PUMPKIN™
REAL-TIME SOFTWARE

Using the Salvo™ RTOS on TI's MSP430

The RTOS that runs in tiny places.™

speaker: Andrew E. Kalman, Ph.D.

Presented at TI's 2nd Annual MSP430 Advanced Technical Conference, Ft. Worth, Texas, Nov. 11-13, 2003

PART I

Introduction to Salvo

The Source – Pumpkin, Inc.

- An embedded solutions company
- Established 1995 in San Francisco, California
- Focused on providing highest-quality tools for embedded system designers
- Active in both hardware and software design for a variety of end-user clients
- Works closely with chip and compiler vendors to assure maximum value for Salvo users

Salvo – A Unique RTOS

- **Minimal on-chip resource requirements**
- **Designed expressly for use in single-chip µC's**
- **Event-driven, priority-based, cooperative multitasking**
- **Certified for use with all major MSP430 compilers:**
- **Available in different flavors:**
  - Salvo Lite — freeware / demo / evaluation
  - Salvo tiny — included with some compilers / IDEs
  - Salvo SE — available from certain compiler vendors
  - Salvo LE — all supported functionality
  - Salvo Pro — Salvo LE + source code
- **Portable (cross-compiler and cross-target)**
- **Highly configurable (written 98% in C)**
- **Easy to learn**
- **Royalty-free**

Who Uses It, and How

- **World-wide user base**
  - Large Corporations
  - Smaller Companies
  - Individual Consultants
  - U.S. Military
  - Educational Institutions
  - Governmental Organizations
- **Applications include:**
  - Military
  - Avionics
  - Recreation
  - Data logging
  - Safety devices
  - GPS equipment
  - Medical devices
  - Handheld devices
  - Industrial / process control
  - Space
  - Telecom
  - Wireless
  - Robotics
  - Food handling
  - Transportation
  - Instrumentation
  - Alternative energy
  - Autonomous vehicles

What's Included

- **Comprehensive user manual (over 500 pages)**
- **Every distribution contains:**
  - Configurable installer
  - Salvo libraries
  - Tutorial and example projects
  - "Getting started" application notes
  - Compiler reference manuals
- **Additional resources for Salvo users:**
  - Responsive tech support
  - Web forums

Compared to other Programming Methodologies

	Foreground / Background	Preemptive RTOS	Cooperative RTOS	Salvo RTOS
Interrupt latency	**low**	moderate	**low**	**low**
Interrupt response	**low**	high	**low**	**low**
Task response	low	**fast**	moderate	moderate
Stack requirements	**low**	high	moderate	**low**
RAM requirements	**varies**	high	moderate	**low**
ROM requirements	user	high	moderate	moderate
Intrusiveness	user	high	moderate	**low**
Coupling	tight	**loose**	**loose**	**loose**
Extensibility	poor	**excellent**	**excellent**	**excellent**
Handles complexity	poorly	**well**	**well**	**well**
Effort to learn	**least**	most	more	more

The RTOS Approach to System Software

Features:

- Loosely-coupled: Each task can run independently of others
- Priority-based: The highest-priority, eligible task is always running, or will run as soon as the current task yields (i.e. context-switches)
- Event-driven: While a task is waiting, delayed or stopped, no processing (i.e. 0 CPU cycles) is expended in "maintaining" the task
- Inter-task Communications: Distributed program execution based on task-to-task or ISR-to-task actions

Benefits:

- Loose Coupling: Adding and / or removing tasks from the application – even during runtime – is very simple. Application features can be easily compartmentalized, enabled, tested, etc. Especially beneficial where multiple programmers are involved in creating a single, large application.
- Priority-based Task Execution: Important, time-critical tasks get CPU resources when they need them. Less-important, "do-whenever" tasks get the CPU only when appropriate.
- Event-driven Behavior: System exhibits excellent overall system responsiveness, because *there is no polling*. CPU resources are always directed towards the highest-priority eligible task. System is always "primed, waiting for an event" and can sleep between events.
- Inter-task Communications: Connect loosely-coupled processes in a well-defined manner.

Features and Operational Details

Tasks:

- 16 dynamic task priority levels
- "Run forever" task structure
- Tasks can be created, started, stopped, destroyed, etc.
- A context switch *always* results in the most-eligible task running
- Constraints:
  - Context switch may only occur at the task level
  - A tasks' local / auto variables are usually replaced with `static` variables[i]

Events:

- Binary semaphores, semaphores, messages, message queues and event flags are supported
- Events can be created and signaled from anywhere. Tasks can wait events (with optional timeouts)

Timers:

- Single system timer controls all task delays and timeouts, as well as system tick services
- `OSTimer()` can be called from any periodic timer

MSP430 Port

Memory Requirements

- RAM usage per task control block: 14 bytes max.[ii]
- RAM usage per event control block: 6 bytes max.[iii]
- Stack size: Similar to typical foreground / background application
- ROM usage: 400-1700 bytes

tutorial memory usage[iv]	total ROM[v]	total RAM[vi]
`tu1lite`	450	22
`tu2lite`	596	22
`tu3lite`	638	24
`tu4lite`	1148	34
`tu5lite`	1562	50
`tu6lite`	1678[vii]	52[viii]
`tu6pro`	1550[ix]	48[x]

Table 1: ROM and RAM requirements for Salvo Applications built with IAR's MSP430 C Compiler

Context Switching

25µs @ MCLK = 8MHz (with priorities, events, etc.)[xi]

Interrupt Control

- Default configuration is for `GIE` to be disabled during critical sections
- Interrupt latency can be minimized via user (re-)configuration of interrupt control[xii]

MSP430 Real-world Results

Suitability

- MSP430's 2K RAM and 60K ROM are ideal for Salvo applications – 20-task, 30-event application consumes under 15% RAM and 5% ROM, leaving plenty of RAM and ROM for user application
- Salvo runs on *every* member of the MSP430 family

Low Power

- Salvo's event-driven multitasking allows application to sleep at all times, waking only for activity (i.e. internal or external events)

Performance

- MSP430's highly orthogonal instruction set and comprehensive addressing modes mean rapid execution of Salvo services

Tools

- Non-intrusive, easy to debug
- Works seamlessly with all major toolsets
- Pumpkin and MSP430 compiler vendors are actively involved in further integrating Salvo into their toolsets

Conclusion

- **Using Salvo on the MSP430 helps the embedded designer in:**
  - Implementing new designs quickly
  - Enhancing functionality using existing on-chip resources
  - Improving real-time performance
  - Multitasking
  - Using memory efficiently
  - Minimizing costs
  - Managing complexity
  - Reducing time-to-market

"… let me say that the RTOS has exceeded all of our expectations and we are grateful for your excellent support."

Mark Mayernick
Salvo + MSP430 user
Datex-Ohmeda

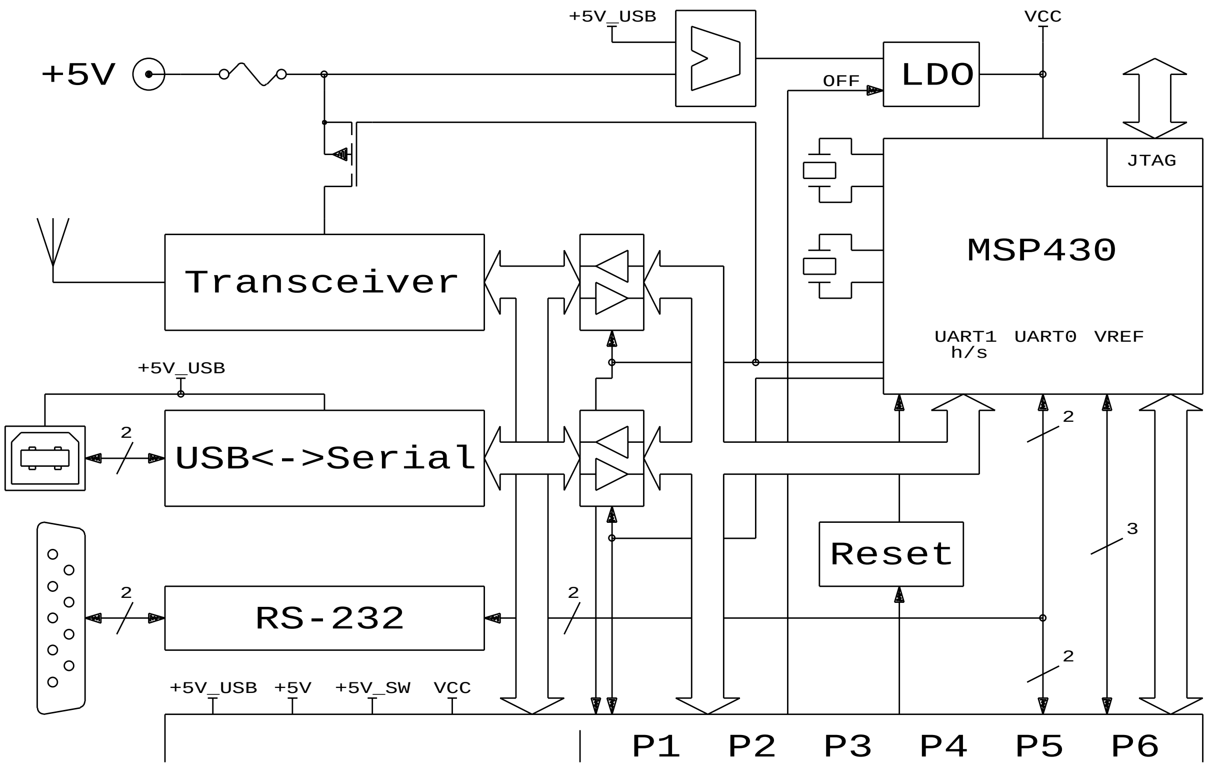
PART II

Example Salvo Application

MSP430F149-based Design Example

- **Hardware Details:**
  - P6 shared between:
    - USB / transceiver handshake / control interface
    - Transceiver isolation
    - Analog sampling channels (e.g temp sensors)
  - USART1 shared between:
    - Serial-to-USB converter
    - 2.4GHz spread-spectrum wireless transceiver
    - User (off-board)
  - Mixed +3.3V / +5V design:
    - Level translators & buffers provide isolation, incl. unpowered states
    - USB (+5V, bus-powered[xiii]) interfaces to MCU at +3.3V via isolator
    - Transceiver (+5V) interfaces to MCU via isolators & level-shifters
    - MCU controls –0E's on isolators & level-shifters
    - MCU controls power to +5V transceiver
  - Low-Power:
    - Sleep at < 30µA,[xiv] operate at < 2mA, Tx (occasionally) at > 750mA
    - Powered via internal +5V or via USB

● Block Diagram:



- **Software Requirements:**
  - USART1:
    - Manage isolation & interface to USB / transceiver / user to avoid contention
  - USB:
    - Detect when USB I/F is present
    - Acquire & release USB interface
  - Transceiver:
    - Tx / Rx when requested
    - Acquire & release transceiver interface, including the control of transceiver power when Tx'ing / Rx'ing
  - P6:
    - Sample at a variety of rates via A/D inputs
    - Handshake control to USB / transceiver interface
  - Other Processes:
    - Perform a myriad of other simultaneous operations (e.g. data processing, system status reporting, storing and retrieving data to / from external NVRAM, etc.)
  - Power Consumption:
    - Sleep whenever no activity is warranted

Task to Read Ambient Temperature

Configure ADC12 and read internal temperature sensor at 1/2Hz

```
unsigned int ADCresult;
unsigned long int DegC;
…
void TaskMeasureAmbientTemp( void )
{
  /* setup ADC12 to read ch 10, etc. */
  ADC12CTL0  = ADC12ON+REFON+REF2_5V+SHT0_6;
  ADC12CTL1  = SHP;
  ADC12MCTL0 = INCH_10+SREF_1;

  /* wait 10ms for reference startup */
  OS_Delay(1, label);

  /* enable conversions */
  ADC12CTL0 |= ENC;

  for (;;)
  {
    ADC12CTL0 |= ADC12SC;   // start conversion
    OS_Delay(200, label);   // wait 2s
    ADCresult = ADC12MEM0;  // read result
    DegC = ((((long)ADCresult-1615)*704)/4095); // calc. DegC
  }
}
```

TaskMeasureAmbientTemp()

- **Attributes:**
  - Runs independently of others, i.e. loosely-coupled.
  - Runs at a low priority. Ambient temp sensing is not a high-priority issue in this system. OK if other, higher-priority tasks prevent it from running immediately after its 2s delay expires.
  - Uses minimal run-time resources. During the 2s period between successive reads of `ADC12MEM0`, no CPU cycles are expended on `TaskMeasureAmbientTemp()`, and other tasks are free to run.
  - No inter-task communications, because it runs alone, accessing global variables.
- **Additional Features:**
  - Salvo's ability to context-switch at any place in the task allows other tasks to run while `TaskMeasureAmbientTemp()` is waiting for 10ms delay during ADC12 initialization.

Task to Detect if USB is Connected

Check for USB every 250ms, signal system if present

```
void TaskDetectUSB( void )
{
  for (;;)
  {
    /* proceed if USB/MHX I/F is not in use */
    OS_WaitBinSem(BINSEM_USB_MHX_AVAIL_P, OSNO_TIMEOUT, label);
    OpenUSBMHXIF(USB);

    if ( !FM430status.USBpresent && (P1IN & BIT7) )
    {
      FM430status.USBpresent = 1;
      FM430Msg0("DetectUSB: USB connected.");
    }
    else if ( FM430status.USBpresent && !(P1IN & BIT7) )
    {
      FM430status.USBpresent = 0;
      FM430Msg0("DetectUSB: USB disconnected.");
    }

    /* release USB/MHX I/F */
    CloseUSBMHXIF(USB);
    OSSignalBinSem(BINSEM_USB_MHX_AVAIL_P);

    /* come back in 25 ticks */
    OS_Delay(25, label);
  }
}
```

TaskDetectUSB()

- **Attributes:**
  - Runs independently of others, i.e. loosely-coupled.
  - Runs at a moderate priority. System should detect USB connections quickly.
  - Uses minimal run-time resources. During the 250ms period between testing for USB presence, no CPU cycles are expended on `TaskDetectUSB()`, and other tasks are free to run.
  - A binary semaphore is used to control access to a shared resource, the USB / transceiver interface.
- **Additional Features:**
  - `TaskDetectUSB()` will be "held off" until the USB / transceiver interface is available. If the interface is not available (i.e. another task is using it), `TaskDetectUSB()` will acquire it when the interface is released and `TaskDetectUSB()` is the highest-priority task waiting to use the interface.

Task to Enable Transceiver Power During Transmission

When Interface is Available, Turn on Transceiver for 5s

```
void TaskTalkMHX( void )
{
  for (;;)
  {
    /* proceed if USB/MHX I/F is not in use */
    OS_WaitBinSem(BINSEM_USB_MHX_AVAIL_P, OSNO_TIMEOUT, label);
    OpenUSBMHXIF(MHX);

    /* turn on power to transceiver */
    Enable_5V_to_MHX();

    /* leave it on for 5s (length of transmission) */
    OS_Delay(500, label);

    /* release USB/MHX I/F */
    CloseUSBMHXIF(MHX);
    OSSignalBinSem(BINSEM_USB_MHX_AVAIL_P);
  }
}
```

TaskTalkMHX ()

- **Attributes:**
    - Runs independently of others, i.e. loosely-coupled.
    - Runs at a moderate priority.
    - Uses minimal run-time resources. During the 5s period that transceiver power is on, no CPU cycles are expended on `TaskTalkMHX()`, and other tasks are free to run.
    - A binary semaphore is used to control access to a shared resource, the USB / transceiver interface.
- **Additional Features:**
    - Like `TaskDetectUSB()`, `TaskTalkMHX()` must acquire the USB / transceiver interface before proceeding, etc.
    - During the 5s period when `TaskTalkMHX()` has acquired the USB / transceiver interface, all other tasks wishing to use the interface must wait.
    - `TaskTalkMHX()` is incomplete. It would likely be expanded to wait on an event that signifies that data is ready to be transmitted. After transceiver power is enabled and the transceiver has completed its power-on sequence, `TaskTalkMHX()` could signal another task to begin transmitting data (packet-wise) over the USB / transceiver interface. When finished, `TaskTalkMHX()` would receive a signal to power-down the transceiver and release the USB / transceiver interface, and resume waiting for a transmit-data event.

Entering and Exiting Low-Power Modes

Sleep whenever there are no eligible tasks

```
void OSIdlingHook (void)
{
  __low_power_mode_1();
}
```

`OSIdlingHook()` is called only when no tasks are eligible to run. Therefore it's the ideal place to sleep the processor, until an event (i.e an internal or external interrupt) occurs.

Exit LPM after each interrupt that calls a Salvo service

```
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  CCR0 += 10000;
  OSTimer();
  __low_power_mode_off_on_exit();
}
```

Don't re-enter LPM until Salvo's scheduler has processed event(s). ISRs that are independent of Salvo can resume LPM on exit.

Putting it All Together

Initialize, create tasks and events, begin multitasking

```
void main (void)
{
  /* user init */
  Init();

  /* Salvo init */
  OSInit();

  /* several interrupts are used */
  __enable_interrupt();

  /* create tasks */
  OSCreateTask(TaskStatusMonitor,      OSTCBP(1),  3);
  OSCreateTask(TaskDetectUSB,          OSTCBP(2),  8);
  OSCreateTask(TaskTalkUSB,            OSTCBP(3),  5);
  OSCreateTask(TaskTalkMHX,            OSTCBP(4),  7);
  OSCreateTask(TaskMeasureAmbientTemp, OSTCBP(5), 11);
  …
  /* create events */
  OSCreateBinSem(BINSEM_USB_MHX_AVAIL_P, 1);

  /* go */
  for (;;)
  {
    OSSched();
  }
}
```

Completing the Application

- **Use additional binary semaphores and task priorities to manage access to resources:**
  - Analog sampling tasks wait for P6 (shared with USB / transceiver interface) to be available before proceeding
  - User USART1 task waits for USART1 (used by `TaskTalkUSB()` and `TaskTalkMHX()` to be available before proceeding
- **Run additional periodic tasks at multiples of system tick period**
- **Use messages and message queues for intertask communications:**
  - Multiple, asynchronous sampling tasks pass data to a single task that logs captured data to NVRAM
  - Highest-priority tasks wait on critical events
- **Use free-running system timer for timestamps**
- **Handle lost events via wait-with-timeout**

Example Application Results

- **Application Configured For / Uses:**
  - 10ms system tick period
  - Multiple interrupt sources
  - LPM1
  - MCLK, SMCLK
  - `sprintf()`, 16-bit multiply & divide
  - Subsystems:
    - Timer_A, USART0, USART1, ADC12, WDT, Digital I/O
- **Salvo Configured For:**
  - 16-bit delays
  - Priority-based multitasking
  - Binary semaphores
  - 15 tasks
  - 32-bit system timer
  - 1 event
- **Salvo's Memory Requirements[xv] on MSP430F149 for this Application:**
  - 1132 bytes ROM (1.8%) for Salvo services
  - 171 bytes RAM (8.3%) for Salvo's global objects
  - Default of 90 bytes RAM (4.4%) for stack is more than sufficient
- **Application's Power Consumption:**
  - Over 97% of the time in LPM

Example Application Runtime Behavior

USART0 sending debug information via RS-232:

```
…
FM430-Tx0 0000005451 $ TalkUSB: Acquired USB/MHX I/F for USB.
FM430-Tx0 0000005458 $ TalkUSB: Sending to USB.
FM430-Tx0 0000005459 $ TalkUSB: Released USB/MHX I/F.
FM430-Tx0 0000005535 $ DetectUSB: USB disconnected.
FM430-Tx0 0000005860 $ DetectUSB: USB connected.
FM430-Tx0 0000005923 $ TalkUSB: Acquired USB/MHX I/F for USB.
FM430-Tx0 0000005930 $ TalkUSB: Sending to USB.
FM430-Tx0 0000005931 $ TalkUSB: Released USB/MHX I/F.
FM430-Tx0 0000005982 $ TalkMHX: Acquired USB/MHX I/F for MHX.
FM430-Tx0 0000005983 $ TalkMHX: +5V_SW is ON.
FM430-Tx0 0000006482 $ TalkMHX: +5V_SW is OFF.
FM430-Tx0 0000006483 $ TalkMHX: Released USB/MHX I/F.
…
```

USART1 sending ambient temp information via USB:

```
…
FM430-Tx1 0000004587 $ Ambient temp: 19 C
FM430-Tx1 0000004687 $ Ambient temp: 19 C
FM430-Tx1 0000004787 $ Ambient temp: 19 C
FM430-Tx1 0000004887 $ Ambient temp: 20 C
FM430-Tx1 0000005452 $ Ambient temp: 20 C
[USB disconnected]
…
[USB re-connected]
FM430-Tx1 0000005924 $ Ambient temp: 20 C
…
```

Thank you for your interest in

Salvo™

The RTOS that runs in tiny places.™

750 Naples Street

San Francisco, CA 94112

USA

tel: (415) 584-6360

fax: (415) 585-7948

web: http://www.pumpkininc.com/

email: info@pumpkininc.com

Speaker Information

Dr. Kalman is Pumpkin's president and chief software architect. He entered the embedded programming world in the mid-1980's. After co-founding a successful Silicon Valley high-tech startup, he founded Pumpkin with an emphasis on software quality. He has also been involved in a variety of other hardware and software projects.

Copyright Notice

---

i Local / auto variables are not preserved across context switches. Note that the use of using static variables in tasks does not impact overall RAM requirements when compared to a typical preemptive or cooperative RTOS.
ii Disabling timeouts reduces tcb size to 10 bytes. Optional tcb extensions (Salvo Pro only) require additional RAM per tcb.
iii Can be reduced to 4 bytes by disabling event types.
iv Salvo v3.2.0-b with IAR MSP430 C v1.26B.
v In bytes. Does not include interrupt vectors.
vi In bytes. Does not include RAM allocated to the stack.
vii Includes 2 bytes from the CDATA0 section.
viii Includes 2 bytes on the IDATA0 section.
ix Includes 2 bytes from the CDATA0 section.
x Includes 2 bytes on the IDATA0 section.
xi As measured with tu4lite.
xii Salvo Pro only.
xiii A bus-powered USB device is one that gets its power from the USB host (i.e. over the USB cable).
xiv This is the total system sleep current, and includes the quiescent current of voltage regulators, leakage across power-control and level-shifting FETs, etc.
xv IAR MSP430 C v2.10A