David Sidrane

From: David Sidrane [david_s5@usa.net]
Sent: Monday, February 10, 2014 8:51 AM
To: 'zachary@spark.io'
Cc: 'Zach@Spark.io'
Subject: status and a written walk through

Hi Zachary,

Great speaking with the other today!

Below this you will find status and a written walk through with code annotation.

I have been testing aucInactivity of 0 with a timeout of 8S for connect and 20 S for all other net ops it is working well for more than 3 days!

It also had code to restart the CC3000 on CFOD detection.

It would be good to have it tested by you guys on the network that causes CFOD regularly.

Bin can be found here: http://nscdg.com/spark/0_inact_20_sec_recovery_log_core-firmware.zip

It Has recovery code in it to detect the CFOD and does a reconnect in about 20 second total.

I have ran it for over 72 hours with no issues (hangs or faults) and recovers from network issues in 20 seconds.!

It will Log CFOD;s as

0000164364: void SPARK_WLAN_Loop() (582):Resetting CC3000 due to 2 failed connect attempts

The test will send a udp packet every 60 seconds to 10.10.0.1 and also does an http get of a lot (3880+ bytes) of data from nscdg.com

Log data goes to Serial1 (3.3V tx,rx pins)

All the log data is critical, because the faults are logged after being detected and mitigated so they do not sent the CPU into a hardfault. Please send the logs and LED observations. (if it dies in a red ...---... Blink N ...---...)

1(Faults,RGB_COLOR_RED,HardFault) 12 (Faults,RGB_COLOR_RED,NMIFault)
3 (Faults,RGB_COLOR_RED,MemManage)4 (Faults,RGB_COLOR_RED,BusFault)
5 (Faults,RGB_COLOR_RED,UsageFault)
6 (Cloud,RGB_COLOR_RED,InvalidLenth)
7 (System,RGB_COLOR_RED,Exit)
8 (System,RGB_COLOR_RED,OutOfHeap)
9 (System,RGB_COLOR_RED,SPIOverRun)
10 (Softare,RGB_COLOR_RED,AssertionFailure)11 (Softare,RGB_COLOR_RED,InvalidCase)

Status and a written walk through with code annotation is below

Aloha,

David Sidrane
NscDg
59-556 Akanoho Place
Haleiwa, Hawaii 96712

David_S5@usa.net
Tel 808.638.9004
Cel 808.780.9004
Fax 206.339.8691

Unsolved:

1. Problem: Socket with inactivity timeout is antithetical to aurduio use case of init in setup() use in loop()

2. Problem: Socket of spark protocol is opened by user code and read from spark
   The inactivity Timer = 0 seems to fix this!

3. Problem: TI Nvram USER 1 or USER 2 "files" may read back as valid but not necessarily with spark's data set. - Causes connectivity issues.

4. Nuisance: Nice if the project was tied in git to libs. So commits that have lib changes and core changes are tied together. This is the classic one-tree vs 3 trees version control argument. For what it is worth - I prefer all source and third party in 1 tree and tools in a referenced sub tree.

Enhancements:

Added compile time and runtime controlled debugging.

See DEBUG_BUILD and RELEASE_BUILD in make files.

The compile time log level is different. For DEBUG and RELEASE.

```
#if defined(DEBUG_BUILD)
#define LOG_LEVEL_AT_COMPILE_TIME LOG_LEVEL
#define LOG_LEVEL_AT_RUN_TIME LOG_LEVEL         // Set to allow all LOG_LEVEL and above messages to be displayed conditionally by levels.
#endif

#if defined(RELEASE_BUILD)
#define LOG_LEVEL_AT_COMPILE_TIME ERROR_LEVEL
#define LOG_LEVEL_AT_RUN_TIME ERROR_LEVEL
#endif
```

To use this feature: compile with DEBUG_BUILD or RELEASE_BUILD
And add this to application.cpp to route the output.

```
void debug_output_(const char *p)
{
  static boolean once = false;
if (!once)
   {
     once = true;
     Serial1.begin(115200);
   }

Serial1.print(p);
}
```

(It probably should be a macro) DEFINE_DEBUG(Serial,1)

N.B define `USE_ONLY_PANIC` To remove **all** but the panic blinker

This adds Debug logging and Panic features.

```
/*
 * This module supports runtime and compile time message filtering.
 *
 * For a message be compiled in the message type has to be >= to then the LOG_LEVEL_AT_COMPIL
 * For a message to be output the message type has to be >= then the LOG_LEVEL_AT_RUN_TIME
 *
 * This module assumes one of two #defines RELEASE_BUILD or DEBUG_BUILD exist
 * Then the LOG_LEVEL_AT_COMPILE_TIME and LOG_LEVEL_AT_RUN_TIME are set based on RELEASE or D
 * For a release build
 *     LOG_LEVEL_AT_COMPILE_TIME = WARN_LEVEL
 *         LOG_LEVEL_AT_RUN_TIME = WARN_LEVEL
 * For a debug build:
 *     LOG_LEVEL_AT_COMPILE_TIME = LOG_LEVEL
 *     LOG_LEVEL_AT_RUN_TIME = LOG_LEVEL
 * The API
 * LOG(fmt,...)
 * DEBUG(fmt,...)
 * WARN(fmt,...)
 * ERROR(fmt,...)
 * PANIC(code,fmt,...)
 */
```

Usage:

```
define DEBUG_BUILD=y on make command line and add the following to your Application.cpp

void debug_output_(const char *p)
{
  static boolean once = false;
 if (!once)
   {
     once = true;
     Serial1.begin(115200);
   }

 Serial1.print(p);
}
The API looks like

LOG("Want %d more cores",count);
WARN("Running %s on cores only %d more left",,"Low",count);
DEBUG("connection closed %d",retValue);
ERROR("Flash write Failed @0x%0x",badd_address);
PANIC(HardFault,"Hit Hardfault");
```

Panic codes are defined in panic_codes.h

Added Panic with SOS on LED.
This is SW magic (or better living through macros :)) ,

To use just add a line to panic_codes.h

```
// Add a panic code def_panic_codes(_class, led, code)
// the panic module will send SOS followed by 900 ms delay
// followed by 300 blinks of the value of code
/// ...---...    code    ...---...

This is added to the while(1) loop in the code to help identify a Fault or Fault condition
Currently used

def_panic_codes(Faults,RGB_COLOR_RED,HardFault)      1 blink
def_panic_codes(Faults,RGB_COLOR_RED,NMIFault)       2 etc....
def_panic_codes(Faults,RGB_COLOR_RED,MemManage)
def_panic_codes(Faults,RGB_COLOR_RED,BusFault)
def_panic_codes(Faults,RGB_COLOR_RED,UsageFault)

def_panic_codes(Cloud,RGB_COLOR_RED,InvalidLenth)

def_panic_codes(System,RGB_COLOR_RED,Exit)
def_panic_codes(System,RGB_COLOR_RED,OutOfHeap)
```

and the then just use it, no muss, no fuss!

```
void _exit(int status)
{
        PANIC(Exit,"Exit Called");
            while(1);
}
```

Bug fixes and Where:

1. Hang in driver on race WIFI_INT (read) v SpiWrite
   Solution: was re-write

WIFI_INT ----------_____
SpiWrite ----------_____

SPARK_Firmware_Driver/src/cc3000_spi.c

```
//
// We need to prevent here race that can occur in case 2 back to back packets are sent to the
// device, so the state will move to IDLE and once again to not IDLE due to IRQ
//
while (sSpiInformation.ulSpiState != eSPI_STATE_IDLE)
{
}

/* Loop until SPI busy */
while (SPI_I2S_GetFlagStatus(CC3000_SPI, SPI_I2S_FLAG_BSY ) != RESET)
{
}
```

WIFI_INT (read) ->

```
while (!tSLInformation.ReadWlanInterruptPin())
{
}

sSpiInformation.ulSpiState = eSPI_STATE_WRITE_IRQ;
sSpiInformation.pTxPacket = pUserBuffer;
sSpiInformation.usTxPacketLength = usLength;
```

This is also wrong and should be a wait while High(H). But only if it was H before the CS asserted. The speed of the proc vs TI sensing CS going low is the only reason there are few hangs here. But run any other ISR that fires at -> below and the code will hang!

```
//
// Assert the CS line and wait till IRQ line is active and then initialize write operation
```

```
                //
                ASSERT_CS();   ------__________
        ->              ----------__________
                while (!tSLInformation.ReadWlanInterruptPin())S/B While high
                {
                }

                //
                // Due to the fact that we are currently implementing a blocking situation
```

2. A return from the Callout **SPIRxHandler** from **SpiTriggerRxProcessing** with no call to SPIResumeSpi() This happened on duplicate send() result on a closed socket. The first returned success, the seconds returned failure for the same send()! This left the TI never to be heard from again.

Solution is: Set a flag in SpiWrite (`tSLInformation.solicitedResponse`) after the write is committed to and before the IO that results in DMA ISR

Then hci_unsolicited_event_handler(void) has been refactored:

```
//*****************************************************************************
//
//!  hci_unsolicited_event_handler
//!
//!  @param None
//!
//!  @return         ESUCCESS if successful, EFAIL if an error occurred
//!
//!  @brief          Parse the incoming unsolicited event packets and issues
//!                  corresponding event handler.
//
//*****************************************************************************
long
hci_unsolicited_event_handler(void)
{
               unsigned long   res = 0;
               int isEvent = 0;
               unsigned char *pucReceivedData;

               // Buffer has message in it
               if (tSLInformation.usEventOrDataReceived != 0)
               {
                               pucReceivedData = (tSLInformation.pucReceivedData);
                               isEvent = (*pucReceivedData == HCI_TYPE_EVNT);
                               // An event did hci_unsol_event_handler handle it?
                               res = isEvent && hci_unsol_event_handler((char *)pucReceivedData);
               }

               // Handled here Or if not there is no outstanding write (solicitedResponse)
               if (res || (isEvent && tSLInformation.solicitedResponse == 0) ) {
            // There was an unsolicited event received - we can release the buffer
            // and clean the event received
                   if (!res)
                   {
                       ERROR("isEvent w/Opcode ==0  0x%02x 0x%02x 0x%02x 0x%02x 0x%02x 0x%02x 0x%02x 0x%02x 0x%02x 0x%02x",
                    pucReceivedData[0],pucReceivedData[1],pucReceivedData[2],pucReceivedData[3],
                    pucReceivedData[4],pucReceivedData[5],pucReceivedData[6],pucReceivedData[7],
                    pucReceivedData[8],pucReceivedData[9]);
                   }
            tSLInformation.usEventOrDataReceived = 0;
                   // Fire it up again
            SpiResumeSpi();
               }
               return res;
}
```

3. Ti code over writes memory when Receve is done. The transaction looks like

```
Host              cc3000
  |--- recv() ------->|
  |                   |
  |<---recv stat -----|
  |                   | Race is here
```

|<---recv data-----|

```
                    if (tSLInformation.usRxDataPending == 0) < fixes race we only return data to the Call that wanted it
                {
                    ERROR("type != HCI_TYPE_EVNT is (%d) usRxDataPending=%d usRxEventOpcode=%u usReceivedEventOpcode=%u",
                        *pucReceivedData,
                        tSLInformation.usRxDataPending,
                        tSLInformation.usRxEventOpcode,
                        usReceivedEventOpcode);
                }
                    else
                    {
                  pucReceivedParams = pucReceivedData;
                  STREAM_TO_UINT8((char *)pucReceivedData, HCI_PACKET_ARGSIZE_OFFSET, ucArgsize);

                  STREAM_TO_UINT16((char *)pucReceivedData, HCI_PACKET_LENGTH_OFFSET, usLength);

                  // Data received: note that the only case where from and from length
                  // are not null is in recv from, so fill the args accordingly
                  if (from)
                  {
                    STREAM_TO_UINT32((char *)(pucReceivedData + HCI_DATA_HEADER_SIZE), BSD_RECV_FROM_FROMLEN_OFFSET, *(unsigned long *)fromlen);
                    memcpy(from, (pucReceivedData + HCI_DATA_HEADER_SIZE + BSD_RECV_FROM_FROM_OFFSET) ,*fromlen);
                  }

                  // Let's vet length < This fixes the Bug #4
                  long length = usLength - ucArgsize;

                  if (length <= 0 || length > arraySize(wlan_rx_buffer))
                  {
                      // Not sane
                      length = -1;
                  }
                  else
                  {
                      memcpy(pRetParams, pucReceivedParams + HCI_DATA_HEADER_SIZE + ucArgsize, length);
                  }

                  // fixes the Nvram read not returning length
                  if (fromlen)
                  {
                    *fromlen = length; < This fixes the nvram API Bug #5
                  }


                  tSLInformation.usRxDataPending = 0;

                }

                            tSLInformation.usEventOrDataReceived = 0;

                            SpiResumeSpi();< if the cc3000 has reinterrupted  the second ISR will hit here

                            // Since we are going to TX - we need to handle this event after the
                            // ResumeSPi since we need interrupts
                            if ((*pucReceivedData == HCI_TYPE_EVNT) && (usReceivedEventOpcode == HCI_EVNT_PATCHES_REQ))
                            {
                                        hci_unsol_handle_patch_request((char *)pucReceivedData);
                            }

                            if ((tSLInformation.usRxEventOpcode == 0) && (tSLInformation.usRxDataPending == 0))
                            {
                                break;
                            }
                }
        }
    return NULL;

}
```

4. Memory overwrites memory for odd data from rxed from CC30000 when it is losing it's cookies. Solution is Above #4

Recived data look like:

```
0x20001F18  02 00 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 38 33 2E 38 38 34 2E 38  ..ÿ.....................................................83.884.8
```

memcpy(pRetParams, pucReceivedParams + HCI_DATA_HEADER_SIZE + ucArgsize, usLength - ucArgsize);

OK OK +104 5 255 0xffffff01

5. nvram_read() Docs say it returns length read.
Solution is Above #5

6. Hung in while(1) if any net op fails

Solution: Add time out API, use it in Set_NetApp_Timeout and will timeout any calls to SimpleLinkWaitEvent and SimpleLinkWaitData. Returning the appropriate error Value!

N.B I have been testing aucInactivity of 0 with a timeout of 8S for connect and 20 S for all other net ops it is working well for more then 3 days! It also had code to restart the CC3000 on CFOD detection.

Added In Spark_wan.c

```
uint32_t SPARK_WLAN_SetNetWatchDog(uint32_t timeOutInMS)
{
  uint32_t rv = cc3000__event_timeout_ms;
  cc3000__event_timeout_ms = timeOutInMS;
  return rv;
}

void Set_NetApp_Timeout(void)
{
            unsigned long aucDHCP = 14400;
            unsigned long aucARP = 3600;
            unsigned long aucKeepalive = 10;
            unsigned long aucInactivity = 32;
            SPARK_WLAN_SetNetWatchDog(S2M(aucInactivity)+250);
            netapp_timeout_values(&aucDHCP, &aucARP, &aucKeepalive, &aucInactivity);
}
```

Assumption is aucInactivity of 0 (infinite) we then use this to shorten the waits

```
  uint32_t ot = SPARK_WLAN_SetNetWatchDog(S2M(8))
  int rv = connect(sparkSocket, &tSocketAddr, sizeof(tSocketAddr));
  uint_SPARK_WLAN_SetNetWatchDog(ot);
```

Changes:

```
//*****************************************************************************
//
//!  hci_event_handler
//!
//!  @param  pRetParams     incoming data buffer
//!  @param  from           from information (in case of data received)
//!  @param  fromlen        from information length (in case of data received)
//!
//!  @return         none
//!
//!  @brief          Parse the incoming events packets and issues corresponding
//!                  event handler from global array of handlers pointers
//
//*****************************************************************************


unsigned char *
hci_event_handler(void *pRetParams, unsigned char *from, long *fromlen)
{
              unsigned char *pucReceivedData, ucArgsize;
              unsigned short usLength;
              unsigned char *pucReceivedParams;
              unsigned short usReceivedEventOpcode = 0;
              unsigned long retValue32;
              unsigned char * RecvParams;
```

```
unsigned char *RetParams;

        volatile int32_t start = millis();
        while (1)
        {
                        if (tSLInformation.usEventOrDataReceived == 0)
                        {
             volatile int32_t now = millis();
             volatile int32_t elapsed = now - start;
               if (elapsed < 0) { // Did we wrap
                  elapsed = start + now; // yes now
               }

                            if (cc3000__event_timeout_ms && (elapsed >= cc3000__event_timeout_ms))
```

< will timeout any calls

```
                            {
                        ERROR("Timeout now %ld start %ld elapsed %ld cc3000__event_timeout_ms
%ld",now,start,elapsed,cc3000__event_timeout_ms);
                        ERROR("Timeout waiting on tSLInformation.usRxEventOpcode 0x%04x",tSLInformation.usRxEventOpcode);

                             // Timeout Return Error for requested Opcode
                            // This sucks because callers should have initialized pucReceivedParams
                        switch(tSLInformation.usRxEventOpcode)
```

< will return the correct opcode

```
                        {

                        default:
                          INVALID_CASE(tSLInformation.usRxEventOpcode);
                          break;

                        case HCI_CMND_SIMPLE_LINK_START:
                        case HCI_CMND_READ_BUFFER_SIZE:
                          break;

                        case HCI_CMND_WLAN_CONFIGURE_PATCH:
                        case HCI_NETAPP_DHCP:
                        case HCI_NETAPP_PING_SEND:
                        case HCI_NETAPP_PING_STOP:
                        case HCI_NETAPP_ARP_FLUSH:
                        case HCI_NETAPP_SET_DEBUG_LEVEL:
                        case HCI_NETAPP_SET_TIMERS:
                        case HCI_EVNT_NVMEM_READ:
                        case HCI_EVNT_NVMEM_CREATE_ENTRY:
                        case HCI_CMND_NVMEM_WRITE_PATCH:
                        case HCI_NETAPP_PING_REPORT:
                        case HCI_EVNT_MDNS_ADVERTISE:
                        case HCI_EVNT_READ_SP_VERSION:
                        case HCI_EVNT_SELECT:
                          *(unsigned char *)pRetParams = -1;
                            break;

                        case HCI_CMND_SETSOCKOPT:
                        case HCI_CMND_WLAN_CONNECT:
                        case HCI_CMND_WLAN_IOCTL_STATUSGET:
                        case HCI_EVNT_WLAN_IOCTL_ADD_PROFILE:
                        case HCI_CMND_WLAN_IOCTL_DEL_PROFILE:
                        case HCI_CMND_WLAN_IOCTL_SET_CONNECTION_POLICY:
                        case HCI_CMND_WLAN_IOCTL_SET_SCANPARAM:
                        case HCI_CMND_WLAN_IOCTL_SIMPLE_CONFIG_START:
                        case HCI_CMND_WLAN_IOCTL_SIMPLE_CONFIG_STOP:
                        case HCI_CMND_WLAN_IOCTL_SIMPLE_CONFIG_SET_PREFIX:
                        case HCI_CMND_EVENT_MASK:
                        case HCI_EVNT_WLAN_DISCONNECT:
                        case HCI_EVNT_SOCKET:
                        case HCI_EVNT_BIND:
                        case HCI_CMND_LISTEN:
                        case HCI_EVNT_CLOSE_SOCKET:
                        case HCI_EVNT_CONNECT:
                        case HCI_EVNT_NVMEM_WRITE:
                        case HCI_EVNT_BSD_GETHOSTBYNAME:
                            *(int32_t *)pRetParams = -1;
                            break;


                        case HCI_EVNT_RECV:
                        case HCI_EVNT_RECVFROM:
                        case HCI_EVNT_ACCEPT:
```

```
 case HCI_EVNT_SEND:
 case HCI_EVNT_SENDTO:
 case HCI_CMND_GETSOCKOPT:
   *(int32_t *)pRetParams = -1;
   pRetParams += sizeof(int32_t );
   *(int32_t *)pRetParams = -1;
   break;

 case HCI_CMND_WLAN_IOCTL_GET_SCAN_RESULTS:
   *(int32_t *)pRetParams = 0;
   break;


 case HCI_NETAPP_IPCONFIG:
   memset(pRetParams,0,sizeof(tNetappIpconfigRetArgs));
   break;
 }
 break; // Exit Loop
            }

}
```