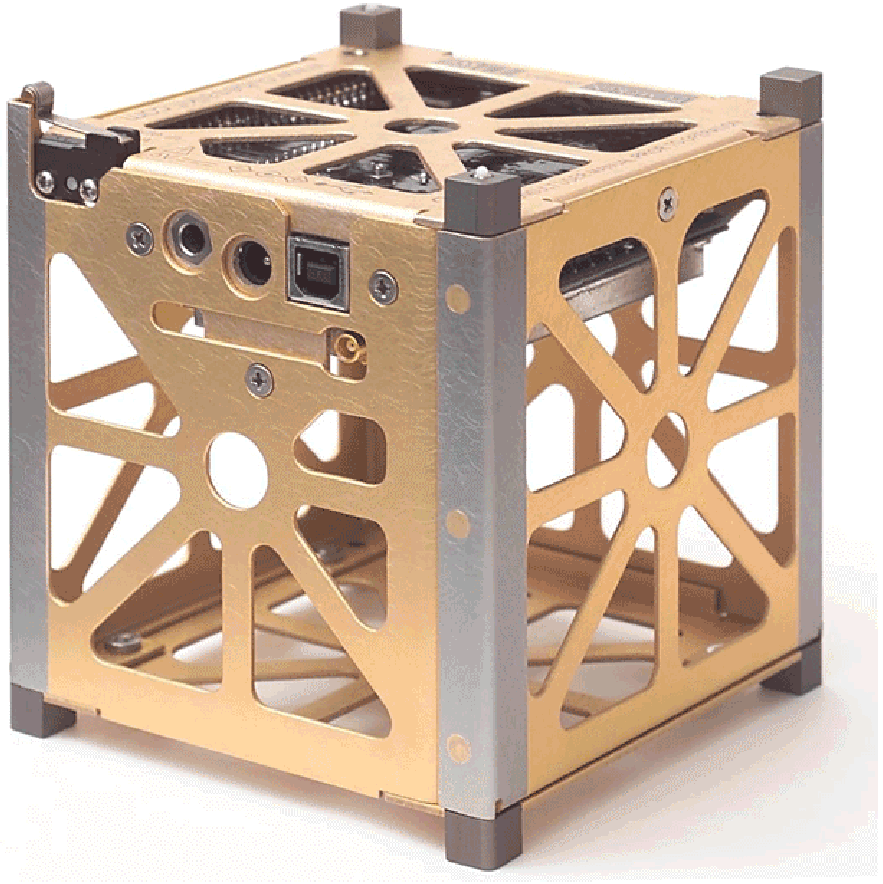
Hardware and Software Design of an MSP430-based Satellite using an RTOS



speaker:

Andrew E. Kalman, Ph.D.
President, Pumpkin, Inc.

Part I

Hardware & Software Requirements for a Small Satellite Mission

Nano- and picosatellites – The future of space?

* Small, low-cost satellites built using commercial off-the-shelf (COTS) components are currently a hot topic. Commercial, educational and military organizations have expressed strong interest in picosatellite missions.

CubeSat as an affordable standard for picosatellites

* The CubeSat is a 10x10x10cm, 1kg open standard proposed by Stanford and Cal Poly San Luis Obispo universities for picosatellites. The space-qualified P-POD CubeSat launcher currently exists, and more launch platforms are planned.

* CubeSats carry payloads for relatively simple missions (e.g. space-qualification or behavioral demonstration).

* Low-earth orbit (LEO) CubeSat missions have typical lifespan of 3-9 months.

* Working from a standard promotes rapid development and idea sharing.

* CubeSat standard includes requirements for safety equipment to satisfy launch providers (e.g. NASA, Russians, etc.).

* Cost to complete a CubeSat mission (inception to launch to operation to end-of-life) ranges from $100,000 to $500,000.

Picosatellite mission environments

* Development, debugging & functional testing: typical lab environment.
* Pre-delivery and pre-launch: temp/vac and shake tests, picosat may remain in storage for months on end waiting for launch.
* Launch & deployment: high g-forces (10g or more).
* Operation in space: vacuum, temperature extremes (-20º to +60º C), solar radiation, and remoteness – just 1W average power from Sun.
* End of mission: burn up in earth's atmosphere.

Picosatellite components

* Structure
* Command & Data Handling (C&DH), with high-frequency transceiver and antenna
* Communications (COM)
* Electrical Power Systems (EPS)
* Attitude Determination & Control (AD&C)
* Payload
* Software, Software, Software

A COTS CubeSat kit should:

* Implement as many of the CubeSat standard's specifications as possible, thereby shortening the design & development timeline.

* Integrate as much functionality into as small a space as possible, so as to leave plenty of room for the CubeSat's payload and other mission-specific hardware.

* Be flexible, with a design that can accommodate a wide range of customer requirements and configurations.

* Be expandable and modular, to accept other COTS hardware and software.

* Provide a software development platform that's easy to use and debug.

* Have a clearly-defined future upgrade path.

* Include high-value-added features (hardware and software) to differentiate itself from the competition and "home-grown" CubeSats.

* Be affordable.

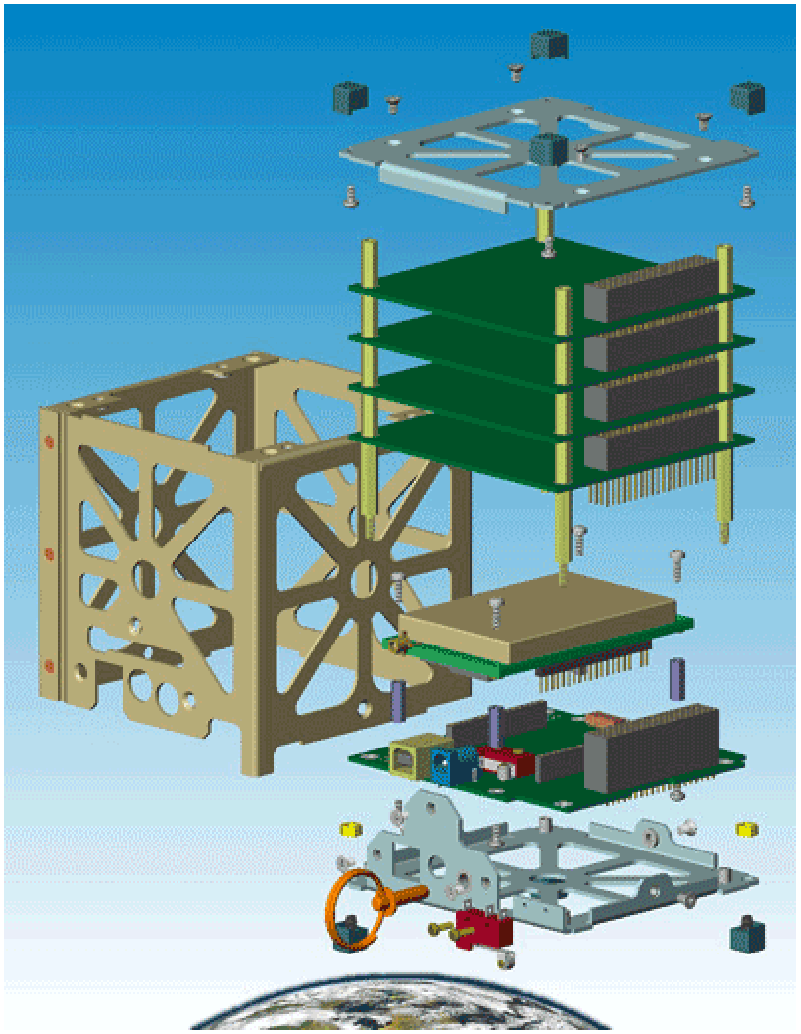


Figure 1: Exploded View of Pumpkin's MSP430-based CubeSat Kit.

MSP430 resides on FM430 Flight Module (PCB at bottom).

Part II

Why Pumpkin Chose TI's MSP430 for the

CUBE SAT KIT™

MSP430's CubeSat Kit-friendly features include:

* -40º to +85º C operating temperature range.
* Relatively large code space (60K in MSP430F149) enables a high level of systems integration, thereby reducing parts count.
* Highly orthogonal and C-friendly architecture makes for efficient high-level programming.
* Good mix of on-board peripherals. Plenty of I/O.
* JTAG & Flash accelerate the design & debug cycle.
* Ultra-low-power design, dual clock sources and fast startup from sleep help minimize power consumption.
* Wide range of tools (assemblers, compilers, RTOS, IDEs, JTAG debuggers, programmers, development kits, etc.) available.

The CubeSat Kit's MSP430 Flight MCU is responsible for:

* Doing supervisory tasks, e.g. monitor in-satellite temperatures, voltages and currents, etc.
* Managing bidirectional communications with the mission's ground station(s) (COM).
* Acting autonomously and/or through commands from the ground station(s) or other satellites (C&DH).
* Handling data storage and retrieval, either on-chip (in RAM or Flash) or off-chip (mass storage).
* Configuring its I/O to interface to multiple peripherals that share the same MSP430 port pins.
* Depending on system complexity, EPS and AD&C functions may be implemented either in the MSP430 or in additional mission-specific CPUs and µC's that communicate with the MSP430.

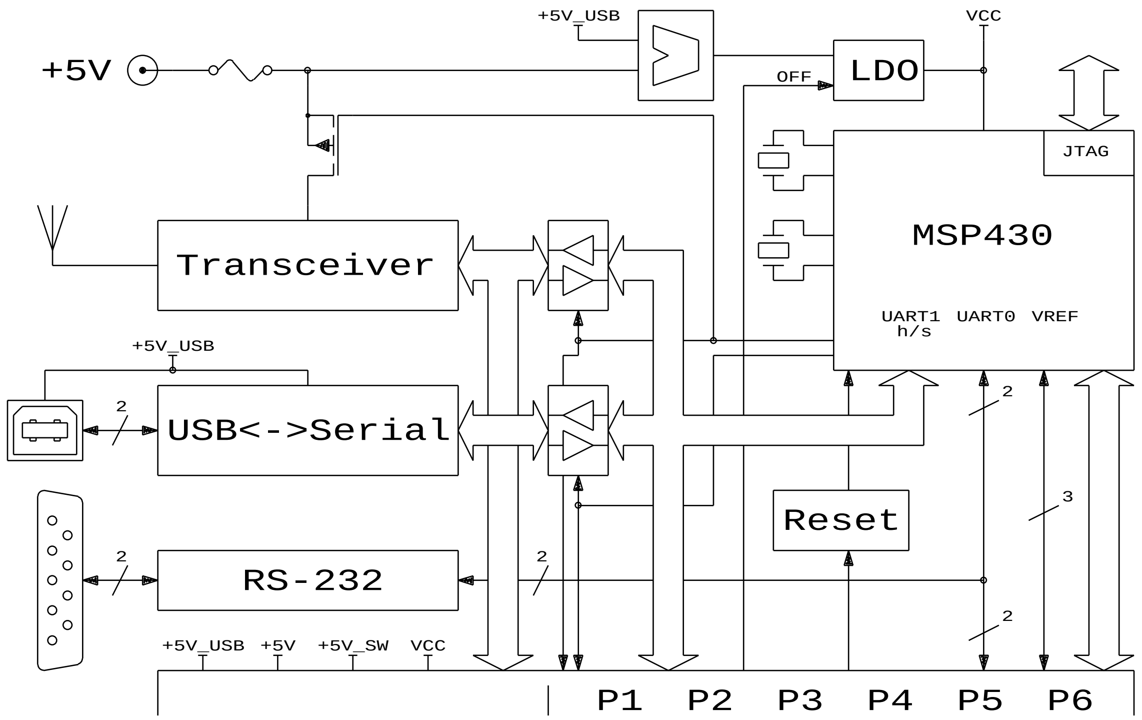


Figure 2: FM430 Flight Module Block Diagram

Part III

MSP430-centric Hardware Architecture

Power system – Requirements

* Many wireless transceivers operate with 5V supplies. The CubeSat Kit also accepts 5V-only PC/104 modules as payload to support those users who might need an additional, much higher-performance CPU, e.g. for image compression. Therefore this must be a mixed-supply system, supporting 5V and the MSP430's 3.3V.

* The CubeSat Kit provides USB connectivity as a means of communicating with ground personnel when fully assembled and / or immediately prior to launch. USB can provide 5V at 500mA max.

* As only 1W average power from the sun is available to charge solar cells on a 1U (10x10x10cm) CubeSat, the power consumption of the MSP430 and all support circuitry must be minimized so that the MSP430 can run 24x7 in its supervisory roles while batteries are being charged. Transmitting alone requires 4-5W instantaneous power.

* All power switching must be solid-state – no relays.

* Power supplies should exhibit good design for EMI resistance.

* Peak current draw can exceed 5A. Switches and connectors must be rated accordingly.

Power system – Solution

* 5V available system-wide on CubeSat Kit Bus™.
* Due to concerns over switching noise and system complexity, TI's TPS769xx family of LDO linear ultralow-power regulators was chosen to supply 3.3V on the CubeSat Kit's Flight Module:
    * 17µA max quiescent current @ 100mA output
    * Overcurrent protection
    * 1µA sleep current via –EN (on CubeSat Kit Bus)
    * -40º to +125º C operating range

These regulators require careful attention to choice of output capacitor for stabilization. Yet their stability is quite tolerant of changes in output capacitor ESR that will occur in space due to temperature fluctuations. Additionally, unused 3.3V power is available system-wide via CubeSat Kit Bus. *This LDO is the largest single power consumer on the Flight Module.* If the Flight Module is the only consumer of 3.3V power, overall system inefficiencies due to the 5V↔3.3V downconversion are irrelevant.

* Because the Flight Module runs at 3.3V, and system power is supplied at 5V, we have the opportunity to feed the 3.3V LDO from a multitude of 5V sources, via a simple Schottky diode drop. Therefore the Flight Module is powered from system 5V or USB 5V, whichever is present (Figure 3).

Power system – Solution (cont'd)

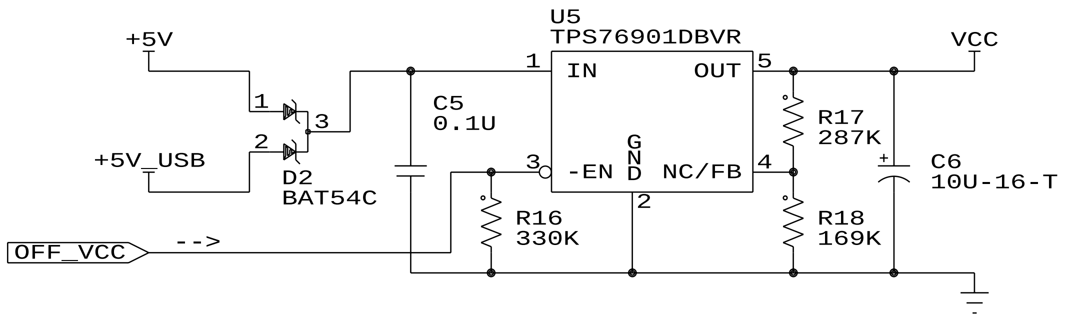


Figure 3: Multiple 5V Sources Feeding Ultralow-power 3.3V LDO Linear Regulator

Mixed 3.3V/5V design – Requirements

* Flight Module components running at 3.3V include:

  * MSP430 (all I/O on CubeSat Kit Bus)

  * Reset supervisor

  * Serial-to-USB converter (serial side)

* Flight Module components running at 5V include:

  * 2.4GHz spread-spectrum transceiver

  * Serial-to-USB converter (USB side)

* The MSP430 drives either the transceiver, the serial-to-USB converter or the CubeSat Kit Bus through USART1. Because of the high quiescent power consumption of the transceiver (1W) and serial-to-USB converter (2mW) compared to the MSP430, these devices should only be powered when connectivity is required. Therefore a zero-power, isolating and (for the transceiver) level-shifting interface must be employed.

* Power-up / power-down of 3.3V and 5V sides may occur separately at any time. MSP430 must be protected from overcurrents on its I/O pins from output drivers. Transceiver and serial-to-USB converter must not be powered through input pins via MSP430 outputs.

Mixed 3.3V/5V design – Challenges

* Any scheme that doesn't keep signals at the CMOS rails will consume large quiescent power.

* Resistive level shifters (e.g. for 3.3V↔5V) waste power and don't provide isolation.

* There are many integrated bidirectional level-shifting solutions on the market today. But few of them offer level translation and power-off isolation with zero current draw.

* Popular FET-based (bus) switches have simple, symmetric flow-through architectures that are well-suited to voltage translation and bus isolation. However, they require charge pumps to increase the voltage at each switch's gate so that V_{GS} is adequate to turn on the switch. This adds 10μA-2mA of current draw, typically on a *per-level-translated-pin* basis.

Mixed 3.3V/5V design – Solution

* Assuming that individual control outputs idle high, they can be level-shifted using discrete MOSFETs and resistors at zero power (Figure 4).
* To translate 3.3V up to 5V, we use half of an AHCT244 powered at 5V (on demand). The –OE control signal must also be level-shifted to 5V for minimum power consumption (Figure 5).
* To translate and isolate 5V down to 3.3V, we use half of an LVC244A powered at 3.3V (permanently). The –OE control signal must also pulled to the chip's VCC for minimum power consumption and power-on / power-off isolation (Figure 6).

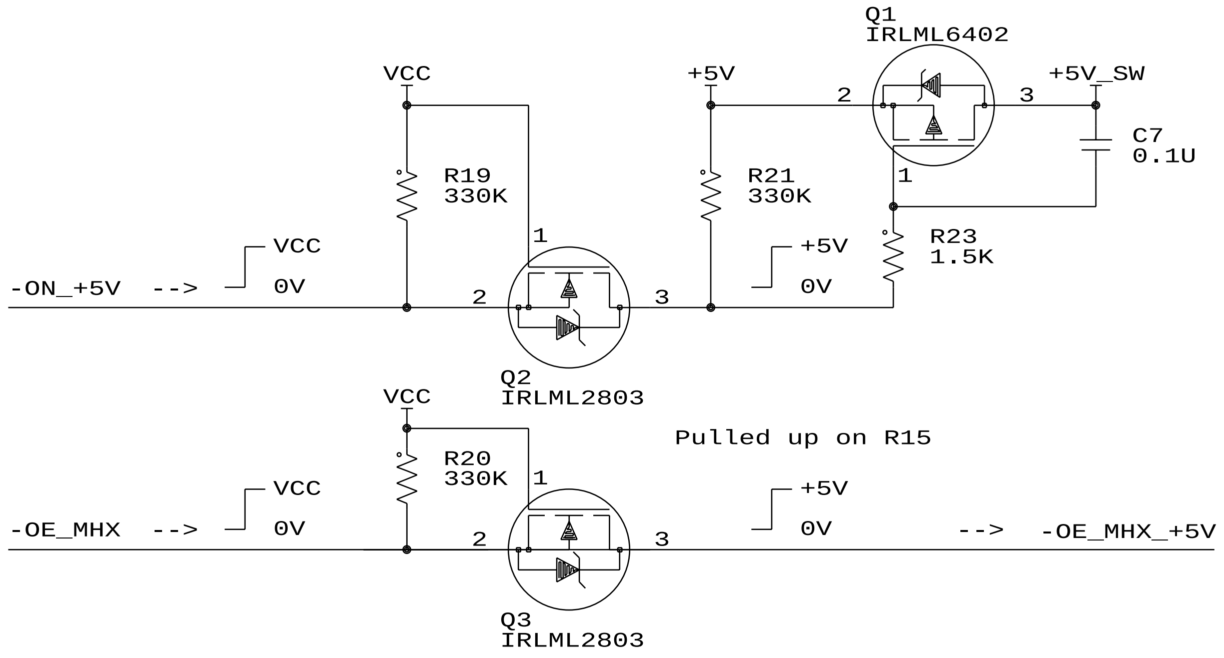


Figure 4: Level Shifting for Individual Unidirectional Control Signals

Mixed 3.3V/5V design – Solution (cont'd)

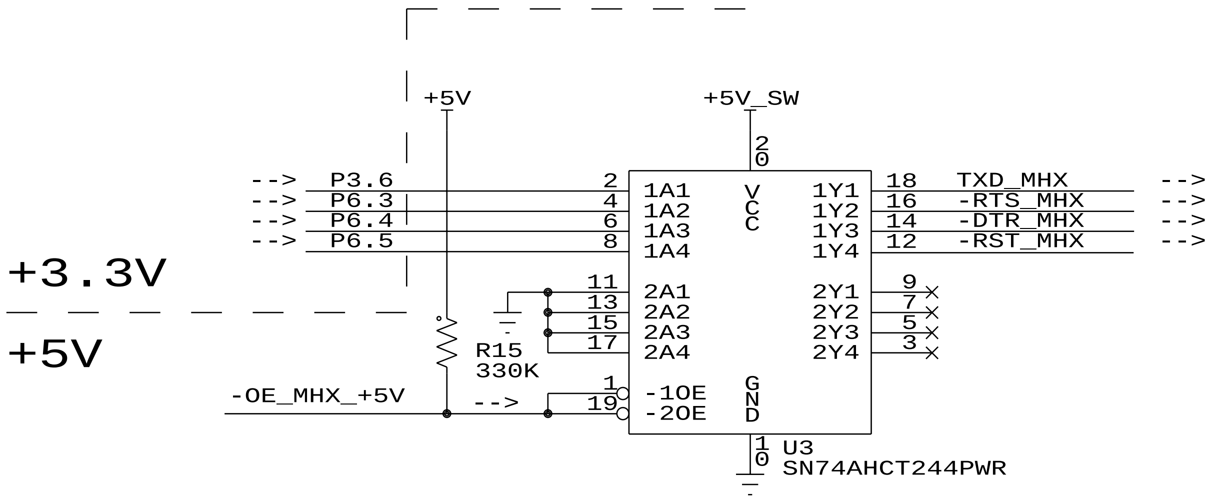


Figure 5: Level Shifting up to 5V using AHCT Logic

Mixed 3.3V/5V design – Solution (cont'd)

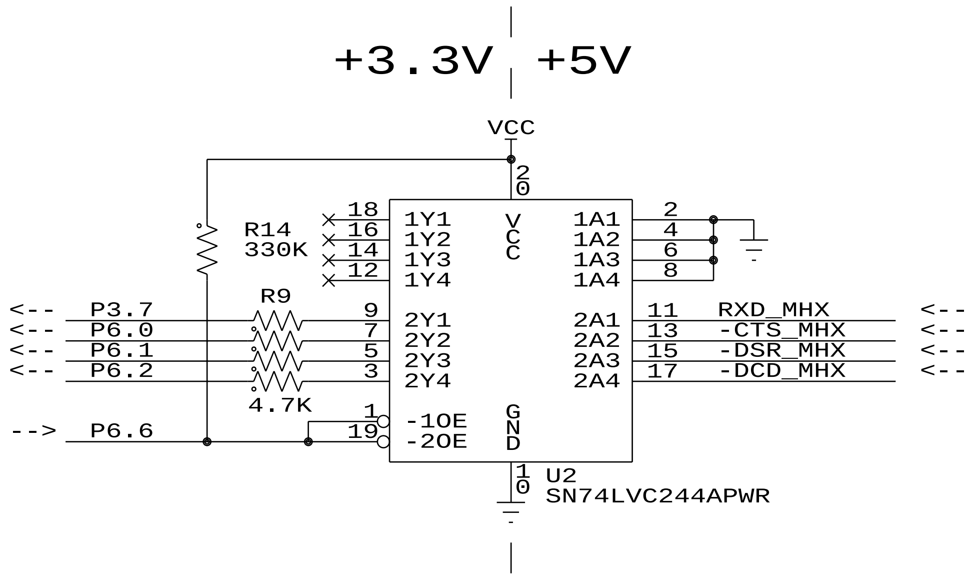


Figure 6: Level Shifting down to 3.3V and Isolating using LVC Logic

Isolated bus-powered serial-to-USB interface – Requirements

* The USB interface must power the Flight Module when a USB cable is connected.
* The Flight Module must be powered from either the system 5V or the USB interface automatically.
* When unpowered, the serial-to-USB converter must be completely isolated from the MSP430, and consume no power.
* Programmable support for USB VID, PID, etc.
* Customizable USB drivers for multiple platforms (Win, Mac OS, Linux, etc.).

Isolated bus-powered serial-to-USB interface – Solution

* FTDI's FT232BM with external configuration EEPROM and Win/Mac OS/Linux drivers.
* 5V from USB bus drives 3.3V LDO for Flight Module.
* Bus-powered design provides 3.3V to power LVC244A isolation circuitry via FT232BM's 3V3_OUT pin.
* When USB cable is removed, LVC244A automatically isolates MSP430 from FT232BM through its partial power-down mode (Figure 7).

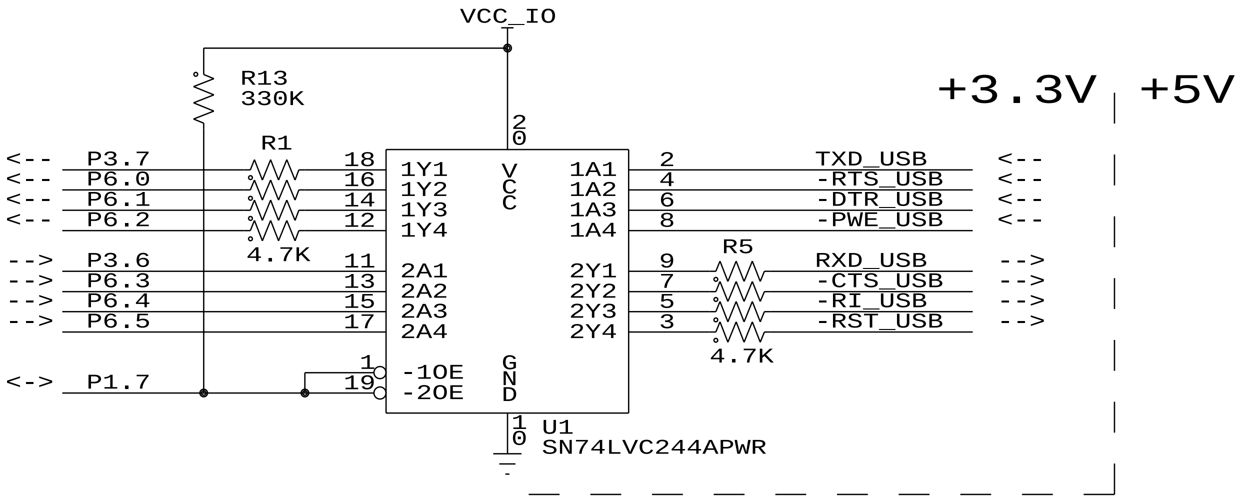


Figure 7: 3.3V Isolation Driven by Presence of External Power VCC_IO

Isolated bus-powered serial-to-USB interface – Solution (cont'd)

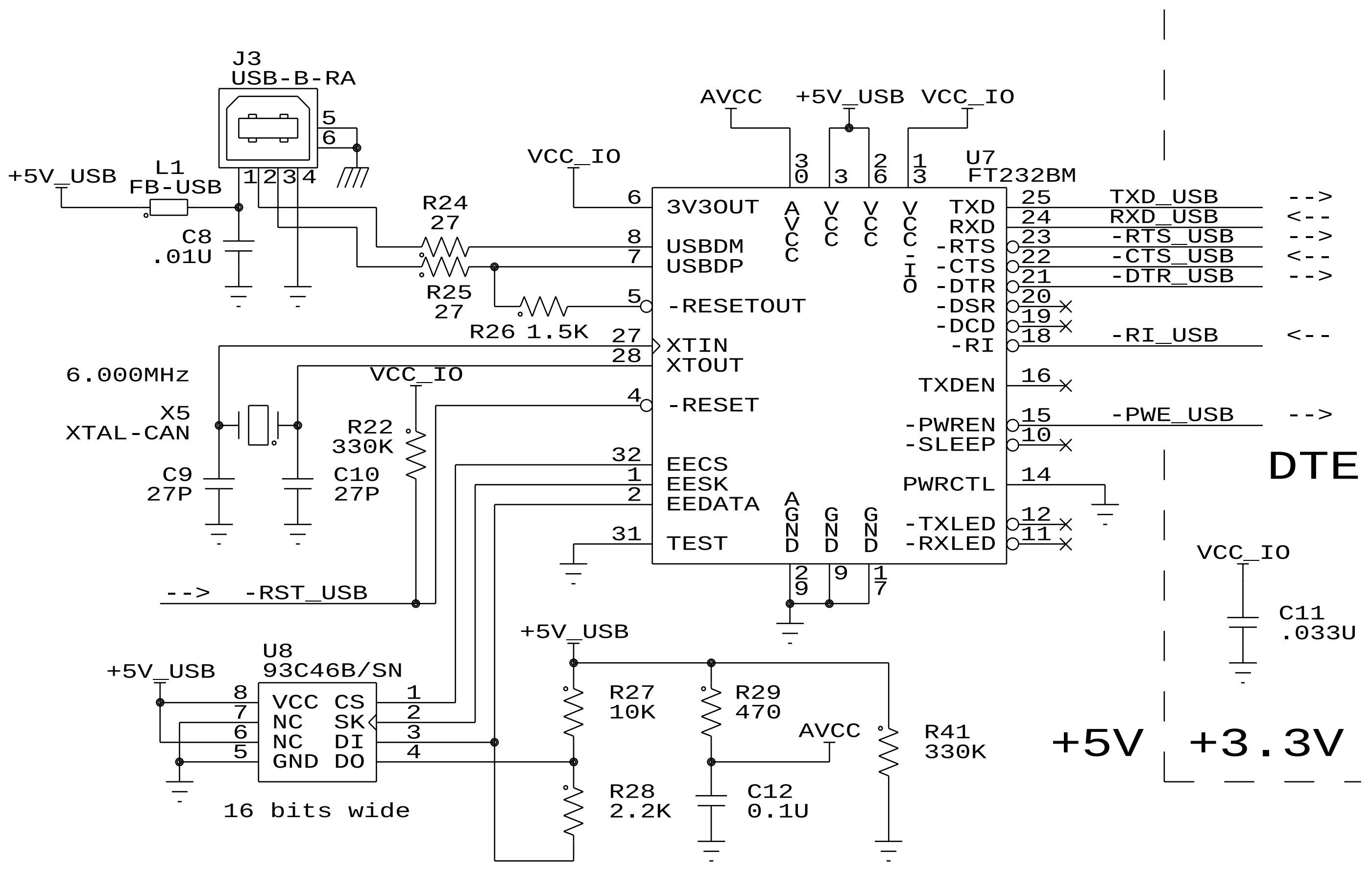


Figure 8: Bus-powered Serial-to-USB Converter for 3.3V Interface

Protecting the MSP430's I/O

* Absolute maximum over / undervoltage: ±0.3V
* Overvoltage specification can be violated if MSP430 is OFF and driving circuitry is ON. In-line resistor will protect MSP430 by dropping voltage and limiting current.
* Absolute maximum diode current at any pin: ±2mA
* Overcurrent specification can be violated if MSP430 I/O is configured as an output and connected to another output. This could happen due to a programming error if I/O pin is used as both input and output, depending on overall system configuration. In-line resistor will protect MSP430 by limiting current.
* Figure 6 and Figure 7 illustrate this technique. 4.7kΩ series limiting resistors chosen for adequate margin under all possible operating conditions (including user error).

Reset supervisor

* TI's micropower TPS3838 series nanopower supervisory circuit is used.
* Supply current < 300nA.
* -40º to +85º C operating range
* Open-drain output is required for reliable JTAG debugging.
* –MR input on CubeSat Kit Bus to allow external MSP430 reset.

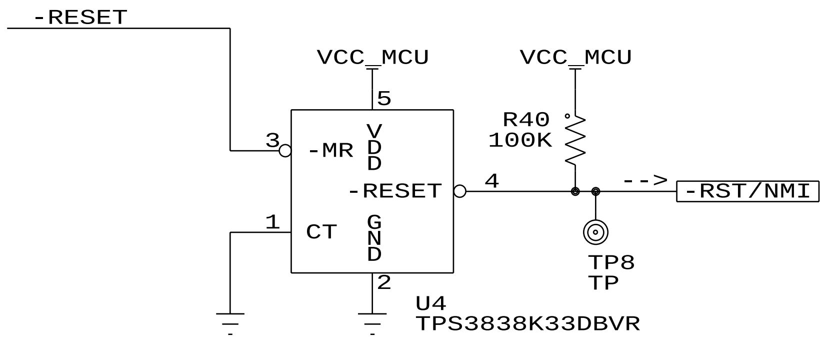


Figure 9: Nanopower Reset Supervisor with Open-drain Output

Multitasking the MSP430's I/O pins

* By isolating the transceiver and serial-to-USB converter from the MSP430's bus, we can configure the directions of the eight associated pins for multiple purposes: RS-232 Tx/Rx + DCE handshaking (transceiver enabled), RS-232 Tx/Rx + DTE handshaking (serial-to-USB converter enabled), and user-defined (neither enabled).

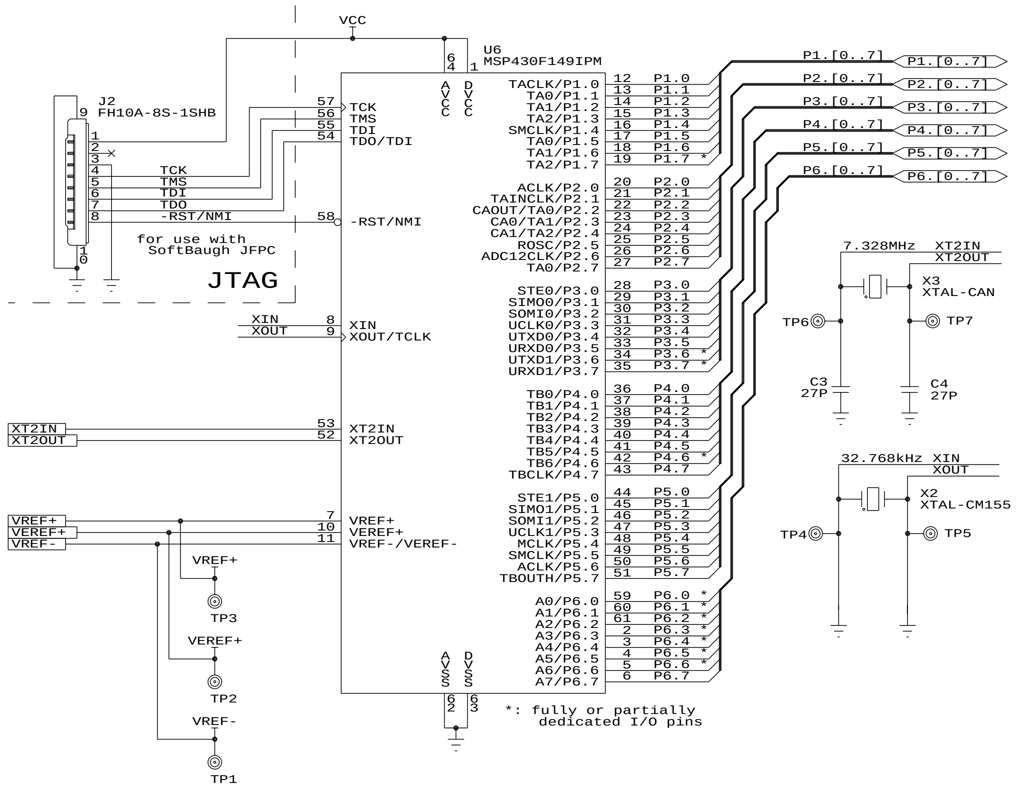


Figure 10: Dedicated I/O Pin Usage. Pin Modes Dictated by Configuration

Hardware Results

* System-wide power consumption of <250µW asleep, <10mW operational, functional over -40º to +85º C operating range.
* Implemented on a single 90x96mm six-layer PCB, topside components only.
* MSP430F149's entire I/O pinout (and more) is available on two PC/104-type stacking connectors.
* Mini-JTAG connector present for programming and debugging.

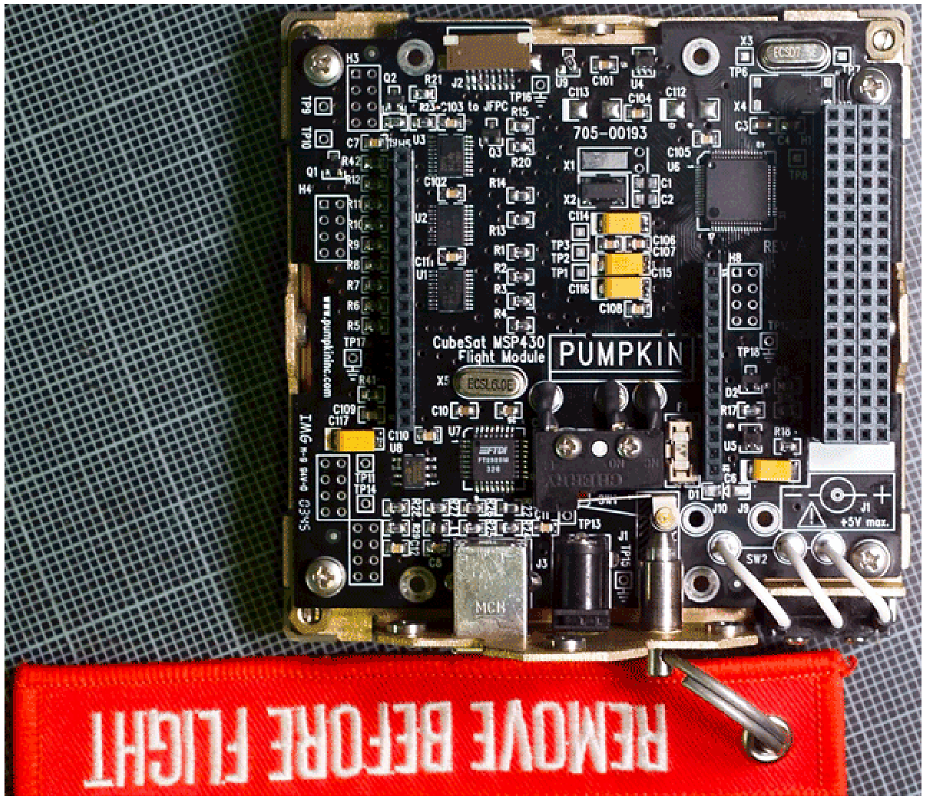


Figure 11: FM430 Flight Module Top View

Part IV

Rapid MSP430 Application Development

using Salvo™

The RTOS that runs in tiny places.™

Salvo™ RTOS features and operational details

* **Tasks**
    * 16 dynamic task priority levels.
    * "Run forever with one or more context switches" 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.[1]
* **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.

Salvo port to TI's MSP430

* **Memory requirements**
    * RAM usage per task control block: 14 bytes max.[2]
    * RAM usage per event control block: 6 bytes max.[3]
    * Stack size: Similar to typical foreground / background application.
    * Flash usage: 400-1700 bytes.

tutorial memory usage[4]	total Flash[5]	total RAM[6]
`tu1lite`	450	22
`tu2lite`	596	22
`tu3lite`	638	24
`tu4lite`	1148	34
`tu5lite`	1562	50
`tu6lite`	1678[7]	52[8]
`tu6pro`	1550[9]	48[10]

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

* **Context switching**

    25µs @ MCLK = 8MHz (with priorities, events, etc.).[11]

* **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.[12]

Salvo on the MSP430

* **Suitability**
  * MSP430's 2K RAM and 60K Flash are ideal for Salvo applications – 20-task, 30-event application consumes under 15% RAM and 5% Flash, leaving plenty of RAM and Flash 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.

CubeSat Kit 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 transmit data via transceiver

* **When interface is available, send data**

```
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 transceiver on, wait 1s while it resets */
    Enable_5V_to_MHX();
    OS_Delay(100, label);

    /* enter command mode, wait 1s */
    FM430PutstrTx1("ATA\r");
    OS_Delay(100, label);

    /* send some data, wait 1s, turn transceiver off */
    sprintf(strTmp, "TalkMHX:" " Ambient temp is %d C", DegC);
    FM430Msg1(strTmp);
    OS_Delay(100, label);
    Disable_5V_to_MHX();

    /* 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 3s period that the task waits for delays to expire, 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.
  - Once `TaskTalkMHX()` has acquired the USB / transceiver interface, all other tasks wishing to use the interface must wait (i.e. "are blocked") until `TaskTalkMHX()` releases the interface at the end of transmission.

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();
  }
}
```

Expanding 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.**
* **Run uIP-based web server supporting multiple simultaneous connections.**

Example application results

* **Application uses and is configured for:**
  * 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[13] on MSP430F149 for this application:**
  * 1132 bytes Flash (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.

Part V

Conclusion

MSP430 is right choice for Pumpkin's CubeSat Kit Flight MCU

* Single-chip MSP430F149 minimizes parts count.

* On-board peripherals (USARTS, A/D, timers, interrupts) and multi-purpose I/O enables very high level of application functionality and system integration.

* Ultralow-power operation and configurable clock module enable 24x7 MCU operation in hostile environment.

* 3.3V operation easily integrated into 5V design.

* C-friendly architecture with 60K Flash and 2K RAM presents opportunity for substantial functional software integration, thereby reducing system parts count.

* Wide range of high-quality programming tools available.

* Salvo RTOS on MSP430 simplifies software development, enables code modularity, and results in maintainable and expandable high-performance application with low power requirements.

* MSP430 upgrade path will increase functional integration and performance, thereby further reducing overall CubeSat Kit parts count, cost and complexity.

Salvo™
The RTOS that runs in tiny places.™

and the

CUBE SAT KIT™

are products of

PUMPKIN™ INCORPORATED

750 Naples Street
San Francisco, CA 94112 USA
tel: (415) 584-6360
fax: (415) 585-7948
web: http://www.pumpkininc.com/
web: http://www.cubesatkit.com/
email: info@pumpkininc.com

First presented at TI's 3rd annual MSP430 Advanced Technical Conference in Dallas, Texas on November 9-11, 2004.

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 and applicability to a wide range of microcontroller-based applications. He is also involved in a variety of other hardware and software projects.

CubeSat information

More information on the open CubeSat standard and the CubeSat community can be found at http://www.cubesat.info/ .

Copyright notice

End Notes

1	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.
2	Disabling timeouts reduces tcb size to 10 bytes. Optional tcb extensions (Salvo Pro only) require additional RAM per tcb.
3	Can be reduced to 4 bytes by disabling event types.
4	Salvo v3.2.0-b with IAR MSP430 C v1.26B.
5	In bytes. Does not include interrupt vectors.
6	In bytes. Does not include RAM allocated to the stack.
7	Includes 2 bytes from the CDATA0 section.
8	Includes 2 bytes on the IDATA0 section.
9	Includes 2 bytes from the CDATA0 section.
10	Includes 2 bytes on the IDATA0 section.
11	As measured with tu4lite.
12	Salvo Pro only.
13	IAR MSP430 C v2.10A