TAP

Technical

Reference

TheLab.ms
Plano, Texas
August 5, 2016, Preliminary

Table of Contents

The Display

The TAP contains eight rows each made up of eight pixels which in turn consists of three light emitting diodes, red, green and blue.

Electrically, the LEDs are arranged in a matrix of 24 columns by 8 rows. The columns are controlled by pins from the MSP430; a "1" on a pin grounds its corresponding column. The rows are addressed using a shift register; a single "0" is loaded when starting a scan and the shift register clocked once to send +5V to each subsequent row.

A display buffer consists of an array of the desired intensities of the LEDs, each one (unsigned char) ranging from 0 to 255 (FULL_INTENSITY.) Row 1 appears after row 0; column 1 appears after column 0 within its row and the individual colors are in the order red, green then blue. Generally, only a single display buffer is needed, namely display_buffer. Constants are defined to facilitate direct access to the intensities. For example, the green LED of column 3 on row 2 (all zero based) can be addressed as follows: display_buffer[ROW2+COL3+GREEN].

The scanning engine is conceptually very simple. Each time scan_LEDs is called, it performs one scan of the LEDs, using the intensity values in the display buffer to turn on each LED for an appropriate amount of time. A global variable, display_addr, contains the address of the currently active display buffer. This added level of indirection makes it possible to maintain multiple display buffers and switch between them very quickly. While the current firmware does not exploit this capability, the flexibility opens up the possibility of supporting animation or even simple video.

A macro is defined in tap.h for turning each column on or off. Each name is in the form:

COLxy_ON or COLxy_OFF

where

x = the pixel column from 0 to 7

y = the color element within the pixel, R, G or B for red, green or blue

For example, the statement COL1G_ON; turns on the column containing the green LED of the second pixel from the left.

For maximum brightness, the scanning code is structured so that as much as possible of the overhead of scanning occurs when LEDs are on because the cumulative time when no LED is lit takes away from the overall display intensity.

We are not polling the hardware except for the USB subsystem. Scanning the LEDs is the foreground task. We are not wasting any CPU cycles by not using interrupts.

Each LED is allotted a fixed time slice of 1/192 of the total available time during each scanning pass. If an LED is to be lit at full intensity, it is turned on for its entire time slice. For a zero or partial intensity, the remainder of the time slice is absorbed by a software delay loop; that time is not given to any other LED so that the intensity of any particular LED is not affected by the state of the others.

The high speed of the MSP430 microcontroller allows support of a full 256 levels of intensity for each color without visible flicker.

If we had wished to run the MSP430 in its lowest power mode, we could have used interrupts to wake up a sleeping controller when it is time to turn an LED on or off. But the amount of power saved by running the microcontroller in low power mode is very small in comparison to the current used by the lit LEDs, so the added complexity is not justified.

Bright Mode Versus Low-Power Mode

Our original intent was to light no more than one LED at a time for the lowest power consumption. As we began working with our engineering prototypes, we quickly realized that a brighter display is sometimes preferable despite greater current draw. Bright mode handles each pixel (all three of its constituent red, green and blue LEDs) at the same time, making the TAP roughly three times brighter. Instead of a static configuration at code build time, we chose to make this feature software selectable.

Handling even more LEDs simultaneously is possible, but at the cost of much greater software complexity. The general approach is to sort the applicable intensity values, turn on LEDs as needed, then turn them off individually from lowest to highest intensity.

USB Support

Development of the TAP firmware began before we knew anything about the USB sample code from Texas Instruments.

We chose to use their Local Echo Demo project as the starting point for our USB interface since we are essentially receiving a series of bytes from the host computer and doing something with it. The two code bases were spliced together by using the USB initialization code and calling the TAP code from the USB main() to initialize our specialized hardware, periodically scan the display and process data packets received from the host.

The combination of the two components is too large to be built using the free version of Code Composer, so we provided a means to build a version of the firmware sans USB support.

To configure for building without the USB subsystem, edit tapif.h to comment out the line

#define USB_SUPPORT

and also right-click on and Exclude from Build the directories USB_API, USB_app and USB_config in the Project Explorer.

To reverse the process, ensure that

#define USB_SUPPORT

is not commented out in tapif.h and also that none of the directories USB_API, USB_app and USB_config are Excluded from Build in the Project Explorer.

Serial Ports

The MSP430F5510 microcontroller contains two serial input and output ports.

TAP units have been designed to be plugged together to form a larger display panel. The B input port receives data from the adjacent unit to the left; the B output port sends data to the adjacent unit to the right. The A input port receives data from the adjacent unit below; the A output port sends data to the adjacent unit above.

Communications is in one direction only; up and to the right from the unit in the lower left corner. There is no provision to send information to the unit below or to the left.

We currently initialize the serial ports for 115200 baud. The hardware is capable of much higher data rates should it be needed for demanding amounts of binary image data.

Packet Forwarding

In order to form a large display panel from individual TAP units, there must be some means of communication among them and targeting an individual unit. The firmware has been designed to support assigning each unit a unique ID so that it can be directly addressed. For this mechanism to work, each TAP must pass packets through its serial ports. To complicate matters, an incoming packet (from either port) needs to be relayed out both ports.

To prevent a problem of cascading duplicate packets, if the same packet is received in both ports, it should be passed on only once. How to efficiently do this comparison of packets? A CRC is calculated on each packet relayed and incoming packets have their CRC computed and compared. A list of the CRCs of the most recent N packets is kept for this purpose. N should not need to be a very large number.

Learn Mode

The command packet definition has provision to direct information to a single specific unit when several are connected together. Learn Mode is how to set this up.

Generally, the unit in the lower left corner of the arrangement is assigned the address (0,0). The one to its right is (0,1), then (0,2), and so on. The unit above it is (1,0) and the units to that one's right is (1,1) and so on. Assigning an address to any particular unit triggers Learn Mode for all units above and to the right of that unit.

By default, no address ID is set and the firmware responds to all of the packets it receives.

When Learn Mode is triggered for a unit, it assigns itself the specified address. It informs the unit to its right that its address is (y,x+1) and the one above, (y+1,x). This process ripples through the connected units rightward and upward.

Once Learn Mode is complete, each individual unit will respond only to command packets bearing its address in bytes 3 and 4 of the packet or if the ignore address flag is set.

Scrolling Text Synchronization

A special provision has been implemented in the standard firmware to allow easily making a large scrolling text marquee out of multiple TAP units.

First, download the same text string to all of the units. Then connect them together in a horizontal row. When this assembly is powered up, that text string is scrolled across all of the units as if they were a single long display.

How this is done involves a bit of sleight of hand.

When an individual TAP is powered up, it does not know whether it is alone or connected to others. And in the latter case, its position in relation to the others or how many there are.

Each one begins by assuming that there is one unit attached to its right. It asserts "ownership" of that unit. After a short delay, only the unit on the left has not been "owned" and it now knows that. The "master" then informs the unit to its right that it is second which in turn informs the next one that it is third. And so on down the line.

After another short delay, the master sends a packet down the chain to start scrolling. The first unit starts with the first character in the string; the second with the second; and so on.

Data Organization

Marinara is the central database of state information. It is of the type burgermaster; its fields are:

```
typedef struct burgermaster
{
	char		tag[3];
	unsigned char	flags;
	unsigned char	x_address;
	unsigned char	y_address;
	char		*data_address;
	char		*font_address;

} burgermaster;
```

tag is not used in the memory-resident copy of the database.

flags contains several small items of information.

> BRIGHT_MODE (0x80) = 1 if bright mode is set
> ID_ASSIGNED (0x40) = 1 if an ID address has been assigned
>
> The low two bits indicate the type of data downloaded:
>
> DOWNLOADED_NONE (0x00)
> DOWNLOADED_TEXT (0x01)
> DOWNLOADED_IMAGE (0x02)

x_address is the zero-based column number of this TAP unit if the ID_ASSIGNED flag is set.

y_address is the zero-based row number of this TAP unit if the ID_ASSIGNED flag is set.

data_address is the address of the downloaded data or 0xFFFF if none. The DOWNLOADED_TEXT or DOWNLOADED_IMAGE flags identifies the type of the data.

font_address is the address of the downloaded font table or 0xFFFF if none.

A text fragment consists of a header followed immediately by the text.

```
typedef struct text_fragment
{
	unsigned char font_and_flags;
	unsigned char red_intensity;
	unsigned char green_intensity;
	unsigned char blue_intensity;
	char          *next_address;
	unsigned char num_chars;
	unsigned char text[0];

} text_fragment;
```

flags contains several small items of information.

SUPPRESS_TEXT_SYNC (0x80) = 1 if
DISPLAY_WHOLE_CHARS (0x40) = 1 if

The low order bit is 0 if the text uses the built-in extended ASCII font; it uses one byte per character. A 1 indicates Unicode text; it uses two bytes per character, high order byte first. The next two lowest bits are set to 01 to indicate a text fragment.

Red green and blue intensity are the constituent parts of the text color.

next_address is the address of the next text fragment or 0xFFFF if none.

num_chars is the number of characters in the text fragment, not necessarily the number of bytes.

text is the contents of the fragment. It is not zero terminated.

```
typedef struct binary_image
{
	unsigned char flags;
	unsigned char num;
	unsigned char intensities[0];

} binary_image;
```

flags is for future expansion. It is currently set to 6 to indicate a binary image.

num indicates how many intensity values follow.

The intensities array is in the same layout as the display buffer: red followed by green, then blue.

Font lookup table format very TBD at this time.

Flash Memory Organization

Several fields of the burgermaster gain significance when stored in flash memory.

A shadow copy of the burgermaster is kept so that temporary settings are not written into flash memory.

tag consists of the letters “TAP” so that the firmware can detect that it had written the data.

Pointers stored in flash memory have the value 0xFFFF when unused because the erased state of flash memory is all “1”s and the pointer can be updated with an actual address without having to erase the memory again.

__TI_CINIT_Limit

Writing a Demo

The firmware has been designed primarily to implement the basic functionality of scanning the display and handling the download of text and images from a host computer or the upstream TAP unit(s). However, also providing an easy way to implement standalone demonstrations and applications was a strong secondary consideration.

The demo interface is modeled loosely after the well-known organization used by sketches in the popular Arduino and Energia development environments.

At the highest level, a demo consists of two functions:

void setup(void);
void loop(void);

setup is called once as part of the power-up reset sequence to allow the demo to initialize any needed data.

loop is called once before each scan of the LEDs to give the demo an opportunity to alter the display. Because the scan rate is fairly quick, keeping count and skipping a number of frames will generally be desired between modifications of the display contents.

The demo interface is activated by putting code into demo.h and rebuilding. The provided demo.none is an empty file; copy it to demo.h to build the basic firmware. The file demo.blank is a skeleton for writing a demo from scratch. It may be easier to take one of the sample demo files and modify it.

Much of the modifications to the display buffer will be done with direct access to the values. For example, to make the top right pixel green,

```
display_buffer[ROW0+COL7+RED] = 0;
display_buffer[ROW0+COL7+GREEN] = FULL_INTENSITY;
display_buffer[ROW0+COL7+BLUE] = 0;
```

Note that the addressing indices are zero-based. This is C, after all…

There is not much of an Application Program Interface to the firmware, but several internal routines may be useful for demo and application developers.

```
/*
 * fillRect ****************************************************
 *
 * Fill a rectangle in the display buffer with the specified color
 *
 * buffer = the address of the display buffer
 * x = the left edge of the rectangle
 * y = the top edge of the rectangle
 * width = the width of the rectangle
 * height = the height of the rectangle
 * red = the intensity of the red part of the color
 * green = the intensity of the green part of the color
 * blue = the intensity of the blue part of the color
 *
 */
void fillRect(unsigned char *buffer,
              int x,
              int y,
              int width,
              int height,
              unsigned char red,
              unsigned char green,
              unsigned char blue)

/*
 * bltChar *****************************************************
 *
 * Block transfer a character glyph to the display buffer
 *
 * buffer = the address of the display buffer
 * glyph = the bits for the character
 * origin = origin column of the character
 * clear_background = 0 if background color is rendered
 *
 * Globals
 * txt_red = foreground red intensity
 * txt_green = foreground green intensity
 * txt_blue = foreground blue intensity
 * bg_red = background red intensity
 * bg_green = background green intensity
 * bg_blue = background blue intensity
 *
 */
void bltChar(unsigned char *buffer, const unsigned char *glyph, int origin,
       int clear_background)



/*
 ************ find_glyph *****************************************************
 *
 * Find the glyph representation for a character
 *
 * ch = the character
 *
 */
const unsigned char * find_glyph(unsigned int ch)
```

```
/*
 ************ create_text_fragment ******************************************
 *
 * Create a text fragment
 *
 * buffer = the buffer to contain the fragment
 * flags = flags and font for the fragment
 * red = red intensity
 * green = green intensity
 * blue = blue intensity
 * length = the length of the text string
 * s = the text string to put into the fragment
 *
 */
void create_text_fragment(unsigned char *buffer,
                          unsigned char flags,
                          unsigned char red,
                          unsigned char green,
                          unsigned char blue,
                          unsigned int  length,
                          const unsigned char *s)


/*
 ************ start_scrolling_text ******************************************
 *
 * Start scrolling text
 *
 * fragment = the first text fragment to scroll
 *
 */
void start_scrolling_text(text_fragment *fragment)
```

Example 1: Red White And Blue

```
/*
 * demo.RedWhiteAndBlue
 *
 * This demo cycles the display between red, white and blue.
 *
 */

#ifndef DEMO_H_
#define DEMO_H_

#define      DEMO

#define      HOLD_FRAMES   30

int    color;
int    numFrames;

void setup(void)
{
       // This code is invoked once to initialize the demo...
       numFrames = 0;
       color = 3;
}

void loop(void)
{
       // This code is invoked once each time the LEDs are scanned...
       if (numFrames > 0)
              {
              // Hold current image for some frames
              numFrames--;
              return;
              }
       numFrames = HOLD_FRAMES;

       // Go to next color
       if (color >= 2)
              color = 0;
       else
              color++;

       switch (color)
              {
              case 0:      // Red
                     fillRect(display_addr, 0, 0, 8, 8, FULL_INTENSITY, 0, 0);
                     break;
              case 1:      // White
                     fillRect(display_addr, 0, 0, 8, 8, FULL_INTENSITY,
                                   FULL_INTENSITY, FULL_INTENSITY);
                     break;
              case 2:      // Blue
                     fillRect(display_addr, 0, 0, 8, 8, 0, 0, FULL_INTENSITY);
                     break;
              }
}

#endif /* DEMO_H_ */
```

Example 2: Exes And Ohs

```
/*
 * demo.ExesAndOhs
 *
 * This demo cycles the display between an X and an O using two video buffers.
 *
 */

#ifndef DEMO_H_
#define DEMO_H_

#define      DEMO

#define      HOLD_FRAMES  30

int    letter;
int    numFrames;

unsigned char display_buffer2[8 * 8 * 3];

void setup(void)
{
       // This code is invoked once to initialize the demo...
       numFrames = 0;
       letter = 1;

       txt_red = 0;
       txt_green = FULL_INTENSITY;
       txt_blue = 0;
       bg_red = 0;
       bg_green = 0;
       bg_blue = 0;

       bltChar(display_buffer, find_glyph('X'), 0, 0);
       bltChar(display_buffer2, find_glyph('O'), 0, 0);
}

void loop(void)
{
       // This code is invoked once each time the LEDs are scanned...
       if (numFrames > 0)
              {
              // Hold current image for some frames
              numFrames--;
              return;
              }
       numFrames = HOLD_FRAMES;

       // Go to next letter
       if (letter >= 1)
              letter = 0;
       else
              letter++;

       if (letter == 0)
              display_addr = display_buffer;
       else
              display_addr = display_buffer2;
}

#endif /* DEMO_H_ */
```

Example 3: Red Green and Blue

```
/*
 * demo.RedGreenBlue
 *
 * This demo displays text in multiple colors.
 *
 */

#ifndef DEMO_H_
#define DEMO_H_

#define      DEMO

int    first_time;

void setup(void)
{
       // This code is invoked once to initialize the demo...
       text_fragment *txt;

       create_text_fragment(temp_text, 0, FULL_INTENSITY, 0, 0,
                     strlen("Red"), "Red");

       create_text_fragment(temp_text + 20, 0, 0, FULL_INTENSITY, 0,
                     strlen("Green"), "Green");
       txt = (text_fragment *)temp_text;
       txt->next_address = (text_fragment *)(temp_text + 20);

       create_text_fragment(temp_text + 40, 0, 0, 0, FULL_INTENSITY,
                     strlen("Blue"), "Blue");
       txt = (text_fragment *)(temp_text + 20);
       txt->next_address = (text_fragment *)(temp_text + 40);

       first_time = 1;
}

void loop(void)
{
       // This code is invoked once each time the LEDs are scanned...
       if (first_time)
              {
              // Transition to scrolling text
              start_scrolling_text((text_fragment *)temp_text);

              first_time = 0;
              }
}

#endif /* DEMO_H_ */
```

Example 4: id

```
/*
 * demo.id
 *
 * This demo displays the current ID address.
 *
 */

#ifndef DEMO_H_
#define DEMO_H_

#define      DEMO

#define      HOLD_FRAMES  5

int                        numFrames;
unsigned char final[8];

const unsigned char slash[8] =
       { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };

const unsigned char dashes[8] =
{ 0x00, 0x00, 0x07, 0x00, 0x00, 0xE0, 0x00, 0x00 };

const unsigned char y[10][8] =
       {
                     { 0x02, 0x05, 0x05, 0x05, 0x02, 0x00, 0x00, 0x00 },   // '0'
                     { 0x02, 0x03, 0x02, 0x02, 0x07, 0x00, 0x00, 0x00 },   // '1'
                     { 0x03, 0x04, 0x02, 0x01, 0x07, 0x00, 0x00, 0x00 },   // '2'
                     { 0x03, 0x04, 0x03, 0x04, 0x03, 0x00, 0x00, 0x00 },   // '3'
                     { 0x05, 0x05, 0x07, 0x04, 0x04, 0x00, 0x00, 0x00 },   // '4'
                     { 0x07, 0x01, 0x03, 0x04, 0x03, 0x00, 0x00, 0x00 },   // '5'
                     { 0x02, 0x01, 0x03, 0x05, 0x02, 0x00, 0x00, 0x00 },   // '6'
                     { 0x07, 0x04, 0x02, 0x02, 0x02, 0x00, 0x00, 0x00 },   // '7'
                     { 0x02, 0x05, 0x02, 0x05, 0x02, 0x00, 0x00, 0x00 },   // '8'
                     { 0x02, 0x05, 0x06, 0x04, 0x02, 0x00, 0x00, 0x00 }    // '9'
       };

const unsigned char x[10][8] =
       {
                     { 0x00, 0x00, 0x00, 0x40, 0xA0, 0xA0, 0xA0, 0x40 },   // '0'
                     { 0x00, 0x00, 0x00, 0x40, 0x60, 0x40, 0x40, 0xE0 },   // '1'
                     { 0x00, 0x00, 0x00, 0x60, 0x80, 0x40, 0x20, 0xE0 },   // '2'
                     { 0x00, 0x00, 0x00, 0x60, 0x80, 0x60, 0x80, 0x60 },   // '3'
                     { 0x00, 0x00, 0x00, 0xA0, 0xA0, 0xE0, 0x80, 0x80 },   // '4'
                     { 0x00, 0x00, 0x00, 0xE0, 0x20, 0x60, 0x80, 0x60 },   // '5'
                     { 0x00, 0x00, 0x00, 0x40, 0x20, 0x60, 0xA0, 0x60 },   // '6'
                     { 0x00, 0x00, 0x00, 0xE0, 0x80, 0x40, 0x40, 0x40 },   // '7'
                     { 0x00, 0x00, 0x00, 0x40, 0xA0, 0x40, 0xA0, 0x40 },   // '8'
                     { 0x00, 0x00, 0x00, 0x40, 0xA0, 0xC0, 0x80, 0x40 }    // '9'
       };

void setup(void)
{
       // This code is invoked once to initialize the demo...

       bg_red = 0;
       bg_green = 0;
       bg_blue = 0;
}
```

```
void loop(void)
{
	// This code is invoked once each time the LEDs are scanned...

	if (numFrames > 0)
		{
		// Hold current image for some frames
		numFrames--;
		return;
		}
	numFrames = HOLD_FRAMES;

	// Clear the display
	fillRect(display_buffer, 0, 0, 8, 8, 0, 0, 0);

	// Render slash
	txt_red = EIGHTH_INTENSITY;
	txt_green = EIGHTH_INTENSITY;
	txt_blue = EIGHTH_INTENSITY;
	bltChar(display_buffer, slash, 0, 0);

	// Implant X and Y addresses
	if (Marinara.flags & ID_ASSIGNED)
		{
		// Show ID in green
		txt_red = 0;
		txt_green = FULL_INTENSITY;
		txt_blue = 0;

		bltChar(display_buffer, x[Marinara.x_address], 0, 1);
		bltChar(display_buffer, y[Marinara.y_address], 0, 1);
		}
	else
		{
		// Show unassigned state in yellow
		txt_red = FULL_INTENSITY;
		txt_green = FULL_INTENSITY;
		txt_blue = 0;

		bltChar(display_buffer, dashes, 0, 1);
		}

	txt_red = FULL_INTENSITY;
	txt_green = FULL_INTENSITY;
	txt_blue = FULL_INTENSITY;
}

#endif /* DEMO_H_ */
```

Appendix A - Command Packet Format

General Packet Layout

Byte 0 - Packet type (upper bits are treated as flags)

Bit 0x80 - set to ignore the address. That is, the packet is honored by every unit which receives it despite ID address mismatch. See Learn Mode for more information.

Bit 0x40 - set to inhibit forwarding the packet to other TAP units. This is used for direct communication from a TAP to one of its immediate downstream neighbors.

Bit 0x20 – set to inhibit writing this information to flash memory. This is primarily for use when a TAP is used as a display device with frequent updates. Skipping the write to flash memory saves the time needed to do the write as well as prolonging the life of the flash memory. Information sent with this flag set is lost when power is removed.

Byte 1 - High byte of packet size (including all of packet header starting with byte 0)
Byte 2 - Low byte of packet size (including all of packet header starting with byte 0)
Byte 3 - ID X address, 0 is the bottom row
Byte 4 - ID Y address, 0 is the leftmost column
:
: Optional additional packet contents depending upon packet type
:

Packet Type 0x10: Reset to Factory Defaults

Resets the TAP to factory default settings. Low-power mode. No downloaded text, bitmap or fonts. No X, Y address assigned.

no additional data bytes

Packet Type 0x11: Set Brightness Mode

Set the TAP brightness mode. In low-power mode, no more than one LED is lit at any time. The bright mode is approximately three times brighter than low-power mode, at the cost of around double the power consumption.

Byte 5 – Brightness mode, 0 for low power mode
1 for bright mode
other values reserved for future expansion

Note: legacy usage - when packet length is 5, set bright mode.

Packet Type 0x12: Reserved for user expansion

This packet type is specifically set aside for the end user who desires to modify the firmware to define their own packet types.

Byte 5 – packet subtype, user defined

Note: legacy usage - when packet length is 5, set low power mode.

Packet Type 0x13: Set ID

Set the TAP address when multiple units are connected - the contents of X and Y address bytes (bytes 3 and 4) become the new address for this unit. The ignore address flag should be set on this packet in order to override any previous address assignment. A packet with the X address incremented is sent to the next horizontal unit and one with Y incremented to the next vertical one. Learn mode is implemented using this packet. The X and Y address fields of packets are ignored until an ID is set.

If the inhibit writing to flash memory flag is set in the incoming packet, it must be propagated in the generated packets so that temporary IDs may be assigned which are forgotten when power is removed.

no additional data bytes

Packet Type 0x14: Text String

Download a text string. Any individual text string may contain up to 246 characters of ASCII text or 123 Unicode characters. More than one text string may be downloaded; each may use different colors and fonts. They will be displayed in the order they are received. The total amount of text is limited only by the amount of flash memory in the TAP. Downloading a text string overrides any previously downloaded binary image.

Byte 5 - Red color level, 0..0xFF
Byte 6 - Green color level, 0..0xFF
Byte 7 - Blue color level, 0..0xFF
Byte 8 - font #, 0 is the built-in extended ASCII font, one byte per character; 1 is the downloaded font, two bytes per character, high order byte first.

(the upper bits of the font number are treated as flags and the flags on the first text string governs the behavior of all of them)

Bit 0x80 - set to suppress automatic text scrolling synchronization (see packet types 0x15, 0x16 and 0x17)

Bit 0x40 - set to display the string a single character at a time instead of scrolling

Byte 9 - First text character
:
: etc - text string is not 0 terminated
:

Packet Type 0x15: Text Sync Establish

Sent by each TAP to the next horizontally downstream unit to assert ownership of the text sync stream as part of power-up initialization and before displaying text. To prevent an unnecessary avalanche, this packet is not passed down the chain (0x40 bit must be set;) each TAP is expected to establish dominion over its immediate horizontal downstream neighbor in the same way. The final state is that only the leftmost TAP is the master.

no additional data bytes

Packet Type 0x16: Text Sync Origin

Sent to the downstream unit by the leftmost TAP (the one not owned during the establish phase - see description for packet type 0x15) via the horizontal serial port after sync assert and before text display begins. This packet is not passed down the chain (0x40 bit must be set,) but each TAP stores its assigned index, increments the character index in the packet and passes that on. The final state is that the origins of the connected TAPs from left to right are 0, 1, 2, etc. Each TAP is responsible for performing a modulus operation on its assigned origin by the length of the text string(s) plus one and setting the first_char state to account for the synthesized leading space in case of wrap around (text string shorter than the number of connected TAPs.)

Byte 5 - Index of the character to display upon receipt of the mark (see packet type 0x17.)

Packet Type 0x17: Text Sync Mark

Sent by the leftmost TAP to the downstream unit via the horizontal serial port just before the first character of a text string is displayed. Receipt of this packet while in text string display mode resumes text scrolling by displaying the origin character; this automatically synchronizes a series of TAPs arranged horizontally and containing the same downloaded text string. This synchronization should be performed periodically to account for differences in the clock frequencies between units.

no additional data bytes

Packet Type 0x18: Binary Image

Download a bitmapped image. A partial image may be downloaded by specifying a packet containing fewer than 192 values; in reality, this is only useful for N rows where N is less than 8. Pixel numbering is from the upper left corner of the TAP. Downloading a binary image overrides any previously downloaded image or text strings.

Byte 05 - Red color level of pixel (0,0), 0..0xFF
Byte 06 - Green color level of pixel (0,0), 0..0xFF
Byte 07 - Blue color level of pixel (0,0), 0..0xFF
Byte 08 - Red color level of pixel (0,1), 0..0xFF
Byte 09 - Green color level of pixel (0,1), 0..0xFF
Byte 0A - Blue color level of pixel (0,1), 0..0xFF
:
: etc...
:
Byte C2 - Red color level of pixel (7,7), 0..0xFF
Byte C3 - Green color level of pixel (7,7), 0..0xFF
Byte C4 - Blue color level of pixel (7,7), 0..0xFF

Packet Type 0x19: Clear Downloaded Data

Clears all downloaded text strings and images. This is not as drastic as a factory reset (0x10) as it preserves downloaded fonts. Because more than one text string may be downloaded, this is the way to delete previously downloaded strings.

no additional data bytes

Packet Type 0x1A: Download Font Glyph

Download a glyph for a single two-byte character. A character code with a zero upper byte replaces the appearance of a character in the built-in font. The low-order bit of each byte is the leftmost column of the glyph.

Byte 5 - High byte of character code
Byte 6 - Low byte of character code
Byte 7 - Row 0
Byte 8 - Row 1
Byte 9 - Row 2
Byte A - Row 3
Byte B - Row 4
Byte C - Row 5
Byte D - Row 6
Byte E - Row 7

Appendix B – Preparing for Development

First, you need tools to build the binary image from the source files. I use the Windows version of the Code Composer Studio from Texas Instruments.

The caveat is that the free version is code size limited and the USB enabled version of the firmware exceeds that limit. If you use the free version, see the section on USB Support for information about how to disable that functionality to fit within the size limit.

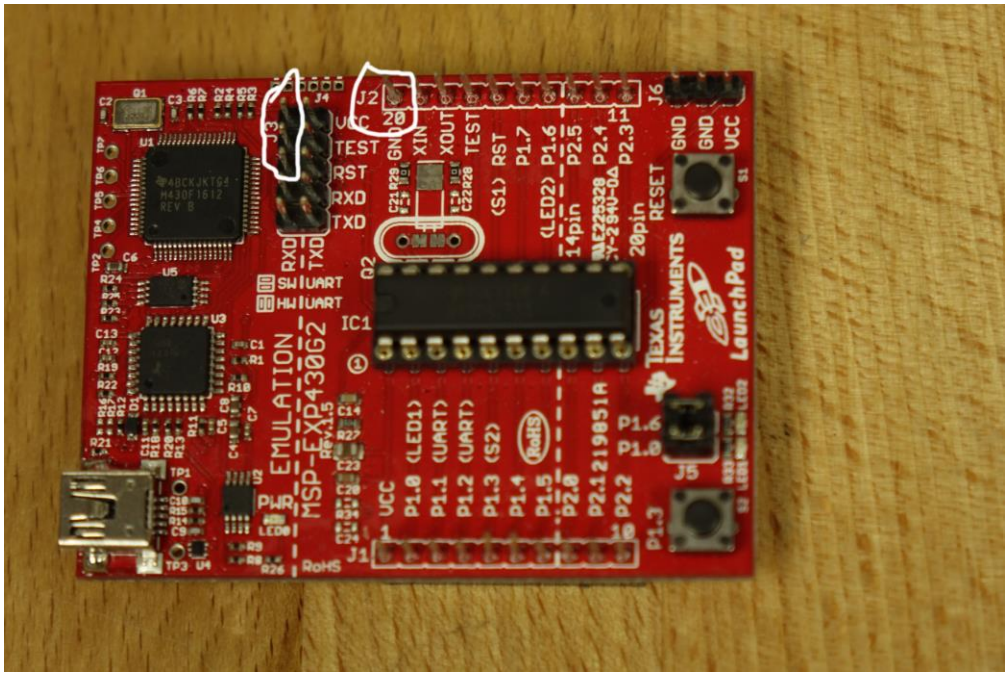
We have people working to build using GCC or Code Composer Studio Cloud. More on those as we figure them out.

Second, you need a way to get your code into the MSP430 on the TAP. I use the Value Line LaunchPad, TI part MSP-EXP430G2. When viewed with the USB connector on the left, the left third of the LaunchPad is the programmer/debugger – that is the part we are using. The rest of the LaunchPad is the MSP430 development board.

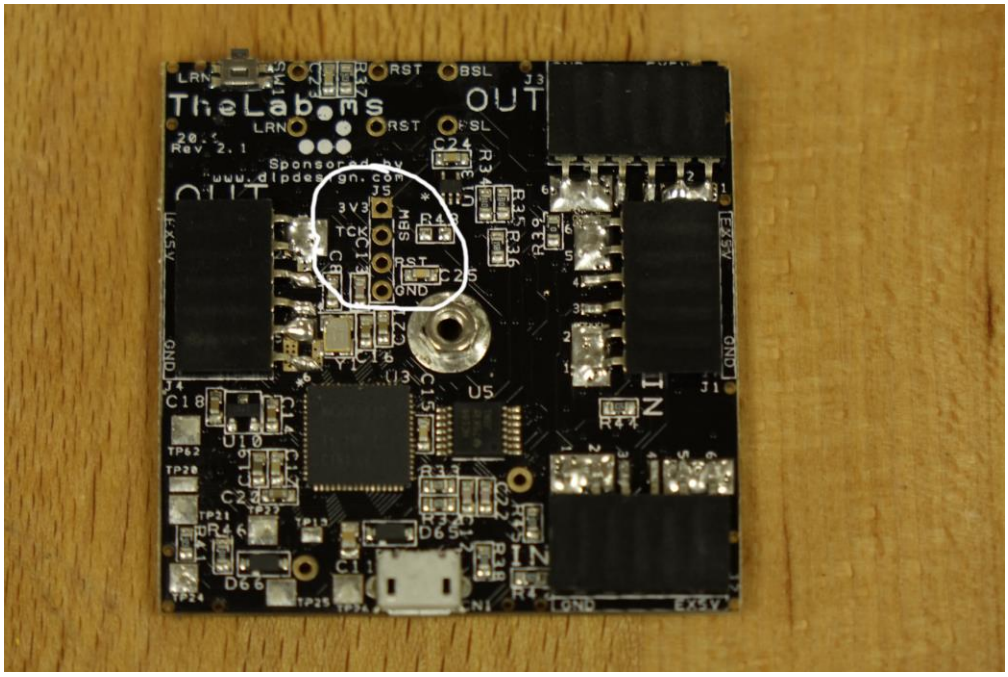
Remove the jumper blocks from the header bridging the two sections and save them for when you want to use the development board portion again.

Third, you need a cable to connect the TAP to the programmer/debugger on the LaunchPad.

Only four signals are needed for this connection. The required signals on the LaunchPad are VCC, TEST, RST and GND. Three are on the bridging header and GND is on a nearby row of pins.



The corresponding signals on the TAP are 3V3, TCK, RST and GND. When viewed on the back of the TAP with the USB connector on the bottom, these signals are the four pads in the upper-left quadrant.

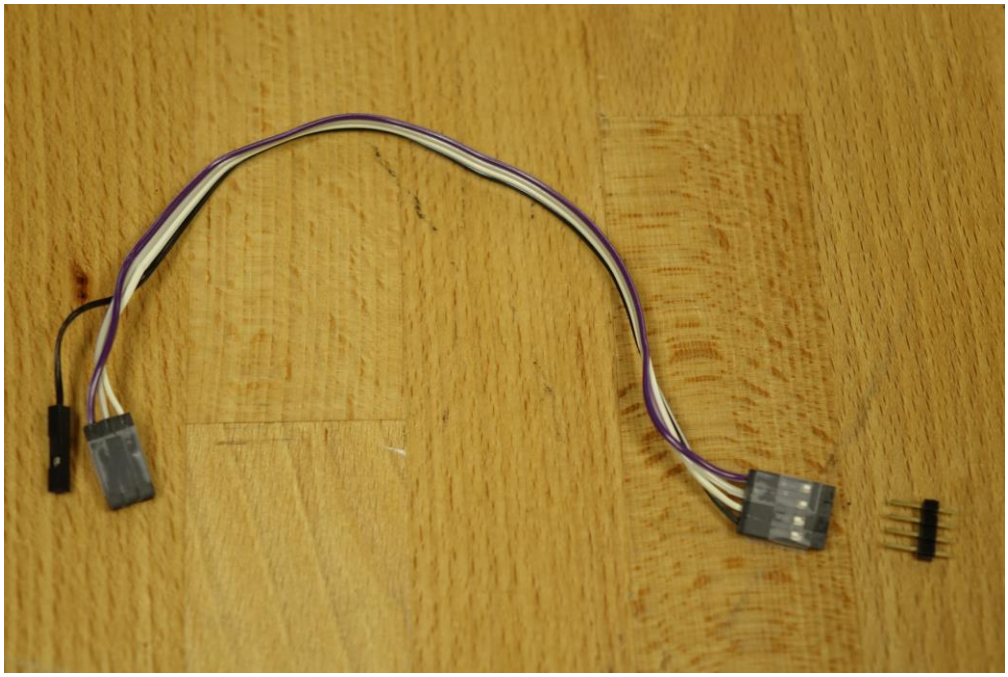


I built a cable using four conductors of jumper wire. I taped the VCC, TEST and RST sockets together on the LaunchPad end and all four together on the TAP end.

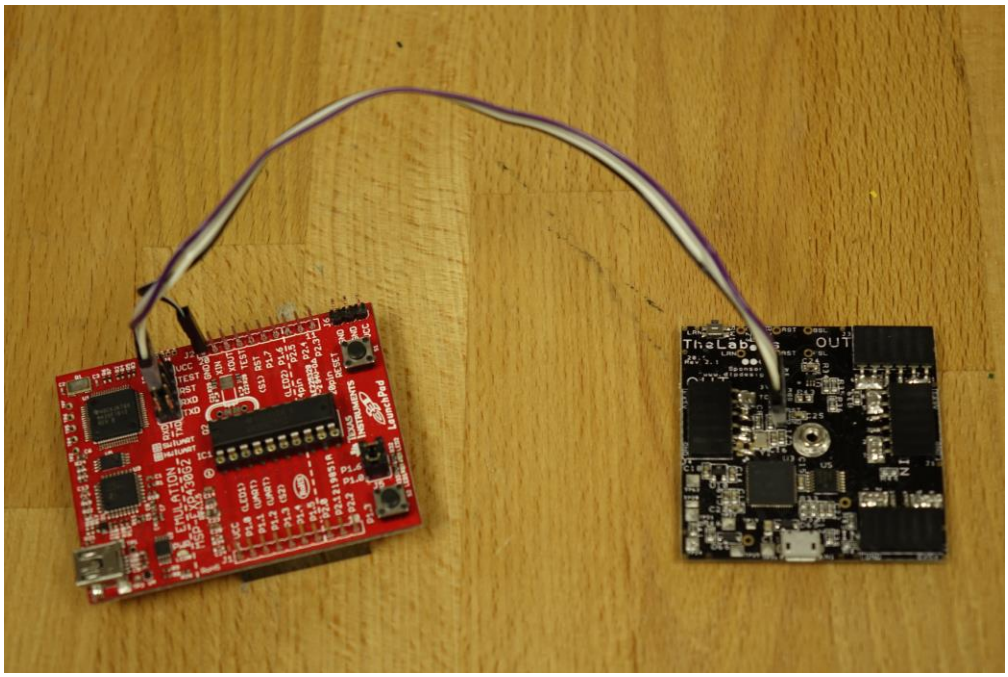
```
VCC  ------------------------------------------------------------------------------  3V3

TEST  -----------------------------------------------------------------------------  TCK

RST  ------------------------------------------------------------------------------  RST

                                  ------------------------------------------------  GND
                                /
                               /
                              /
GND  ------------------------/
```

It is very important that this connector be attached oriented correctly; I have accidentally plugged it in upside down once and let the magic smoke out of a voltage regulator on a TAP. Choose a color coding scheme and double check it every time you hook the cable up.

I strongly suggest that you solder a four-pin header to the pads on the TAP. For just programming the microcontroller, you can get away with plugging the header into the cable and holding it into the pads at an angle with slight pressure from a fingertip; a sturdy, hands-free connection is a must if you use the debugger.



And this is how it looks when ready to go…



Appendix C - Source Code Organization

main.c - the foundation of the firmware. It manages the power-up reset process, setting up the clock and initializing input and output ports. It is based upon the file provided by the USB local echo sample code from Texas Instruments.

system_pre_init.c – part of the USB sample code. This mostly serves to disable the watchdog timer until application initialization is complete.

driverlib\ - the device support library from Texas Instruments. It is heavily used by the USB code, but is generally beneficial for applications as well.

USB_API\ - the bulk of the USB support library.

USB_app\ – the place recommended to put files for a USB application. We chose to keep using the root directory.

USB_config\ - files to configure the USB support library to the hardware.

hal.h – definitions to configure the USB support library.

hal.c – routines to initialize the run-time environment for the USB support library.

tapif.h – definitions to interface the TAP firmware with the outside world, in particular, the USB support library.

tap.h – the bulk of the definitions for the TAP firmware.

tap.c – the bulk of the TAP firmware.

tappacket.h – the definitions for the command packet structure. This file is intended to also be used for building the host application on the PC.

font_ascii.c – contains the character glyphs for the built-in extended ASCII font.

demo.h – contains the code and data for a standalone demo or application.

demo.blank – an empty file for creating a demo or application.

demo.* - an assortment of the demonstration applications we developed to show the capabilities of the TAP. Copy the desired one to demo.h and rebuild. Copy demo.none to demo.h to rebuild the core firmware.

lnk_msp430f5510.cmd – the link control file.

Appendix D – Effect of Compiler Settings

We have discovered that optimizing aggressively for size (as well as not optimizing at all) has an adverse effect on the frame rate.

The size and frame rate varied over the several compiler optimization level and size versus speed settings in the Texas Instruments Code Composer as shown in this table:

Size --->	0	1	2	3	4	5	<--- Speed
Opt lvl							
off	23824 37	23824 37	23714 37	23710 37	23710 37	23710 37	
0	18956 >60	18956 >60	18854 >60	18854 >60	18908 >60	18908 >60	
1	17696 >60	17696 >60	17594 >60	17646 >60	17704 >60	17704 >60	
2	15716 <60	15790 60	16746 >60	17382 >60	17578 >60	18610 >60	
3	19970 >24	20106 <26	20760 48	21072 <61	21226 <61	22354 <61	
4	18472 24	18602 <26	19248 48	19548 <61	19734 <61	20862 <61	

A good compromise is setting 2 on both.