontSleep: default is to sleep
/* speed: start speed
sysStat.dontSleep = 0; * / * / = FAN START SPEED; speed /* set non-default baud rate if selected. */ if (!kevUP) oneBitDelay = BAUD_KEY_UP; else if (!keyDN) oneBitDelay = BAUD_KEY_DN; else oneBitDelay = BAUD_DEFAULT; /* initialize Salvo. * / */ /* required because Salvo's vars are declared as /* persistent. OSInit(); /* create tasks. For those builds that use task */ */ /* priorities (not really necessary here), /* TaskBeep() must be lowest since it normally */ /* only yields. * / OSCreateTask(TaskReadKeys, TASK_READ_KEYS_P, 1); OSCreateTask(TaskSpinFan, TASK_SPIN_FAN_P, 2); OSCreateTask(TaskBeep, TASK_BEEP_P, 3); } these variables are (re-)initialized on power-up */ /* */ and wake-up. , /* no beeping required */ .beep: /* */ .change: force PWM & bargraph init /* */ .onPWM: PWM output is OFF /* .xmitOK: * / OK to transmit to remote /* sleepTimer: reset this counter sysStat.beep = 0; sysStat.change = 1;



Application Note

```
sysStat.onPWM =
                                 0;
    sysStat.napBit =
                                 0;
    sysStat.xmitOK =
                                 1;
                 = TIME_TO_SLEEP;
    sleepTimer
    /* reset TimerO prior to entering main loop. This
                                                            */
    /* clears the prescalar, too.
    TMR0 = 0;
    /* the usual "infinite for() loop", with extra code to */
    /\,\star\, handle timer and sensing for sleep.
    for (;;) {
        /* if TimerO has rolled over mark it so we can call*/
        /* OSTimer. A temporary placeholder for Timer0 is */
        /* required because it might change value between */
        /* the first and second reads. This is the normal */
        /* way the system keeps track of time.
                                                           */
        tmpTMR0 = TMR0;
        if ( tmpTMR0 < oldTMR0 ) {
            /* call Salvo's Timer to process delayed tasks.*/
            OSTimer();
            /* time-to-sleep countdown timer runs at
                                                            */
                                                            * /
                system tick rate.
            if ( sleepTimer )
                sleepTimer--;
        oldTMR0 = tmpTMR0;
        /* act on command, if present. Note that this has \ */
        /* no deleterious effect on the command that woke */
/* us up from sleep. */
        RcvCmd();
        /* if sleep timer times out, then there's been no
                                                            * /
        /* no user activity for a while, and it's safe
                                                            */
        /\,{}^{\star} to shut things down. Wake up on pin change or
                                                            */
        /* incoming RS-232 data.
                                                            */
        /* GPIO pins are in the GPIO_NORMAL_CONFIG mode,
                                                            */
        /* so no housekeeping is necessary.
                                                            */
        if ( !sleepTimer && !sysStat.dontSleep ) {
                                                           */
            /* turn off fan, bargraph and beeper. No need
            /* to do anything else, since it will all
                                                            * /
            /* be reset on POR/wake-on-pin anyway.
                                                            */
            outPWM = PWM OFF;
            OutBargraph(LEDs[0]);
            /* reset woke-up flag.
                                                            */
            ClrWakeFlag();
            /* dummy read for proper wake-on-change
                                                            */
            /* operation.
            oldTMR0 = PORT;
            /\,\star\, sleepy time - will wake up on pin change.
                                                            */
            GoToSleep();
        }
        /* dispatch most eligible task.
                                                            */
        OSSched();
    }
}
* * * *
                                                          ****
* *
                                                            * *
TaskReadKeys()
Interprets user activity on the two pushbutton keys, and
does key-repeat without acceleration. External action from
RS-232 port is processed here by monitoring sysStat.change
periodically.
* *
                                                            * *
* * * *
                                                          * * * *
```

{

```
void TaskReadKeys ( void )
   static persistent i; /* persistent to avoid init code */
   char tickTockPattern;
   /* intialize this once, when task first runs.
                                                           */
   i = NAP_TOCK_PERIOD;
   for (;;) {
       /* sample keys every 20ms.
                                                           */
       OS_Delay(SAMPLE_PERIOD, TaskReadKeys1);
       /* keypress means user wants to wake system and/or */
          change fan speed.
       if ( !keyUP || !keyDN ) {
            /* wait the debounce period.
                                                           */
           OS_Delay(DEBOUNCE_PERIOD, TaskReadKeys2);
            /* if keyUP is still pressed, then it's valid. */
            /* by not waiting for the key to be released,
                                                           */
            /* we get key-repeat for free!
                                                           */
           if ( !keyUP ) {
                /* do keyUP stuff. force beep, etc. even
                                                           */
                /* if speed doesn't change.
                                                           * /
               sysStat.change = 1;
               if ( speed != PWM_STEPS )
                   speed++;
           }
            /* repeat for keyDN.
                                                           */
           else if ( !keyDN ) {
               sysStat.change = 1;
               if ( speed != 0 )
                   speed--;
           }
       }
       /\,\star time for some fun -- couldn't leave any free
                                                           */
       /* ROM leftover, after all ... do a two-bit
                                                           */
       ,
/*
           metronome at 1Hz when the unit is napping.
                                                           * /
       if ( !sysStat.dontSleep
         && ( sleepTimer <= (TIME_TO_SLEEP - TIME_TO_NAP) ) ) {
              /* countdown timer of SAMPLE_PERIOD.
                                                           */
             if ( --i == 0 ) {
                                                           */
                  /\,\star\, reset timer. Note the error in the
                  /* nap tock period -- it's compounded
                                                           */
                  /* by this multiplier.
                                                           * /
                 i = NAP_TOCK_PERIOD;
                 /* tick or tock on bargraph.
sysStat.napBit ^= 1;
                                                           */
                  if ( sysStat.napBit )
                      tickTockPattern = TICK PATTERN;
                  else
                     tickTockPattern = TOCK PATTERN;
                 OutBargraph(tickTockPattern);
           }
       }
       /\,\star\, make necessary changes if the user requested
                                                           * /
       /* a change in speed or if an external command
/* was received.
                                                           */
       if ( sysStat.change ) {
            /* reset changed flag.
                                                           * /
           sysStat.change =
                                         0;
            /* reset sleep timer.
                                                           */
           sleepTimer = TIME_TO_SLEEP;
                                                           */
            /* request a beep.
           sysStat.beep
                                         1;
            /* show new speed on LED bargraph. Force
                                                           * /
```

/* -MON/BEEP low to enable Rx data monitoring.*/



OutBargraph(LEDs[speed]);

/* now send the speed out via RS-232. */ OutRS232(speed | '0'); } } } **** * * * * * * ** TaskSpinFan() Drives the fan at the current fan setting via a PWM. ++ ++ **** * * * * ***** void TaskSpinFan (void) { OStypeDelay delay; for (;;) { /* speed of 0 means fan is completely OFF. No PWM -*/ /* dc only. Revisit in one system tick. */ if (speed == 0) { ONE_TICK; delay outPWM PWM_OFF; = } /* speed of PWM_STEPS means fan is completely ON. */ /* No PWM - dc only. Revisit in one system tick. */
else if (speed == PWM_STEPS) { delay ONE TICK; = outPWM = PWM ON; } /* intermediate speeds (1 <= speed <= 7)</pre> */ /* require PWM action. */ else { /* If PWM output is OFF, we need to create the */ /* "ON pulse" whose length is directly propor-*/ /* tional to the fan speed. if (!sysStat.onPWM) { delay = PWM_EXTRA_PERIODS + speed; outPWM = PWM_ON; sysStat.onPWM = 1; } /* If PWM output is ON, we need to finish the $\ \ */$ /* PWM period with the output OFF. else { = PWM_STEPS - speed; delav outPWM PWM_OFF; sysStat.onPWM = 0; } } /* PWM port value has been set -- now delay either */ /* 4ms (dc output) or PWM high- or low-cycle. */ OS_Delay(delay, TaskSpinFan1); } } **** * * * * * * * * TaskBeep() Beep by outputting a 4kHz waveform to the beeper for a short time: ____etc. Pulse width is ca. 125us. Checks for the need to beep every system tick, except when it's in the middle of beeping. Yields to more important tasks. NOTE: since no interrupts are used, semaphore management is very simple ...

```
* *
                                                      * *
****
                                                    ****
void TaskBeep ( void )
{
   char j;
   for (;;) {
       /* no services are available for us to wait the
                                                      */
       /* sem, so we have to poll it ...
                                                      */
       OS_Delay(1, TaskBeep1);
       /* is semaphore set?
                                                      */
       if ( sysStat.beep ) {
           /* yes, clear it.
                                                      */
           sysStat.beep = 0;
           /* we're gonna enter a tight loop (below), so
                                                      * /
           /* we won't be able to receive any incoming
                                                      * /
           /* RS-232 data ...
           OutRS232(XOFF);
           /* create BEEP_DURATION pulses, each of
                                                      */
           /* minimum width.
                                                      * /
           j = BEEP_PULSES;
           do {
              OutShiftRegister(BEEPER_ON, SHIFT_ONLY);
              OutShiftRegister(BEEPER_OFF, SHIFT_ONLY);
           } while ( --j );
           /* now that we're done beeping, we can afford \ \ */
           /* to listen for incoming RS-232 again.
                                                      */
           OutRS232(XON);
       }
   }
}
.
* * * *
                                                   * * * *
* *
                                                      * *
OutShiftRegister(byte, useStrobe)
Transfer data to U2 serially and latch it if requested.
Bargraph updates require latching, beeper updates do not.
* *
                                                      * *
****
                                                    * * * *
*****
void OutShiftRegister ( char byte, char useStrobe )
{
   char i;
   /* force data, clock and strobe to be outputs with
                                                      * /
   /* all-zero values. Enable these outputs.
                                                      */
   outDATA = 0;
   outCLK = 0;
outSTB = 0;
   TRIS = GPIO_SERIAL_CONFIG;
   /* clock the 8 bits of the serial byte out GPO.
                                                      */
   i = 8;
   do {
    /* take shift clock LOW.
                                                      */
       outCLK = 0;
       /* send serial data, msb first.
                                                      */
       if ( byte & 0x80 )
           outDATA = 1;
       else
           outDATA = 0;
       /* shift data locally.
                                                      */
       byte <<= 1;
       /* take shift clock HIGH and transfer data
                                                      */
       /* into shift register.
                                                      */
       outCLK = 1;
```

} while (--i); /* transfer the newly-shifted byte to the */ */ latch if requested. if (useStrobe) { outSTB = 1; outSTB = 0; } /* lastly, restore PORTB directions to their /* normal sense. TRIS = GPIO_NORMAL_CONFIG; } /***** , **** **** * * OutBargraph(pattern) Since the need to update the bargraph and keep the beeper off arises in several places, it makes sense to turn this sequence into a function. * * **** ++++ ***** void OutBargraph (char pattern) { OutShiftRegister(pattern, SHIFT_AND_LATCH); OutShiftRegister(BEEPER_OFF, SHIFT_ONLY); } /***** **** * * * * * * * * InRS232() Read a character from the RS-232 port at 2400 baud. Returns RX_NO_CHAR if no activity was detected, and RX BAD CHAR if the incoming char was clearly bad. Adapted from HI-TECH example code. **** **** ****** char InRS232 (void) { char c, i, j; $/ \, \star \,$ first, see if the line is not idle. If so, then * / /* maybe it's a start bit. If it isn't, then it
/* can't possibly be a start bit. */ if (inRX) return NO_RX_CHAR; /* okay, we may have got a start bit. Let's delay half */ /* a bit time so that if we did in fact pick up the */ /* very beginning of the start bit, we won't end up on*/ /* edges of the data later on due to timing */ /* variations. */ ShortDelay((oneBitDelay/2), j); /* Now sample 8 bits of data. No need to initialize c, */ /* since all 8 bits are shifted out anyway. c = 0;i = 8;do { ShortDelay(oneBitDelay, j); c = (c >> 1) | (inRX << 7);
} while (--i);</pre> /* now that we've got the data bits, we must check the */ /* stop bit -- it had better be high. In fact, by */
/* testing the next 9 bit for stop bit, we can avoid */ /* some ambiguities that occur when the next data */ /* follows closely. // /* E.g. '5''5'... is 0 1010 1100 1 0 1010 1100 1 ... */ */ /* start bit in the fifth bit of the transmission.



```
i = 9;
   ShortDelay(oneBitDelay, j);
   do {
       if ( !inRX )
          return BAD_RX_CHAR;
    } while ( --i );
    /* return w/received char.
                                                        */
   return c;
}
* * * *
                                                       * * * *
* *
                                                        ++
OutRS232()
Send a character out the RS-232 port.
Adapted from HI-TECH example code.
* *
                                                        * *
* * * *
                                                      * * * *
*****
void OutRS232 ( char byte )
{
   char i, j;
   /\,{}^{\star} can send chars unless remote system has told us not {}^{\star}/
    /* to.
   if ( sysStat.xmitOK ) {
       /* send start bit. outTX was previously high/idle. */
       outTX = 0;
       /* send data, LSB first, one it at a bit time. */
       i = 8;
       do {
           ShortDelay(oneBitDelay, j);
           if ( byte & 0x01 )
               outTX = 1;
           else
               outTX = 0;
           byte >>= 1;
       } while ( --i );
       /* send stop bit. Line returns to idle condition. \ */
       ShortDelay(oneBitDelay, j);
       outTX = 1;
   }
}
,
* * * *
                                                      * * * *
* *
                                                        * *
RcvCmd()
Get incoming RS-232 data. Echo most of them.
Note that we will be in here for between zero to three
character times depending on what is detected and how we
act on it.
                           report speed
turn fan off
set fan speed to 1-8
go to sleep immediately
sleep when times
                   '?':
'0':
Command list:
                   '1'-'8':
                   'S','s': go to sleep immediately
'T','t': sleep when timer expires
'W','w': stay awake (don't sleep)
* *
* * * *
                                                      ****
void RcvCmd ( void )
{
   char c;
```



Application Note

/* get incoming RS-232 character, if any. */ c = InRS232();/* if it's a speed command, update the speed and force */ /* a speed change -- char will be echoed via update. */ if ((c >= '0') && (c <= '8')) { = c & 0x0F; speed sysStat.change = 1; c = 0; С } /* other commands don't require speed updates -- echo */ /* certain ones. * / else { switch (c) { /* report fan speed. */ case '?': = speed | '0'; break; /* sleep now. */ case 'S': case 's': sysStat.dontSleep = 0; sleepTimer 0; = break; /* sleep TIME_TO_SLEEP from now. */ case 'T': case 't': sysStat.dontSleep = 0; sleepTimer = TIME_TO_SLEEP; break; /* stay awake forever. */ case 'W': case 'w': sysStat.dontSleep = 1; break; /* received XOFF -- stop transmitting. */ case XOFF: sysStat.xmitOK 0; = 0; С = break; /* received XON -- OK to transmit. */ case XON: sysStat.xmitOK 1; = 0; С = break; /* not present, bad or unknown command. */ default: = 0; С break; } } */ /* echo if required. if (c) OutRS232(c);

Listing 6: main.c Source File

}

Build Results

RTOS Size

Examination of the map file¹⁴ shows that the ROM required by Salvo alone is 394 words. This represents the code for initialization, delay, scheduling, timer and assorted utility services.

Application Size

Listing 7 below displays the linker output for project salvo\demo\d3\sysj\d3.pjt when the Fan Controller application is built using the necessary Salvo source files as nodes in a project. All of Salvo's variables (task control blocks, etc.) are in RAM Bank 1. The application's auto and static variables, and the RAM required by the compiler for function argument passing and other purposes, are in Bank 0.

Linking:					
Command line: "C:\HT-PIC\BIN\PICC.EXE -G -INTEL -Md3.map -12C509A -					
oD3.HEX -fake	local -I\salvo\:	include -I	\salvo	olsource	
\salvo\demo\d	3\MAIN.OBJ \sal	vo\source\	DELAY.	.OBJ \salvo\source\MEM.OBJ	
\salvo\source	OINS.OBJ \salv	o\source\U	TIL.OE	BJ \salvo\source\INIT.OBJ	
\salvo\source	INTTASK OBT \	salvo\sour	ce\SCF	HED OBJ	
\salvo\source	TIMER OBT "	04110 (0041	00 (001		
Enter DICC -H	FLD for help				
BIICCI FICC III	The for herb				
Memory Ilsage	Man:				
nemory obuge i	nup.				
Program ROM	\$0000 - \$0002	\$0003 (3)	words	
Program ROM	\$0009 - \$0261	\$0259 (601)	words	
Program ROM	\$028F - \$03FE	\$0170 (368)	words	
Program ROM	\$0FFF - \$0FFF	\$0001 (1)	words	
		\$03CD (973)	words total Program ROM	
Bank 0 RAM	\$0007 - \$0008	\$0002 (2)	bytes	
Bank 0 RAM	\$000B - \$001F	\$0015 (21)	bytes	
		\$0017 (23)	bytes total Bank 0 RAM	
				-	
Bank 1 RAM	\$0030 - \$003F	\$0010 (16)	bytes total Bank 1 RAM	
Build completed successfully.					

Listing 7: Build Results using Salvo Source Files

Listing 8 below displays the linker output for project salvo\demo\d3\sysj\d3lib.pjt when the Fan Controller application is built using a Salvo standard library for the PIC12C509A. Library spl22ld-.lib supports multitasking and delays. The ROM and RAM requirements are identical to those of Listing 7.

Linking: Command line: "C:\HT-PIC\BIN\PICC.EXE -G -INTEL -Md3lib.map -12C509A oD3LIB.HEX -fakelocal -I\salvo\include -I\salvo\source \salvo\demo\d3\MAIN.OBJ D:\SALVO\LIBRARY\SLP211D-.LIB " Enter PICC -HELP for help

Memory Usage Map:

\$0000 - \$0002 Program ROM \$0003 (3) words Program ROM \$0009 - \$0261 \$0259 (601) words \$028F - \$03FE \$0170 (368) words Program ROM Program ROM \$0FFF - \$0FFF \$0001 (1) words \$03CD (973) words total Program ROM Bank 0 RAM \$0007 - \$0008 \$0002 (2) bytes Bank 0 RAM \$000B - \$001F \$0015 (21) bytes 23) bytes total Bank 0 RAM \$0017 (\$0030 - \$003F \$0010 (Bank 1 RAM 16) bytes total Bank 1 RAM Build completed successfully.

Listing 8: Build Results using Salvo Standard Library

Listing 9 displays the linker output for salvo\demo\d3\sysj\d3free.pjt when the Fan Controller application is built using a Salvo freeware library. The extra bounds-checking code contained in the freeware libraries accounts for slightly larger ROM usage.

Linking:					
Command line: "C:\HT-PIC\BIN\PICC EXE -G -INTEL -Md3free map -12C509A					
D3FREE HEX -1	fakelocal -T\s	alvo\includ	e -T\salvo		
\allocalitation load	2\MAIN OPT \GO	lwo\library	\ CED211D_	TTP "	
	S MAIN.OBU (Sa	IVO/IIDIALY	\SFPZIID	LIB	
Enter PICC -H	FTA TOL UEID				
Memory Usage N	Map:				
Program ROM	\$0000 - \$0002	\$0003 (3) word	ls	
Program ROM	\$0004 - \$0261	\$025E (606) word	ls	
Program ROM	\$028D - \$03FE	\$0172 (370) word	ls	
Program ROM	\$0FFF - \$0FFF	\$0001 (1) word	ls	
5		\$03D4 (980) word	ls total	Program ROM
		+ • • (
Bank 0 RAM	\$0007 - \$0008	\$0002 (2) byte	es	
Bank 0 RAM	\$000B - \$001F	\$0015 (21) byte	s	
	+	\$0017 (23) byte	es total	Bank 0 RAM
		4001/ (23, 2700		Dami V Idai
Bank 1 RAM	\$0030 - \$003F	\$0010 (16) byte	es total	Bank 1 RAM
baint 1 iani	40030 40031	\$0010 (10, 2,00		Danni 1 Iani
Build completed successfully.					

Listing 9: Build Results using Salvo Freeware Library

Listing 10 below displays the linker output for salvo\demo\d3\sysa\d3.pjt when the Fan Controller application is built for the midrange PIC16C77 instead of the baseline PIC12C509A.¹⁵ The smaller ROM usage is due to the PIC16C77 having an 8-level call...return stack, which obviates the need for many of the jump tables found in the other builds.



Linking:

Command line: "C:\HT-PIC\BIN\PICC.EXE -G -INTEL -Md3.map -16C77 -oD3.HEX -fakelocal -I\salvo\include -I\salvo\source \salvo\demo\d3\MAIN.OBJ \salvo\source\DELAY.OBJ \salvo\source\INIT.OBJ \salvo\source\MEM.OBJ \salvo\source\QINS.OBJ \salvo\source\SCHED.OBJ \salvo\source\TIMER.OBJ \salvo\source\UTLL.OBJ \salvo\source\INITTASK.OBJ " Enter PICC -HELP for help

Memory Usage Map:

Program R Program R	OM \$0000 OM \$04E4	-	\$0038 \$07FF	\$0039 \$031C \$0355	(((57) 796) 853)	words words words	total	Progra	m ROM
Bank 0 RA Bank 0 RA	M \$0020 M \$0070	-	\$0033 \$0071	\$0014 \$0002 \$0016	(((20) 2) 22)	bytes bytes bytes	total	Bank 0	RAM
Bank 1 RA	M \$00A0	-	\$00B2	\$0013	(19)	bytes	total	Bank 1	RAM
Build completed successfully.										

Listing 10 : PIC16C77 Build Results

Schematic Diagram



Figure 12 below displays the schematic diagram for the version of the Fan Controller.

Figure 12: PIC12 Demo Board Schematic Diagram

Bill of Materials

Listing 11 below is the Bill of Materials for the Fan Controller.

Salvo 00196B	PIC12 Demo .SCH	Board	Revised: April 8, 2001 Revision: B
Pumpki: 750 Naj San Fra (415) www.pum	n, Inc. ples Stree ancisco, C 584-6360 mpkininc.c	t A 94112 om	
Bill O	f Material	s April 8, 20	01 19:42:22 Page 1
Item	Quantity	Reference	Part
1	1	P 1	BH3VY-DC
2			0 1II
2	5		10011 10 10
3	2		1000-10-LP
4	4	D1,D3,D4,D5	1N4148
5	1	D2	HDSP-4830
6	1	H1	DB-9P
7	1	J1	PJ-202B
8	1	M1	FAN-25X10
9	1	Nl	330X9
10	1	N2	1KX3I
11	1	01	2N3906
12	1	R1	47
13	1	R2	4 7K
14	1	P3	10
15	1	RJ PA	1M
16	1	CD1	1 ^M
17	1	SP1 GM2	BRIIZU9P-UI
10	2	SW1, SW2	EVQ-PAC04M
18	11	TP1, TP2, TP3, TP4, TP5, TP6, TP7 TP8 TP9 TP10 TP11	TP
10	1	11	DTC12CE00A = 04/D
20	1	112	74UCE0EN
20	1	02	7460330
21	1	03	DS2765
22	1	04	LM2940CT-5.0
23	4	ZH1, ZH2, ZH3, ZH4	PAD4
24	1	Z1	PCB, Salvo PIC12 Demo Board
25	1	Z2	Socket, 8-pin, DIP
26	3	Z3,Z4,Z5	Battery, AA
27	19	Z6,Z7,Z8,Z9,Z10,Z11,Z12,	PAD
		Z13,Z14,Z15,Z16,Z17,Z18,	
		Z19,Z20,Z21,Z22,Z23,Z24	
28	1	Z25	Screw, 4-40x.250" slotted pan
			head (x5)
29	1	Z26	Screw, 4-40x.500" slotted pan
			head (x4)
30	1	Z27	Washer, 4-40 split-lock (x4)
31	1	Z28	Washer, 4-40 flat (x1)
32	1	Z29	Nut, 4-40 (x1)
32	1	730	Stdoff, 4-40 Hex 0 187"x 250"
55	-	230	(x4)
34	1	Z31	Washer, Special TO-220
			insulating (x1)

Listing 11: Fan Controller Bill of Materials

PCB Plots

Shown below are the assembly and artwork plots for the Fan Controller printed circuit board (PCB).

Assembly Drawing (Component Side)



Figure 13: Assembly Drawing (Component Side)

Artwork (Layer 1 / Top)



Figure 14: Artwork (Layer 1 / Top)

Artwork (Layer 2 / Bottom)



Figure 15: Artwork (Layer 2 / Bottom)

- ¹ Costs substantially less than one dollar in high volumes.
- ² The lowest two segments display RS-232 activity.
- ³ 2400,N,8,1 with software flow control (XON/XOFF).
- ⁴ E.g. via messages.
- ⁵ 41 instructions, including the call to OS_Delay().
- ⁶ HI-TECH PICC v7.86PL3 or later.
- ⁷ Unity-gain Sallen-Key topology, d=1.414 for maximally flat amplitude.
- ⁸ Hanrahan, David, "Fan Speed Control Techniques in PCs," Analog Dialogue, Volume 34, Number 04, June-July, 2000.
- ⁹ Waveforms shown are for a fan circuit that turns the fan on with a logic high (+5V) signal.
- ¹⁰ A difference in price of 2x to 5x is typical.
- ¹¹ Note that alternate values for BEEPER_OFF and BEEPER_ON can be used to drive the beeper at a higher frequency.
- 12 +4.5V supply.
- ¹³ Remember, it's a software, non-interrupt-driven UART, so while you may have to send a character to it a few times before it's accepted, the Fan Controller will not misidentify it. Command reception can be verified by the fact that the Fan Controller echoes each character it successfully receives.
- ¹⁴ salvo\demo\d3\sysj\d3free.map.
- ¹⁵ The salvocfg.h for this project differs only by #define OSTARGET OSPIC16.