```
#include <main.h>

//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------

// These are all of the commands that are recognized - END is used to indicate
the end of the list when searching, it is not a command
const char  *mcm_commands[] = {"*CALIBRATE", "*ERASEALL", "*EVTLSTHDR",
"*EVTLSTDATA", "*RESET", "*STATUS", "*EVTTRCHDR",
                               "*EVTTRCDATA", "*ORIENT", "*DELTAV",
                               "*SYNCRTC", "*ACTIVATE", "*LOWPOWER",
                               "*SERIALID", "*GTRIGLVL",
                               "*ERASE", "*WIIFIRMWARE", "*DVX", "*DVY",
                               "*DVZ", "*SHOWEVENTS", "*SHOWFAT", "*DUMP",
                               "END"};

// Memory related buffers
const Configuration_t    CONFIGURATION @ "INFO A";                  // 128 bytes
holding the current configuration in flash
EVENT_DATA_t             events[NUMBER_OF_EVENTS];                  // These are
the headers for all of the events
UINT8                    eventBuffer[EVENT_ALLOCATION] @ "BUFFER"; // This
buffers the current event before copying to archive memory - allows faster
writes
UINT8                    glbTimer;
UINT16                   glbTriggers = 0;
Pretrigger_t             preTrigData;
UINT16                   zeros[3] = {2048, 2048, 2048};
char                     currentTime[20];
INT16                    glbCountDown = -1;
UINT8                    glbStatus = 0xff;

void main(void)
{
  char         msg[UART_BUF_LEN];
  UINT16       trigger;
  ERROR_STATE result;
  UINT8        i, j;
  FAT_DEFINITION_t   *myFAT;
  BOOL         reset = true;
  UINT32       dest;
  UINT8        *p;
  FAT_ENTRY_t *freeentry;

  WATCHDOG_OFF;
  BCSCTL2 = 0;
  BCSCTL1 = XT2OFF | RSEL2 | RSEL1 | RSEL0;
  DCOCTL = DCO0 | DCO2 | DCO1;
  FCTL2 = FWKEY + FSSEL_1 + FN3 + FN2;
// Initialize the hardware
  InitializeMSPPorts();
  TIMERA0_OFF;
  EVENT_OFF;
  P2TIMER_OFF;

// Configure Timer A to drive main loop operations (90Hz acquisition / timer
loop)
```

```
  TACTL = TASSEL_1 | TACLR | MC_0;          // src=ACLK (32.768kHz)
  TACCTL0 = OUTMOD_0;                      // CCR0 interrupt enabled, output mode
  = toggle
  TACCR0 = CCR_225HZ;                      // set the timer interval here approx.
  11.1msec
  TACTL |= MC_1;                           // Start Timer_A in up mode.
// Configure Timer B to drive fast ADC operations
  TBCTL = TBCLR | TBSSEL_1 | CNTL_0 | MC_0; // halt, clear, ACLK, 16-bit
  counter
  TBCCTL0 = OUTMOD_0;
  TBCCR0 = CCR_1800HZ;
  TBCTL |= MC_1; // start in the up mode
// Initialize the ADC for normal operating state (sample at 90Hz)
  ADC12CTL0 = ADC12ON | MSC | SHT0_3;       // Turn on ADC12, set sampling
  time, one start/sequence, 8clk S&H
  ADC12CTL1 = SHP | CONSEQ_1 | CSTARTADD_0; // Use sampling timer, single
  sequence
  ADC12MCTL0 = INCH_2 | SREF_2;                        // ref+=VeRef+, channel =
  A0
  ADC12MCTL1 = INCH_1 | SREF_2;                        // ref+=VeRef+, channel =
  A1
  ADC12MCTL2 = INCH_0 | SREF_2 | EOS;                  // ref+=VeRef+, channel =
  A2, end of seq.
//  ADC12CTL0 |= (ENC | ADC12SC);   // Start conversion by asserting enable
conversion
//  ADC12CTL0 &= (~ENC);             // This forces only a single round of
conversions to be performed

  // this should be done before MSP1 is powered on and it's reset is released
  if(MMA1201_init() != OK || MMA3201_init() != OK) glbStatus &= ~SENSOR_STATUS;

  SPI_init();
  if (RTC_init() == FAIL) set_RTC_date("00:00:12:01:01:2001");
  InitUART0(BAUD_RATE);

  _EINT();

  preTrigData.index = 0;
  preTrigData.active = false;

  if (CONFIGURATION.Firmware_Rev != FIRMWARE_REV) ClearSystem(&CONFIGURATION,
  &eventBuffer[0], &eventBuffer[EVENT_ALLOCATION]);
  if (ValidConfiguration(&CONFIGURATION) != OK)
  SetToDefaultConfiguration(&CONFIGURATION); // Check the configuration
  information

// Build the data structures to get to the data
  myFAT = w2FAT_build();
  reload_event_list(myFAT);

  myFAT->free_entries = 0;
  for (i=0;i<NUMBER_OF_EVENTS;i++) if (events[i].state == FREE)
  myFAT->free_entries += 1;
  myFAT->next_guid = 1;
  for (i=0;i<NUMBER_OF_EVENTS;i++) if ((events[i].state != FREE) &&
  (myFAT->next_guid <= events[i].guid)) myFAT->next_guid = events[i].guid+1;

// Need to set the number for the next event here
```

```
  TIMERA0_ON;
  P2TIMER_ON;
  EVENT_ON;
  glbTriggers = 0;
// Check the local flash to see if there is an event in it
  if (((EVENT_HEADER_t *)eventBuffer)->state != FREE) SET_TRIGGER(FLASH_EVENT);

  for (;;)
  {

    if (glbTriggers == 0) LPM3;

    BITSET(P2OUT, P2_MSP1_RST_OUT, MSP1_RESET_STATE);
    BITSET(P2OUT, P2_MSP2_RST_OUT, MSP2_RESET_STATE);

    if (glbTriggers)
    {
      trigger = 0x8000 >> __bit_count_leading_zeros_short(glbTriggers);
      CLR_TRIGGER(trigger);
      switch (trigger)
      {
        case EVENT_TRIGGER:
          if (reset != false)
            reset = false;
          else
          {
            if (preTrigData.active != false)
            {
              if (myFAT->free_entries > 0)
              {
                result = AcquireEvent(currentTime, myFAT->next_guid);
                myFAT->next_guid += 1;
                get_RTC_date(currentTime); // Update the current time here -
                guarantees that events have different times
                preTrigData.active = false;
              }
            }
          }
          break;
        case ADC_TRIG:
          ADC12CTL0 |= (ENC | ADC12SC);   // Start conversion by asserting
          enable conversion
          ADC12CTL0 &= (~ENC);             // This forces only a single round of
          conversions to be performed
          while ((ADC12IFG & BIT2) == 0) {}
          preTrigData.buffer[0][preTrigData.index] = ADC12MEM0;
          preTrigData.buffer[1][preTrigData.index] = ADC12MEM1;
          preTrigData.buffer[2][preTrigData.index] = invert(ADC12MEM2,
          CONFIGURATION.Intercepts[2]);
          preTrigData.index = (preTrigData.index + 1) & (PRETRIGGER_SAMPLES-1);
          if (preTrigData.index >= (PRETRIGGER_SAMPLES-1)) preTrigData.active =
          (preTrigData.active | true);
          break;
        case FLASH_EVENT:
          SET_TRIGGER(FLASH_EVENT);
          switch (((EVENT_HEADER_t *)eventBuffer)->state)
          {
            case FREE:
```

```
            CLR_TRIGGER(FLASH_EVENT);
            break;
          case NEW:
            dest = ((EVENT_HEADER_t *)eventBuffer)->baseAddr;
            for
            (p=eventBuffer;p<&eventBuffer[EVENT_ALLOCATION];p+=PAGE_BYTES)
            {
              while (FlashBusy() == true) {}
              dest = PP(dest, p, PAGE_BYTES, false);
              if (Event_Check()) break;
            } // 160 msec
            if ((glbTriggers & EVENT_TRIGGER) == 0)
            WriteFlashByte(&((EVENT_HEADER_t *)eventBuffer)->state, DELETED);
            // Writes data to the flash memory
            break;
          case DELETED:
            SafeLocalFlashErase(&eventBuffer[0],
            &eventBuffer[EVENT_ALLOCATION-1], Event_Check);
            CLR_TRIGGER(FLASH_EVENT);
            break;
        }
        break;
      case TIMER_4HZ:
        processNew(CONFIGURATION.UVWDeltaV, Event_Check);
        sendNotifications();
        break;
      case UART_TRIGGER:
        ProcessMessages(myUART.U0RXBuffer, myFAT);
        preTrigData.index = 0; // restart the buffers after communications
        preTrigData.active = false;
        break;
      case TIMER_1HZ:
        if(glbStatus != 0xff)
        {
          P3OUT |= 0b10000000; // green LED off
          P3OUT ^= 0b01000000; // toggle red LED
        }
        else
        {
          P3OUT |= 0b01000000; // red LED off
          P3OUT ^= 0b10000000; // toggle green LED
        }

        get_RTC_date(currentTime); // Update the current time here
        if (preTrigData.active)
        {
          zeros[0] = median(preTrigData.buffer[0], PRETRIGGER_SAMPLES);
          zeros[1] = median(preTrigData.buffer[1], PRETRIGGER_SAMPLES);
          zeros[2] = median(preTrigData.buffer[2], PRETRIGGER_SAMPLES);
        }

        if (myFAT->free_entries == 0) // Create a new free sector by cleaning
        up the fat
        {
          UINT8 sector, entry, counter;
          UINT8 index;
          UINT8 kill_sector;
```

```
          kill_sector = myFAT->sector_to_erase;
        // Find sector with all entries not used or locked and free it up
          for (sector = 0; sector < NUM_SECTOR; sector++)
          {
            counter = 0;
            for (entry = 0; entry < EVENTS_PER_SECTOR; entry++)
            {
              index = myFAT->fat[sector][entry].index;
              switch (events[index].state)
              {
                case FREE: case BELOW: case DELETED:
                  counter++;
                  break;
                case NEW: case PROCESSED: case SENT: case LOCKED:
                  break;
              }
            }
            if (counter == EVENTS_PER_SECTOR)
            {
              kill_sector = sector;
              break;
            }
          }
          if (kill_sector == myFAT->sector_to_erase) myFAT->sector_to_erase =
          (myFAT->sector_to_erase + 1 > NUM_SECTOR) ? 0 :
          myFAT->sector_to_erase + 1;

          SE(myFAT->sectorAddr[kill_sector], false); // sector erase
          for (entry = 0; entry < EVENTS_PER_SECTOR; entry++)
          {
            myFAT->fat[kill_sector][entry].free = true;
            index = myFAT->fat[kill_sector][entry].index;
            events[index].state = FREE;
            events[index].guid = 0xFFFFFFFF;
            events[index].id = 0xFF;
            myFAT->free_entries++;
          }
        }

        if (glbCountDown >= 0) if (glbCountDown == 0) SET_TRIGGER(SHUTDOWN);
        else glbCountDown--;
        break;
      case SHUTDOWN:
        while ((UTCTL0 & TXEPT) == 0) _NOP(); // Hold for last character to
        exit
        P3OUT |= (P3_LED1 | P3_LED2); // Turn off the LEDs
        TIMERA0_OFF;
        WATCHDOG_OFF;
        MSP2_POWER_OFF;
        MSP1_POWER_OFF;
        POWER_SENSORS(false);
        POWER_ANALOG(false);
        EVENT_OFF;
        P2TIMER_OFF;
        LPM4; // Shutdown everything but the UART0
        while (ActivateReceived(myUART.U0RXBuffer, currentTime) == false)
        LPM4;
        glbCountDown = -1;
```

```
          preTrigData.index = 0;
          preTrigData.active = false;
          InitUART0(BAUD_RATE);
          set_RTC_date(currentTime);
          POWER_ANALOG(true)
          __delay_cycles(10000L);
          POWER_SENSORS(true)
          __delay_cycles(10000L);
          MSP1_POWER_ON;
          __delay_cycles(10000L);
          MSP2_POWER_ON;
          __delay_cycles(10000L);
          get_RTC_date(currentTime);
          sprintf(msg,"*ACTIVATE:1/%s/\r", currentTime);
          SendMCMLine(msg);
          TIMERA0_ON;      // Start the base timer
          glbTriggers = 0;
          reset = true;
          TIMERA0_ON;
          WATCHDOG_ON;
          EVENT_ON;
          P2TIMER_ON;
          break;
        default:
          break;
      }
    }
  }
}

/*************************************************************************A
 * Utility functions
 *************************************************************************/

void  InitializeMSPPorts(void)
{
// Port 1
  P1SEL = 0;
  P1DIR = P1_MSP1_PWR | P1_MSP2_PWR;
  P1OUT = 0; // Turns off the power to the other processors (may need to
  stagger these)
  P1IES = P1_EVENT | P1_DMM_EVENT;
  P1IE = P1_EVENT;
// Port 2
  P2SEL = 0;
  P2DIR = P2_MSP1_RST_OUT | P2_MSP2_RST_OUT;
  P2OUT = 0; // Hold other processors in reset
  P2IES = P2_TIRQ;
  P2IE = P2_TIRQ;
// Port 3
  P3SEL = P3_WITNESS_TX | P3_WITNESS_RX | P3_UART_CLK;
  P3DIR = P3_AMUX0 | P3_AMUX1 | P3_AMUX_EN | P3_WITNESS_TX | P3_LED1 | P3_LED2;
  P3OUT = P3_LED1 | P3_LED2;
// Port 4
  P4SEL = 0;
  P4DIR = P4_TEST_XY | P4_TEST_Z | P4_CS_ADC16 | P4_CS_MSP1 | P4_CS_MSP2;
  P4OUT = P4_CS_ADC16 | P4_CS_MSP1 | P4_CS_MSP2;
// Port 5
```

```
  P5SEL = P5_SPI_MASTER | P5_SPI_MOSI | P5_SPI_MISO | P5_SPI_CLK;
  P5DIR = P5_CS_FLASH0 | P5_CS_FLASH1 | P5_ANALOG_ON_1 | P5_ANALOG_ON_2 |
  P5_SPI_MOSI | P5_SPI_CLK;
  P5OUT = P5_CS_FLASH0 | P5_CS_FLASH1 | P5_SPI_MOSI | P5_SPI_CLK;
  P5OUT |= P5_ANALOG_ON_1;
  __delay_cycles(100000);
  P5OUT |= P5_ANALOG_ON_2;
  __delay_cycles(100000);
// Port 6 - Configure the ports for the triax
  P6SEL = P6_Z_DATA | P6_Y_DATA | P6_X_DATA | P6_REF;
  P6DIR = 0;
  P6OUT = 0;
// Start up other processors
  MSP1_POWER_ON;
}

ERROR_STATE AcquireToFlash(const char *t, EVENT_HEADER_t *hdr)
{
  UINT16  data[3];

  UINT16  *dest, *pre;
  UINT16  sample = 0;
  UINT16  pretriggersample = 0, subsample = 0;

  if (eventBuffer[0] != 0xFF) return FAIL;

  dest = (UINT16 *)((UINT32)&eventBuffer + hdr->dataOffset);
  pre = (UINT16 *)((UINT32)&eventBuffer + hdr->preTriggerDataOffset);

  while (sample < SAMPLES_PER_EVENT)
  {
    WATCHDOG_ON;
    while ((TBCCTL0 & CCIFG) == 0) {};
    TBCCTL0 &= ~CCIFG;
    ADC12CTL0 |= (ENC | ADC12SC);   // Start conversion by asserting enable
    conversion
    ADC12CTL0 &= (~ENC);             // This forces only a single round of
    conversions to be performed
    while ((ADC12IFG & BIT2)==0) {}   // Wait for the conversion to end
    data[0] = ADC12MEM0;
    data[1] = ADC12MEM1;
    data[2] = ADC12MEM2;
    data[2] = invert(data[2], CONFIGURATION.Intercepts[2]);

    FCTL3 = FWKEY;                  // Clear Lock bit
    FCTL1 = FWKEY + WRT;            // Set WRT bit for write operation
    *(dest++) = data[0];
    *(dest++) = data[1];
    *(dest++) = data[2];
    FCTL1 = FWKEY;                  // Clear WRT bit
    FCTL3 = FWKEY + LOCK;           // Reset LOCK bit

    if (pretriggersample < PRETRIGGER_SAMPLES)
    {
      FCTL3 = FWKEY;                  // Clear Lock bit
      FCTL1 = FWKEY + WRT;            // Set WRT bit for write operation
      *(pre++) = preTrigData.buffer[0][(preTrigData.index + pretriggersample) &
      (PRETRIGGER_SAMPLES-1)];
```

```
      *(pre++) = preTrigData.buffer[1][(preTrigData.index + pretriggersample) &
      (PRETRIGGER_SAMPLES-1)];
      *(pre++) = preTrigData.buffer[2][(preTrigData.index + pretriggersample) &
      (PRETRIGGER_SAMPLES-1)];
      FCTL1 = FWKEY;                    // Clear WRT bit
      FCTL3 = FWKEY + LOCK;             // Reset LOCK bit
      pretriggersample++;
    }
    else
    {
      if (subsample++ >= RATIO)
      {
        preTrigData.buffer[0][preTrigData.index] = data[0];
        preTrigData.buffer[1][preTrigData.index] = data[1];
        preTrigData.buffer[2][preTrigData.index] = data[2];
        preTrigData.index = (preTrigData.index + 1) & (PRETRIGGER_SAMPLES-1);
        subsample = 0;
      }
    }

    sample++;
    if (sample < SAMPLES_100MSEC_LEFT) P1IFG &= ~P1_EVENT;
  }
  WriteFlashData(eventBuffer, hdr, sizeof(EVENT_HEADER_t)); // Writes data to
  the flash memory
}

ERROR_STATE AcquireToSPI(const char *t, EVENT_HEADER_t *hdr)
{
  UINT16  data[3];
  UINT16  i;
  UINT16  bufferA[40], bufferB[40], bufA = 0, bufB = 0;

  UINT32  dest, pre;
  UINT16  sample = 0;
  UINT16  subsample = 0;

  dest = hdr->baseAddr + hdr->dataOffset;
  pre = hdr->baseAddr + hdr->preTriggerDataOffset;

  for (sample=0;sample<SAMPLES_PER_EVENT;sample++)
  {
    WATCHDOG_ON;
    while ((TBCCTL0 & CCIFG) == 0) {};
    TBCCTL0 &= ~CCIFG;
    ADC12CTL0 |= (ENC | ADC12SC);   // Start conversion by asserting enable
    conversion
    ADC12CTL0 &= (~ENC);             // This forces only a single round of
    conversions to be performed
    while ((ADC12IFG & BIT2)==0) {}   // Wait for the conversion to end
    data[0] = ADC12MEM0;
    data[1] = ADC12MEM1;
    data[2] = ADC12MEM2;
    data[2] = invert(data[2], CONFIGURATION.Intercepts[2]);

    for (i=0;i<3;i++)
    {
      bufferA[bufA++] = data[i];
```

```
      if (bufA >= 32)
      {
        while (FlashBusy()) {}
        dest = PP(dest, bufferA, 64, false);
        bufA = 0;
      }
    }

    if (sample < PRETRIGGER_SAMPLES)
    {
      for (i=0;i<3;i++)
      {
        bufferB[bufB++] = preTrigData.buffer[i][(preTrigData.index + sample) &
        (PRETRIGGER_SAMPLES-1)];
        if (bufB >= 32)
        {
          while (FlashBusy()) {}
          pre = PP(pre, bufferB, 64, false);
          bufB = 0;
        }
      }
    }
    else
    {
      if (subsample++ >= RATIO)
      {
        preTrigData.buffer[0][preTrigData.index] = data[0];
        preTrigData.buffer[1][preTrigData.index] = data[1];
        preTrigData.buffer[2][preTrigData.index] = data[2];
        preTrigData.index = (preTrigData.index + 1) & (PRETRIGGER_SAMPLES-1);
        subsample = 0;
      }
    }
    if (sample < SAMPLES_100MSEC_LEFT) P1IFG &= ~P1_EVENT;
  }
  if (bufA > 0) dest = PP(dest, bufferA, bufA<<1, true); // write out the last
  of the buffer if writes buffered
  PP(hdr->baseAddr, hdr, sizeof(EVENT_HEADER_t), true); // write the header
  information for the event
}

BOOL AcquireEvent(const char *t, UINT32 guid)
{
  FAT_ENTRY_t      *freeentry;
  EVENT_HEADER_t   *event;
  UINT8            index;
  BOOL             result;
  UINT32           dest;

  TIMERA0_OFF;
  UART0_RX_IRQOFF; // Disable potentially interfering interrupts
  EVENT_OFF;
  P2TIMER_OFF;

  P3OUT &= ~0b11000000;
  event = EVENT_setconst(CONFIGURATION.Numerator, CONFIGURATION.Denomenator,
  CONFIGURATION.Intercepts, CONFIGURATION.roll, CONFIGURATION.pitch,
  CONFIGURATION.yaw);
```

```
  freeentry = w2FAT_findfree();
  freeentry->free = false;
  index = freeentry->index;
  event = EVENT_setevent(t, guid, zeros);
  event = EVENT_setstate(NEW);
  event = EVENT_setaddr(events[index].baseAddr);
  events[index].state = event->state;
  events[index].guid = event->guid;
  events[index].id = event->id;
  if (FlashBusy() == true)
  {
    AcquireToFlash(t, event);
    SET_TRIGGER(FLASH_EVENT);
    result = false;
  }
  else
  {
    AcquireToSPI(t, event);
    result = true;
  }
  P3OUT |= 0b11000000;

  P2TIMER_ON;
  EVENT_ON;
  TIMERA0_ON;
  UART0_RX_IRQON;

  return result;
}

void ClearSystem(const void *config, const void *bufStart, const void *bufEnd)
{
  SPIIRQON;
  BE(); // make sure to start with empty aux flash
  SPIIRQOFF;
  EraseSegment(config); // Erases the segment pointed to by the address
  EraseSegments(bufStart, bufEnd); // Erases the segment pointed to by the
  address
}

ERROR_STATE ValidConfiguration(const Configuration_t *config)
{
//---------------------------------------------------------------------------
// verifies that the configuration is valid enough to operate
//---------------------------------------------------------------------------
  Configuration_t temp;
  UINT16          check, checkA;

  if (config->Status_Structure_Revision != STATUS_REVISION) return FAIL;
  memcpy(&temp, config, sizeof(Configuration_t));
  check = temp.checksum;
  temp.checksum = 0;
  checkA = XORChecksum(&temp, sizeof(Configuration_t));
  if (checkA != check) return FAIL;
  if (INRANGE(temp.Orientation, 1, 24) == false) return FAIL;
  if (temp.Orientation != AnglesToOrientation(temp.roll, temp.pitch, temp.yaw))
  return FAIL;
  if (INRANGE(temp.UVWDeltaV[0], 1, 400) == false) return FAIL;
```

```
  if (INRANGE(temp.UVWDeltaV[1], 1, 400) == false) return FAIL;
  if (INRANGE(temp.UVWDeltaV[2], 1, 400) == false) return FAIL;

  return OK;
}

ERROR_STATE SetToDefaultConfiguration(const Configuration_t *config)
{
  Configuration_t temp;
  UINT16          i;
  UINT16          check;

  temp.Firmware_Rev = FIRMWARE_REV;
  temp.Status_Structure_Revision = STATUS_REVISION;
  temp.Orientation = 1;
  temp.roll = 0;
  temp.pitch = 0;
  temp.yaw = 0;
  temp.Calibration_Status = 0;
  temp.Percent_Memory_Available = 0;
  strcpy(temp.Witness_ID, "WW00INIT");
  for (i=0;i<3;i++) temp.XYZDeltaV[i] = temp.UVWDeltaV[i] = 1;
  for (i=0;i<3;i++) temp.Numerator[i] = 345;
  for (i=0;i<3;i++) temp.Denomenator[i] = 5632;
  for (i=0;i<3;i++) temp.Intercepts[i] = 2048;
  strcpy(temp.Calibration_Date, "00:00:00:00:00:0000");
  strcpy(temp.RTC_Time, "00:00:00:00:00:0000");
  temp.checksum = 0;
  check = XORChecksum(&temp, sizeof(Configuration_t));
  temp.checksum = check;
  EraseSegment(config);
  WriteFlashData(config, &temp, sizeof(Configuration_t));
}

void SetSerialNumber(const char *sn)
{
  Configuration_t temp;
  UINT16  check;
  memcpy(&temp, &CONFIGURATION, sizeof(Configuration_t));
  strncpy(temp.Witness_ID, sn, 8);
  temp.Witness_ID[8] = 0;
  temp.checksum = 0;
  check = XORChecksum(&temp, sizeof(Configuration_t));
  temp.checksum = check;
  EraseSegment(&CONFIGURATION);
  WriteFlashData(&CONFIGURATION, &temp, sizeof(Configuration_t));
}

void SetOrientation(const char *p)
{
  Configuration_t temp;
  UINT16 roll, pitch, yaw;
  UINT16  check;

  check = sscanf(p,"%d/%d/%d", &roll, &pitch, &yaw);
  if (check == 3)
  {
    memcpy(&temp, &CONFIGURATION, sizeof(Configuration_t));
```

```
    temp.Orientation = AnglesToOrientation(roll, pitch, yaw);
    temp.roll = roll;
    temp.pitch = pitch;
    temp.yaw = yaw;
    RotateXYZtoUVW(temp.Orientation, temp.XYZDeltaV, temp.UVWDeltaV);
    temp.checksum = 0;
    check = XORChecksum(&temp, sizeof(Configuration_t));
    temp.checksum = check;
    EraseSegment(&CONFIGURATION);
    WriteFlashData(&CONFIGURATION, &temp, sizeof(Configuration_t));
  }
}

void SetDeltaV(UINT16 x, UINT16 y, UINT16 z)
{
  Configuration_t temp;
  UINT16          check;

  memcpy(&temp, &CONFIGURATION, sizeof(Configuration_t));

  temp.XYZDeltaV[0] = x;
  temp.XYZDeltaV[1] = y;
  temp.XYZDeltaV[2] = z;
  RotateXYZtoUVW(temp.Orientation, temp.XYZDeltaV, temp.UVWDeltaV);

  temp.checksum = 0;
  check = XORChecksum(&temp, sizeof(Configuration_t));
  temp.checksum = check;
  EraseSegment(&CONFIGURATION);
  WriteFlashData(&CONFIGURATION, &temp, sizeof(Configuration_t));
}

void SetGTrigger(const char *p)
{
  Configuration_t temp;
  UINT16  level;
  UINT16  check;
  check = sscanf(p,"%d", &level);
  if (check != 1) return;
  memcpy(&temp, &CONFIGURATION, sizeof(Configuration_t));
  temp.GTrigger = level;
  temp.checksum = 0;
  check = XORChecksum(&temp, sizeof(Configuration_t));
  temp.checksum = check;
  EraseSegment(&CONFIGURATION);
  WriteFlashData(&CONFIGURATION, &temp, sizeof(Configuration_t));
}

UINT16  command_index(const char *cmd)
{
  UINT16  i=0;

  while (strcmp(mcm_commands[i], "END") != 0)
    if (strcmp(mcm_commands[i++], cmd) == 0) return (i-1);
  return -1;
}

ERROR_STATE parse_MCM(const char *msg, char *command, char *parameters)
```

```
{
//-----------------------------------------------------------------------------
// parses the input string and checks the CRC
//-----------------------------------------------------------------------------
  char    *p, *q;
  UINT16  check, check2;

// Make sure a valid message before parsing
  q = strchr(msg,0x04);
  if (q == 0) return FAIL;
  *q = 0;
  check = ComputeChecksum(msg, strlen(msg));
  if (*(q-1) == CR) *(q-1) = 0; // remove the carriage return
  if (*(q-2) == '/') *(q-2) = 0; // remove trailing slash
  p = strchr(msg, ':');
  if (p == 0) return FAIL;
  *p = 0;
//    *(q-1) = 0;
  strcpy(command, msg);
  strcpy(parameters, p+1);
  sscanf(q+1,"%x", &check2);
  return ((check == check2) || (check2 == 0xFFFF)) ? OK : FAIL;
}

ERROR_STATE DoCalibrate(const Configuration_t *config, const char *datetime)
{
  UINT32          sum[] = {0, 0, 0};
  Configuration_t temp;
  UINT16          check;
  UINT16          i;

  memcpy(&temp, config, sizeof(Configuration_t));
  for (i=0;i<256;i++)
  {
    ADC12CTL0 |= (ENC | ADC12SC);   // Start conversion by asserting enable
    conversion
    ADC12CTL0 &= (~ENC);             // This forces only a single round of
    conversions to be performed
    while ((ADC12IFG & BIT2)==0) {}   // Wait for the conversion to end
    sum[0] += ADC12MEM0;
    sum[1] += ADC12MEM1;
    sum[2] += ADC12MEM2;
    __delay_cycles(100000L);
  }
  for (i=0;i<3;i++) temp.Intercepts[i] = (UINT16)(sum[i] >> 8);

  temp.Numerator[0] = temp.Numerator[2] = 345;
  temp.Denomenator[0] = temp.Denomenator[2] = 5632;
  temp.Numerator[1] = 172;
  temp.Denomenator[1] = 5632;
  strcpy(temp.Calibration_Date, datetime);
  temp.Calibration_Status = true;
  temp.checksum = 0;
  check = XORChecksum(&temp, sizeof(Configuration_t));
  temp.checksum = check;
  EraseSegment(config);
  return (WriteFlashData(config, &temp, sizeof(Configuration_t)) == 0) ? OK :
  FAIL;
```

```
}

BOOL ActivateReceived(const char *msg, char *date)
{
//-----------------------------------------------------------------------------
// Checks to see if activate message received
//-----------------------------------------------------------------------------
  char  command[15], parameters[30];
  UINT16  index, trace;
  UINT16  x, y, z;

  if (parse_MCM(msg, command, parameters) != OK) return false;
  if (strcmp(command, "*ACTIVATE") == 0)
  {
    strcpy(date, parameters);
    return true;
  }
  return false;
}

void ProcessMessages(char *msg, FAT_DEFINITION_t *fat)
{
  char  command[15], parameters[30];
  char  line[UART_BUF_LEN-1];
  UINT16  index, trace;
  UINT16  x,y,z;
  UINT32  addr;

  if (parse_MCM(msg, command, parameters) != OK)
  {
    *msg = 0;
    return;
  }
  index = command_index(command);

  switch (index)
  {
    case 0: // calibrate
      get_RTC_date(parameters);
      sprintf(line,"%s:%d/\r", command, (DoCalibrate(&CONFIGURATION,
      parameters)==OK) ? 1 : 0);
      SendMCMLine(line);
      break;
    case 1: // erase archive flash
      w2FAT_freeall();
      reload_event_list(fat);
      sprintf(line,"%s:1/\r", command);
      SendMCMLine(line);
      break;
    case 4: // reset the system
      get_RTC_date(parameters);
      sprintf(msg,"%s:%d/%s/\r", command, 1, parameters);
      SendMCMLine(msg);
      FCTL2 = 0xDEAD;
      break;
    case 10: // synch the RTC
      set_RTC_date(parameters);
      get_RTC_date(parameters);
```

```
      sprintf(line,"%s:%s/\r", command, parameters);
      SendMCMLine(line);
      break;
    case 2: // get information about the fat - in FAT
      FormatListHeader(command, fat->fat_version);
      break;
    case 3: // get event summary data - in FAT
      if (fat->free_entries == NUMBER_OF_EVENTS)
      {
        sprintf(line, "%s:\r", command);
        SendMCMLine(line);
      }
      else FormatEventSummaryList(command);
      break;
    case 7: // get the trace data - do in EVENT
      sscanf(parameters, "%d", &trace);
      FormatTraceData(command, trace, Event_Check);
      break;
    case 5: // get the status
      sprintf(line,"%s:%d/%d/", command,
      CONFIGURATION.Status_Structure_Revision, CONFIGURATION.Firmware_Rev);
      SendMCMMessage(line, strlen(line), START_MESSAGE);
      sprintf(line,"%d/%d/%d/%d/", CONFIGURATION.roll, CONFIGURATION.pitch,
      CONFIGURATION.yaw, CONFIGURATION.Calibration_Status);
      SendMCMMessage(line, strlen(line), CONTINUE_MESSAGE);
      sprintf(line,"%d/%d/%s/%d/%d/%d/", w2FAT_freemem(),
      CONFIGURATION.Diagnostic, CONFIGURATION.Witness_ID,
      CONFIGURATION.XYZDeltaV[0], CONFIGURATION.XYZDeltaV[1],
      CONFIGURATION.XYZDeltaV[2]);
      SendMCMMessage(line, strlen(line), CONTINUE_MESSAGE);
      sprintf(line,"%d/%d/%d/", CONFIGURATION.Numerator[0],
      CONFIGURATION.Numerator[1], CONFIGURATION.Numerator[2]);
      SendMCMMessage(line, strlen(line), CONTINUE_MESSAGE);
      sprintf(line,"%d/%d/%d/", CONFIGURATION.Denomenator[0],
      CONFIGURATION.Denomenator[1], CONFIGURATION.Denomenator[2]);
      SendMCMMessage(line, strlen(line), CONTINUE_MESSAGE);
      sprintf(line,"%d/%d/%d/", CONFIGURATION.Intercepts[0],
      CONFIGURATION.Intercepts[1], CONFIGURATION.Intercepts[2]);
      SendMCMMessage(line, strlen(line), CONTINUE_MESSAGE);
      sprintf(line,"%s/%s/%u", CONFIGURATION.Calibration_Date, currentTime,
      *(UINT16 *)0xF000);
      SendMCMMessage(line, strlen(line), CLOSE_MESSAGE);
      break;
    case 8: // change the orientation
      SetOrientation(parameters);
      sprintf(line,"%s:%d/%d/%d/\r", command, CONFIGURATION.roll,
      CONFIGURATION.pitch, CONFIGURATION.yaw);
      SendMCMLine(line);
      break;
    case 9: // change the deltav
      sscanf(parameters,"%d/%d/%d", &x, &y, &z);
      SetDeltaV(x, y, z);
      sprintf(line,"%s:%d/%d/%d/\r", command, CONFIGURATION.XYZDeltaV[0],
      CONFIGURATION.XYZDeltaV[1], CONFIGURATION.XYZDeltaV[2]);
      SendMCMLine(line);
      break;
    case 12: // low power mode
      set_RTC_date(parameters);
```

```
      get_RTC_date(parameters);
      sprintf(line,"%s:1/%s/\r", command, parameters);
      SendMCMLine(line);
      glbCountDown = 14400;
      glbCountDown = 4;
      break;
    case 13: // set serial number
      SetSerialNumber(parameters);
      sprintf(line,"%s:%s/\r", command, CONFIGURATION.Witness_ID);
      SendMCMLine(line);
      break;
    case 14: // set G trigger level
      SetGTrigger(parameters);
      sprintf(line,"%s:%d/\r", command, CONFIGURATION.GTrigger);
      SendMCMLine(line);
      break;
    case 15: // erase single event
      sscanf(parameters, "%d", &trace);
      sprintf(line,"%s:%d/\r", command, (erase_event(trace) == OK) ? trace :
      0);
      SendMCMLine(line);
      break;
    case 16: // reset the system to reprogram
      sprintf(msg,"%s:%d/\r", command, 1);
      SendMCMLine(msg);
      FCTL2 = 0xDEAD;
      break;
    case 17:
      x = CONFIGURATION.XYZDeltaV[0];
      y = CONFIGURATION.XYZDeltaV[1];
      z = CONFIGURATION.XYZDeltaV[2];
      sscanf(parameters, "%d", &x);
      SetDeltaV(x, y, z);
      sprintf(line,"%s:%d/\r", command, CONFIGURATION.XYZDeltaV[0]);
      SendMCMLine(line);
      break;
    case 18:
      x = CONFIGURATION.XYZDeltaV[0];
      y = CONFIGURATION.XYZDeltaV[1];
      z = CONFIGURATION.XYZDeltaV[2];
      sscanf(parameters, "%d", &y);
      SetDeltaV(x, y, z);
      sprintf(line,"%s:%d/\r", command, CONFIGURATION.XYZDeltaV[1]);
      SendMCMLine(line);
      break;
    case 19:
      x = CONFIGURATION.XYZDeltaV[0];
      y = CONFIGURATION.XYZDeltaV[1];
      z = CONFIGURATION.XYZDeltaV[2];
      sscanf(parameters, "%d", &z);
      SetDeltaV(x, y, z);
      sprintf(line,"%s:%d/\r", command, CONFIGURATION.XYZDeltaV[2]);
      SendMCMLine(line);
      break;
    case 20: //"SHOWEVENTS"
      for (x=0;x<NUMBER_OF_EVENTS;x++)
      {
        sprintf(line,"  SE:%u/%u/%lx/%u/%lx", x,events[x].state,
```

```
        events[x].guid, events[x].id, events[x].baseAddr);
        switch (x)
        {
          case 0:
            SendMCMMessage(line, strlen(line), START_MESSAGE);
            break;
          case NUMBER_OF_EVENTS:
            SendMCMMessage(line, strlen(line), CLOSE_MESSAGE);
            break;
          default:
            SendMCMMessage(line, strlen(line), CONTINUE_MESSAGE);
            break;
        }
      }
      break;
    case 21: // "SHOWFAT"
      for (x=0;x<NUM_SECTOR;x++)
        for (y = 0; y < EVENTS_PER_SECTOR; y++)
        {
          sprintf(line,"  FAT:%u/%u/%lx/%u/%u", x, y, fat->fat[x][y].addr,
          fat->fat[x][y].free, fat->fat[x][y].index);
          if ((x==0) && (y==0))
            SendMCMMessage(line, strlen(line), START_MESSAGE);
          else
            SendMCMMessage(line, strlen(line), CONTINUE_MESSAGE);
        }
      line[0] = 0;
      SendMCMMessage(line, strlen(line), CLOSE_MESSAGE);
      break;
    case 22: // "*DUMP" dump the flash out the uart
      sprintf(line,"%s:1/\r", command);
      SendMCMLine(line);
      TIMERA0_OFF;
      UART0_RX_IRQOFF; // Disable potentially interfering interrupts
      EVENT_OFF;
      P2TIMER_OFF;
      addr = 0L;
      UART0_TX_IRQON;
      while (addr < MEMORY_BYTES)
      {
        WATCHDOG_ON;
        addr = READ(line, addr, 32);
        SendBuffer(line, 32, false);
        while(UART0_BUSY) {}
      }
      while(UART0_SHIFTING) {}
      UART0_TX_IRQOFF;
      P2TIMER_ON;
      EVENT_ON;
      TIMERA0_ON;
      UART0_RX_IRQON;
      break;
    default:
      break;
  }
}

void  SafeLocalFlashErase(const void *start, const void *end, BOOL
```

```
(*FLASH_abort)(void)) @ "IDATA0"
{
//-------------------------------------------------------------------------
// This routine MUST run out of RAM in order to continue checking for events
while
// erasing the local memory.
//-------------------------------------------------------------------------
  UINT8          *p;

  WATCHDOG_OFF;
  _DINT();
  for (p=(UINT8 *)end;p>=(UINT8 *)start;p-=SEGMENT_SIZE)
  {
    FCTL1 = FWKEY + ERASE;            // Set Erase bit
    FCTL3 = FWKEY;                    // Clear Lock bit
    *(unsigned char *)p = 0;      // Dummy write to erase Flash segment
    while (FCTL3 & BUSY)
    {
      if (FLASH_abort() != false)
      {
// Terminate the current flash cycle and bail out
        FCTL3 |= EMEX;
        break;
      }
    }
    FCTL1 = FWKEY;                    // Clear WRT bit
    FCTL3 = FWKEY + LOCK;             // Reset LOCK bit
  }
  _EINT();
  WATCHDOG_ON;
}

BOOL Event_Check(void) @ "IDATA0"
{
// This routine needs to reside in RAM so that it can run while erasing MSP430
flash - copy and execute
  if (P1IFG & P1_EVENT)
  {
    SET_TRIGGER(EVENT_TRIGGER);
    P1IFG &= ~P1_EVENT;
    return true;
  }
  return false;
}

//-------------------------------------------------------------------------
// COMMUNICATIONS ROUTINES
//-------------------------------------------------------------------------
ERROR_STATE FormatListHeader(const char *cmd, UINT8 fat_version)
{
  UINT8 counter = 0;
  char  line[UART_BUF_LEN];
  UINT8 i;

  for (i=0;i<NUMBER_OF_EVENTS;i++)
    switch (events[i].state)
    {
      case FREE: case DELETED:
```

```
        break;
      case NEW: case PROCESSED: case SENT: case LOCKED:
        counter++;
        break;
    }
  sprintf(line,"%s:%d/%d/%d/\r", cmd, fat_version, sizeof(EVENT_HEADER_t),
  counter);
  SendMCMLine(line);
}

ERROR_STATE FormatEventSummaryList(const char *cmd)
{
  char          line[UART_BUF_LEN];
  unsigned long delay;
  UINT8         i;
  EVENT_HEADER_t  *event;

  sprintf(line, "%s:", cmd);
  SendMCMMessage(line, strlen(line), START_MESSAGE);
  for (i=0;i<NUMBER_OF_EVENTS;i++)
    switch (events[i].state)
    {
      case FREE: case DELETED:
        break;
      case NEW: case PROCESSED: case SENT: case LOCKED:
        CLEAR_ACK_NAK;
        event = EVENT_fetch(events[i].baseAddr);
        sprintf(line, "%d/%s/%d/%d/%d/%d/%d/%d/?\r\n", event->id, event->time,
        event->deltav[0], event->deltav[1], event->deltav[2], event->peakG[0],
        event->peakG[1], event->peakG[2]);
        SendMCMMessage(line, strlen(line), CONTINUE_MESSAGE); // wait for
        response for each line - does not resend
        glbTimer = 0;
        while (myUART.replyReceived == false) if (glbTimer == 3) break;
        break;
    }
  SendMCMMessage("", 0, CHECKSUM_ONLY);
}

#define BLOCK_BYTES 512
ERROR_STATE FormatTraceData(const char *cmd, UINT8 id, BOOL (*abort)(void))
{
//-----------------------------------------------------------------------------
// Need to find the event id then send the formatted trace data
//-----------------------------------------------------------------------------
  UINT8           sector, entry;
  char            line[UART_BUF_LEN];
  EVENT_HEADER_t  *eventHeader;
  UINT8           data[UART_BUF_LEN];
  register UINT32 start, blockend, addr, end, length;
  UINT16          bufferLength;
  UINT16          sentBlocks = 0;
  UINT8           i;

  for (i=0;i<NUMBER_OF_EVENTS;i++)
    if (events[i].id == id)
    {
      eventHeader = EVENT_fetch(events[i].baseAddr);
```

```
      sprintf(line, "%s:%d/%d/%d/\r", cmd, eventHeader->event_header_version,
      eventHeader->id, EVENT_ALLOCATION);
      SendMCMLine(line);
      glbTimer = 0;
      while (glbTimer < 2) {}

      addr = events[i].baseAddr;
      end = addr + EVENT_ALLOCATION;

      while (addr < end)
      {
        sentBlocks++;
        start = addr;
        blockend = start + BLOCK_BYTES;
        addr = READ(data, start, UART_BUF_LEN);
        CLEAR_ACK_NAK;
        SendMCMMessage(data, UART_BUF_LEN, START_MESSAGE); // initializes
        checksum
        while (addr < blockend)
        {
          addr = READ(data, addr, UART_BUF_LEN);
          SendMCMMessage(data, UART_BUF_LEN, CONTINUE_MESSAGE);
        }
        SendMCMMessage("", 0, CHECKSUM_NOSEP);
        glbTimer = 0;
        while (myUART.replyReceived == false) if (glbTimer == 3) return FAIL;
        if ((myUART.replyReceived == true) && (myUART.ACKreceived == false))
        {
          addr = start; // NAK received
          __delay_cycles(1000000L);
        }
        if (abort()) return FAIL; // cancel download if there is an event
        if (sentBlocks > 50) return FAIL; // There is a real communications
        issue here
      }
      return OK;
    }
  return FAIL;
}

ERROR_STATE processNew(const UINT16 DV[], BOOL (*abort)(void))
{
  UINT8    i;
  EVENT_HEADER_t   *hdr;

  if (FlashBusy() == true) return FAIL; // skip if the aux flash is busy
  for (i=0;i<NUMBER_OF_EVENTS;i++)
    if (events[i].state == NEW)
    {
      hdr= EVENT_process(events[i].baseAddr, DV, abort);
      if (hdr == NULL) return FAIL;
      EVENT_write();
      hdr = EVENT_fetch(events[i].baseAddr);
      if (hdr->state == NEW)
        _NOP();
      events[i].state = hdr->state;
      events[i].guid = hdr->guid;
```

```
      events[i].id = hdr->id;
      break;
    }
  return OK;
}

ERROR_STATE sendNotifications(void)
{
  UINT32  ulPackedDV;
  UINT8   i;
  EVENT_HEADER_t  *event;

  if (FlashBusy() == true) return FAIL; // skip if the aux flash is busy
  for (i=0;i<NUMBER_OF_EVENTS;i++)
    if (events[i].state == PROCESSED)
    {
      event = EVENT_fetch(events[i].baseAddr);
      ulPackedDV =
      PackDeltaV(event->deltav[0],event->deltav[1],event->deltav[2]);
      SendEvent(events[i].id, ulPackedDV);
      EVENT_setstate(events[i].state = SENT);
      EVENT_write();
      break;
    }
  return OK;
}

ERROR_STATE erase_event(UINT8 id)
{
  UINT8 i;
  EVENT_HEADER_t  *event;

  for (i=0;i<NUMBER_OF_EVENTS;i++)
    if (events[i].id == id)
    {
      event = EVENT_fetch(events[i].baseAddr);
      EVENT_setstate(events[i].state = DELETED);
      EVENT_write();
      return OK;
    }
  return FAIL;
}

void reload_event_list(FAT_DEFINITION_t *myFAT)
{
//---------------------------------------------------------------------------
// Scans the aux memory and populates the event list
//---------------------------------------------------------------------------
  UINT8           i,j;
  EVENT_DATA_t    *event;
  EVENT_HEADER_t  *lclEventHeader;

  for (i=0;i<NUM_SECTOR;i++)
    for (j=0;j<EVENTS_PER_SECTOR;j++)
    {
      event = &events[myFAT->fat[i][j].index];
      event->baseAddr = myFAT->fat[i][j].addr;
      lclEventHeader = EVENT_fetch(event->baseAddr);
```

```
      event->state = lclEventHeader->state;
      event->guid = lclEventHeader->guid;
      event->id = lclEventHeader->id;
      if (event->state != FREE) myFAT->fat[i][j].free = false;
    }
}


/*****************************************************************
 * ISRs
 *****************************************************************/

void Port1ISR (void) __interrupt[PORT1_VECTOR]
{
  if (P1IFG & P1_EVENT)
  {
    SET_TRIGGER(EVENT_TRIGGER);
    P1IFG &= ~P1_EVENT;
  }
  LPM3_EXIT;
}

void Port2ISR (void) __interrupt[PORT2_VECTOR]
{
  static UINT8  counterA;

  if (P2IFG & P2_TIRQ)
  {
    P2IFG &= ~P2_TIRQ;
    SET_TRIGGER(TIMER_4HZ);
    if ((counterA = ((counterA + 1) & 0b0011)) == 0) SET_TRIGGER(TIMER_1HZ);
    glbTimer++;
  }
  LPM3_EXIT;
}

void TimerAISR (void) __interrupt[TIMERA0_VECTOR]
{
  WATCHDOG_ON;
  TACCTL0 &= ~CCIFG;
  SET_TRIGGER(ADC_TRIG);
  LPM3_EXIT;
}
```

```
//-------------------------------------------------------------------------
// Flash routines for the MSP430
//-------------------------------------------------------------------------
#ifndef __flash
#define __flash

#include <msp430x14x.h>
#include <main.h>

#define SEGMENT_SIZE   0x200
extern unsigned char __end_CONST[];
extern unsigned char __begin_INTVEC[];
extern unsigned char __begin_CODE[];
extern unsigned char *StartUnusedFlash;
extern unsigned char *EndOfFlash;
extern unsigned char *StartCode;

int           WriteFlashData(const void *dest, const void *src, int nBytes); //
Writes data to the flash memory
int           WriteFlashByte(const void *dest, const UINT8 data); // Writes
data to the flash memory
int           WriteFlashWord(const void *dest, const UINT16 data); // Writes
data to the flash memory
int           EraseSegment(const void *dest); // Erases the segment pointed to
by the address
int           EraseSegments(const void *start, const void *end); // Erases the
segment pointed to by the address
unsigned char *GetUnusedFlashStart(void); // Returns the start of available
FLASH memory
unsigned char *GetFlashEnd(void); // Returns the end of Flash memory
unsigned char *GetCodeStart(void);

#endif
```