1 Overview

This document describes a set of demonstration software used to display a set of odometers on the display of a Neon® board along with accompanying images, animation, and videos. The resulting application (`odomTTY`) allows control of a set of on-screen odometers with display of static images and `MPEG 1/2` video.

The software is implemented on a Neon board to support resolutions of up to 1280x1024, although the initial target is a Hitachi `TX39D**VM1**` 15.4" WXGA panel.

2 Revision History

Date	Revision	Description
2006-08-20	1.0	First draft

3 Prerequisites

The Odometer software is built using GNU tools (Make, GCC) for Linux, generally through the use of a cross-compiler.

Refer to section 5.3 of the Neon® User's Manual for details on how to build a Linux-hosted cross compiler and a set of userland libraries.

In particular, the software requires Oliver Debon's GPL Flash Library, libPNG, and libMPEG2.

Section 5.3.7 of the Neon® User's Manual includes references to the Boundary Devices' userland build scripts for creating an entire root filesystem from scratch (er. sources).

Finally, because the SM-501 drivers were enhanced to support bitblts from video RAM, a kernel version from sources on or after August 15, 2006 is needed, along with a userland build on or after August 17, 2006.

Refer to the Boundary Devices' documentation page for the latest release notes.

4 About this document

This document discusses the source code structure from the device driver layer outwards toward the application to describe the more general-purpose pieces first.

- **Layer one** - the device driver entry points can be applied to any graphic-intensive multimedia application.
- **Layer two** - the wrapper modules may also be useful for speeding any application that needs to drive the `SM-501` graphics controller in an efficient manner.
- **Layer three** - the odometer modules may be used piece-wise without layers four and five, and are also useful examples of how to use the lower layers.

- **Layer four** - the animation and video layer is fairly dependent on the odometer modules, but can be used as an example of how to display images, animation, and video on a Neon board.
- **Layer five** - the application layer is completely dependent on the previous layers, but *could be extended* to contain a more useful program[1].

5 Layer one: from the inside out

Displaying images of digits is not a complicated task in itself, and wouldn't generally be worthy of the effort of documentation by itself. If the odometers must respond very smoothly while the CPU ranges from idle to fully loaded, things become more difficult. Doing all of this on a 1280x800 display on a Neon board requires some finesse.

Hence the document.

The primary mechanisms used to accomplish this are:

- Use the SM-501 graphics accelerator to perform blts.
- Use the SM-501's display buffer to buffer the images.
- Synchronize with the display refresh.
- Use POSIX real-time signals for display updates.

As you might imagine, the list above requires some help from the display device driver(s). The following subsections detail that support. You may want to look at the SM-501 data sheet to get a more complete understanding of what's being driven by these drivers.

5.1 Driver enhancement: Allocate video RAM

In prior releases of the SM-501 device driver for the Neon board, relatively little of the SM-501 device was exposed to userspace applications. The standard Linux frame-buffer interface was supported for efficient access to the display RAM, but no structure for the use of that memory was available.

The August 2006 release of the Boundary Devices kernel patches adds support for `/dev/sm501mem`, a device used to allocate and deallocate video RAM.

It does this through the `SM501_ALLOC` and `SM501_FREE` `ioctl()` calls. Refer to `include/linux/sm501mem.h` for the constants.

RAM is allocated using a *first-fit* algorithm, with the understanding that usage of this scarce resource will be governed by a single-purpose application.

Both the `SM501_ALLOC` and `SM501_FREE` work in terms of 4-byte offsets from the beginning of video RAM and are typically used to create pointers in combination with an `mmap`'ed frame buffer.

[1] one where the odometer values actually mean something

All allocated blocks are kept in a doubly-linked list per process and freed automatically when a owning process exits, so although the handles may be shared between processes, this may only occur during the lifetime of the process that allocates the memory.

5.2 Driver enhancement: Command list support

The SM-501 display controller contains a command list interpreter capable of executing a sequence of instructions. The command list interpreter itself only supports a handful of instructions to load and store memory or register values, along with a few control instructions.

When used in conjunction with the drawing engine, command lists can perform more elaborate sequences of `graphics` operations.

Either the frame-buffer device or the `/dev/sm501cmdlist` device may be used to execute a command list. Both of them operate on command list entries identified by offset in video RAM (allocated through the `SM501_ALLOC` ioctl()).

- The frame buffer driver supports synchronous execution of a single command through the `SM501_EXECCMDLIST` ioctl.
- `/dev/sm501cmdlist` supports execution of a number of command-list operations in a single `write()` system call, signalling completion through the `FASYNC` mechanism.

The command list queue for asynchronous operations is currently 128-entries deep, and is used to support both `/dev/sm501cmdlist` and frame-buffer execution.

5.3 Driver enhancement: Display refresh signals

In order to allow proper sequencing of odometer updates with display refresh, the `/dev/sm501vsync` device provides a mechanism to receive a real-time signal on every vertical sync.

5.4 Driver enhancement: YUV support

In order to reduce the processing needed to display `MPEG` video, the device `/dev/sm501yuv` is used to allow the creation of a YUV overlay with application-defined scaling factors.

A single `ioctl()` entry is required prior to display. The `SM501YUV_SETPLANE` call [2] is used to define the placement and geometry of the overlay. It should be passed a pointer to a structure of the following type:

```
struct sm501yuvPlane_t {
   unsigned xLeft_ ;
   unsigned yTop_ ;
   unsigned inWidth_ ;
   unsigned inHeight_ ;
```

[2] See `include/linux/sm501yuv.h` and `include/linux/sm501-int.h`.

```
   unsigned outWidth_ ;
   unsigned outHeight_ ;
};
```

After a successful call to `SM501YUV_SETPLANE`, subsequent `write()` calls are used to supply the next frame of data.

5.5 Driver enhancement: Alpha layer support

As with the YUV support, a Boundary Devices' driver allows access to the *Alpha* layer in the SM-501 through the `/dev/sm501alpha` device.

Refer to figure 1-9 in the SM-501 data book for a diagram of how these layers interact. The short description is that the *graphics* layer is at the bottom, overlaid by the *YUV* layer, and the *Alpha* layer is placed on top.

The following data structure is passed to the driver through the `SM501ALPHA_SETPLANE` ioctl.

```
#define SM501_ALPHA_RGB565    1
#define SM501_ALPHA_RGBA44    2
#define SM501_ALPHA_RGBA4444  3

struct sm501_alphaPlane_t {
   unsigned       xLeft_ ;
   unsigned       yTop_ ;
   unsigned       width_ ;
   unsigned       height_ ;
   unsigned       mode_ ;
   unsigned       planeOffset_ ;
   unsigned long  palette_[8];
};
```

Although the SM-501 hardware does support scaling of this layer, note that the driver does not. The driver does support a restricted area of the screen through the `xLeft_`, `yTop__`, `width_` and `height_` fields. This can be useful if only a small area of the display will be used for overlays.

Also note that in addition to the geometry information, the setup structure contains a `mode_` field[3] and a set of palette values. The `palette_` values are only read in mode `SM501_ALPHA_RGBA44` and are actually a set of sixteen RGB565 values.

RGBA44 or 4:4 mode contains 4 bits of opacity in the high order bits, and a 4-bit palette index in the low four bits. A value of `0xF0` is completely opaque, and a value of `0x00` is completely transparent.

RGBA4444 or 4:4:4:4 mode contains 4 bits of opacity in the high-order bits of a 16-bit value, followed by four bits each of red, green, and blue.

[3]Only `SM501_ALPHA_RGBA44` and `SM501_ALPHA_RGBA4444` are currently supported.

5.6 Driver enhancement: Direct register access

The frame buffer device (/dev/fb/0) supports two other ioctl() calls. The SM501_READREG and SM501_WRITEREG support reading and writing SM-501 registers directly.

The SM501_READREG call expects a pointer-to-longword parameter with the register number on input. It will return the value of the register if the register number is in the proper range.

The SM501_WRITEREG call expects a pointer to a structure like the following on input, and will write the specified value to the specified register.

```
struct reg_and_value {
   unsigned long reg_ ;
   unsigned long value_ ;
};
```

6 Layer two: wrapper classes and functions

You probably noticed in the preceding section that a lot of power is given to applications through the driver interfaces described.

There's also quite a bit of complexity left out of the drivers, and left up to applications.

In order to manage that complexity, a number of wrapper modules, classes and functions are provided in the bdScript directory.

The following subsections will walk through each, showing how the interfaces are cleaned up to make them easier to use.

6.1 Frame-buffer wrapper: fbDev.h/.cpp

The fbDevice_t class defined in fbDev.h/.cpp has been around a while, and serves a lot of purposes. It also carries some baggage acquired on the way.

Its' primary purpose is to provide convenient access to the frame-buffer RAM in an application. It uses the singleton design pattern to wrap a single instance through the getFB() routine. It exposes the geometry of the screen and the video RAM itself pretty directly:

```
   unsigned short getWidth( void ) const ;
   unsigned short getHeight( void ) const ;
   unsigned long getMemSize( void ) const ;
   void  *getMem( void ) const ;
```

It also provides more structured access to the RAM:

```
   unsigned short getPixel( unsigned x, unsigned y );
   void  setPixel( unsigned x, unsigned y, unsigned short rgb );
```

```
void clear( void ); // clear to black
void clear( unsigned char red, unsigned char green, unsigned char blue );
unsigned short *getRow( unsigned y );
```

and a number of primitive drawing routines for rectangles, lines, and text. Since these are not the emphasis of this document, I'll skip them here.

The frame-buffer device also exposes its' file descriptor for use in the various `ioctl()` calls described above:

```
int getFd() const ;
```

6.2 Alpha layer wrapper: sm501alpha.h/.cpp

In a similar manner as the frame buffer device, the `sm501alpha_t` defines a singleton class with a set of convenience methods for writing to the *Alpha Layer*.

Access to the singleton is through the `sm501alpha_t::get()` static method, which expects a parameter to define the mode.

Note that a number of the methods apply in either 4:4 or 4:4:4:4 mode.

"Either mode"
```
bool isOpen( void ) const ;
void clear( rectangle_t const &r );
mode_t getMode( void ) const ;
unsigned fbRamOffset(void) const ;
```

The 4:4 mode methods are pretty sparse. The `set()` method allows an image in packed 4:4 form to be copied to the display.

The `drawText()` method allows an antialiased text string to be converted into 4:4 mode and placed on the display.

A separate module (ftObjs.h/.cpp) is used to create the input data for this call through the FreeType library.

"4:4 mode methods"
```

// copy image bytes into a rectangle.
// source is packed
void set( rectangle_t const   &r,
          unsigned char const *ac44 );

// draw anti-aliased text into the alpha layer
void drawText( unsigned char const *alpha, // 8-bit alpha
               unsigned             srcWidth,
               unsigned             srcHeight,
               unsigned             destx,
               unsigned             desty,
               unsigned char        colorIdx );
```

The 4:4:4:4 mode method list is even sparser[4]. The `draw4444()` method will display an image onto the display at a given location. The image format must previously be created in 4:4:4:4 mode [5].

```
void draw4444( unsigned short const *rgba4444,
               unsigned             x,
               unsigned             y,
               unsigned             w,      // padded to 16-bytes
               unsigned             h );
```

[4] Is that even a word?
[5] See `fbImage_t` below

Note that while the `sm501alpha_t` class does have its' own file handle, it does *not* `mmap` its' own memory. The `fbDevice_t` class maps all of the frame buffer memory, so the `sm501alpha_t` class uses `fbDevice_t` to translate addresses. This isn't particularly fast, but it does reduce the number of page table entries used.

6.3 SM-501 memory wrapper: fbMem.h/.cpp

The fbMemory_t class declared in fbMem.h also piggybacks on the fbDevice_t class to produce pointers from offsets. It opens a file handle to the /dev/sm501mem device, and produces usable pointers from the offsets returned the SM501_ALLOC ioctl().

This is also pretty low-level, and isn't really used by the odometer application, except to implement the fbPtr_t class, which is much more friendly. fbPtr_t is a reference counted pointer class that wraps everything up into a nice little package.

It has three constructors:

```
class fbPtr_t {
public:
   fbPtr_t( void );
   fbPtr_t( fbPtr_t const &rhs );
   fbPtr_t( unsigned size );
...
```

An fbPtr_t constructed without an argument will not have a reference to any memory. Constructed with a size parameter, the class will attempt to allocate the requested amount of memory, and the copy constructor will share a pointer with the *right-hand-side* object.

fbPtr_t allows access to the memory through the getPtr() method.

6.4 SM-501 image wrapper: fbImage.h/.cpp

That's all very nice, you might say. But what you really want is to put *things* into display memory, not just access to the pointers. Things like *images*.

The fbImage_t class declared in fbImage.h does just that. It supports three methods of construction.

```
   fbImage_t( void );
   fbImage_t( image_t const &image,
              mode_t         mode );
   fbImage_t( unsigned x, unsigned y, unsigned w, unsigned h ); // from FB
```

The *mode* parameter indicates the color-space used to support either the alpha or graphics layers:

```
   enum mode_t {
      rgb565,
      rgba4444
   };
```

Constructed with no parameters, it's not very useful and doesn't reference any memory. Constructed with

an `image_t` parameter[6], the `fbImage_t` class will allocate frame-buffer memory for the image, and convert it as needed.

Conversion occurs in two forms:

1. **Color-space conversion** - If the `mode` parameter is RGBA4444, the RGB input will be cut down to 4 bits per color channel and 4 bits of opacity.
2. **Alignment conversion** - Most operations by the drawing engine require rows to be aligned on 128-bit boundaries. The fbImage will perform this conversion.

The third form of constructon (screen coordinates) will copy a rectangular region of memory from the frame-buffer graphics layer[7]. This form is most useful for saving and restoring state.

6.5 SM-501 command-list wrapper `fbCmdList.h/.cpp`

Enough about memory and pixels, already. The command-list is where the *action* is. Literally.

You may have been scratching your head when I referred to the command list earlier. The driver support is really simple, just "execute the command-list at this offset in RAM", with no support for actually creating the command lists.

This first class in user-space is equally opaque. The `fbCommandList_t` class has the following interface.

```
class fbCommandList_t
public:
   fbCommandList_t( void );
   ~fbCommandList_t( void );

   void push( fbCommand_t *cmd );
   unsigned size( void ) const { return size_ ; }
   void copy( void *where );
};
```

Okay, you may ask

> Now what?

The `fbCommandList_t` class is a container for `fbCommand_t` pointers. It is built to aid in the construction (and destruction) of command lists, and can copy them to frame-buffer memory.

It may not have occurred to you that command lists must be contiguous in frame-buffer RAM. Actually, that's not entirely true: there *is* support for a `jump` command, but code to perform fixups would be needed to implement a command list with jumps, requiring yet more code. To avoid this extra complexity, the `fbCommandList_t` class and the `fbCommand_t` base class use a two-stage model for construction.

[6] Discussion of the `image_t` class is outside the scope of this document. For the purpose of this document, it's enough to say that it contains an image in RGB565 form, with an optional 8-bpp alpha channel.

[7] RGB565 mode

- During the first stage, an object of class `fbCommandList_t` is created, and a set of `fbCommand_t` objects are pushed into (onto?) it. When this stage is complete, the total size of the command list is known.
- In the second stage, the set of `fbCommand_t` objects is walked, allowing each to be retargeted into the pre-allocated video RAM. The second stage is invoked through the `copy()` method on an `fbCommandList_t` object, which in turn calls the `retarget()` method on each contained `fbCommand_t` object.

The following subsections describe the current set of commands available.

6.6 SM-501 `jump` command: `fbCmdList.h/.cpp`

The `fbJump_t` class represents the `jump` instruction to the command-list interpreter. It simply advances the instruction pointer by the specified number of bytes.

6.7 SM-501 `blt` command: `fbCmdBlt.h/.cpp`

The `blt` command is supported through the `fbBlt_t` declared in `fbCmdBlt.h`.

It takes a bunch of parameters in its' constructor to describe the source and destination.

```
fbBlt_t( unsigned long    destRamOffs,
         unsigned         destx,
         unsigned         desty,
         unsigned         destw,
         unsigned         desth,
         fbImage_t const &srcImg,
         unsigned         srcx,
         unsigned         srcy,
         unsigned         w,
         unsigned         h );
```

Note that the source is an `fbImage_t` object to ensure both that it lives in frame-buffer memory and also that it has proper alignment.

A `set()` method is available with the same set of parameters as the constructor to allow the destination or source to be moved after an `fbBlt` command is placed into a command list.

The `fbBlt_t` class also supports conditional blts through the `skip()` and `perform()` methods. These methods replace the command portion of the instruction sequence with a `jump` when skipped, and restore it when not.

```
void skip( void );
void perform( void );
```

This feature is used in the odometer demonstration software to allow a second, inoperative `blt` in the instruction stream for the case when two `blts` are necessary (transition from nine to zero).

6.8 SM-501 wait command: fbCmdWait.h/.cpp

The `wait` command is implemented through the `fbWait_t` class declared in `fbCmdWait.h`.

It allows execution of the command-list interpreter to stall until a particular condition is met.

It takes two longword values in its' constructor. The first indicates which condition bits are to be considered. The second indicates the values of those bits that trigger continued execution.

```
fbWait_t( unsigned long bitsOfInterest,
          unsigned long values );
```

The `fbCmdWait.h` header file also declares the only bits used in the odometer application:

```
#define WAITFORDRAWINGENGINE 0x00000001
#define DRAWINGENGINEIDLE    0x00000000
#define DRAWINGENGINEBUSY    0x00000001
```

6.9 SM-501 finish command: fbCmdFinish.h/.cpp

A command-list must be terminated through the use of a `finish` command. This triggers the end-of-execution by the SM-501 and an interrupt to the driver.

The `fbCmdFinish_t` class can be used to construct this byte sequence.

7 Layer three: We start to see odometers

Almost all of the preceding discussion was general-purpose. It had almost nothing to do with odometer display, and amounted to a set of hardware/software tricks to get speed out of the SM-501 display engine on a Neon board. The following subsections will describe how those interfaces meet an actual application: the display of odometers.

7.1 How odometers are built

As shown in figure 1, odometers supported by this software package are constructed from a handful of images representing a currency symbol, a digit strip, a decimal point and a thousands separator.

The digit strip must be a multiple of ten pixels tall, with each digit evenly distributed within.

The comma and thousands separators are located in separate image files. They each must have a height equal to 1/10th of the digit strip's height.

To support pseudo-3D odometers, two sets of alterations may be applied to odometer digits and punctuation in the graphics layer.

The first shades the top and bottom of each digit to give an appearance of depth. The second provides a specular highlight to give the illusion of reflection.

The graph in figure 2 shows a plot of the shadow and highlight provided with the application. Two binary files contain this information, with one byte value for each pixel of height.

Note that shaded odometers are only supported by odometers in the graphics layer, because the shading is performed by semi-transparent data in the *alpha* layer. Because of this, shaded digits may not sit on top of a video layer.

In other words, alpha-layer odometers do not support shading.

The following sections will describe the library of modules used to put these parts together.

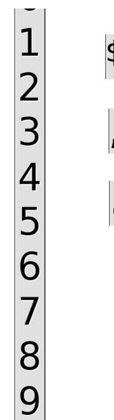

Figure 1: Graphical components

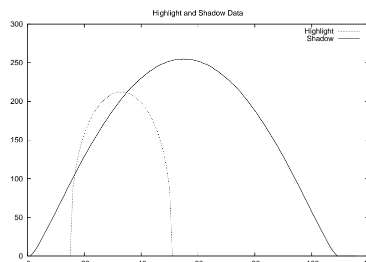

Figure 2: Highlighting data

7.2 The odometer mode `odomMode.h`

The odometer mode referred to above is encapsulated in the `odometerMode_e` enumeration in `odomMode.h`. It is used to drive the form and destination of odometer pixels.

```
enum odometerMode_e {
   graphicsLayer = sm501alpha_t::rgba44,
   alphaLayer    = sm501alpha_t::rgba4444
};
```

7.3 The graphics `odomGraphics.h/.cpp`

The class `odomGraphics_t` is used to load and hold the set of graphics images used to construct an odometer. The constructor takes a directory name and *mode* as input, and attempts to load the four image files, and highlight and shadow files if the mode is *graphicsLayer*.

```
struct odomGraphics_t {
   odomGraphics_t( char const    *dirname,
                   odometerMode_e mode);
   ...
```

The `odomGraphics` module also defines a singleton class that stores a set of odometer graphics objects by name. These are used in the odometer sample application to share graphics that use the *mediumGray* graphics style or *smallWhite* graphics.

```
class odomGraphicsByName_t {
public:
   static odomGraphicsByName_t &get(void); // get singleton

   void add( char const *name, odomGraphics_t *graphics );
   odomGraphics_t const *get(char const *name);
```

7.4 The odometers `odometer.h/.cpp`

Odometers themselves are defined by the `odometer_t` class, whose interface should probably not come as a surprise.

```
class odometer_t {
public:
   odometer_t( odomGraphics_t const &graphics,
               unsigned             initValue,
               unsigned             x,
               unsigned             y,
```

```
            unsigned            maxVelocity,
            odometerMode_e      mode );
   ~odometer_t( void );

   void setValue( unsigned newValue );
   void setTarget( unsigned newValue );

   void advance( unsigned numTicks );

   void hide( void );
   void show( void );

   odomValue_t  &value(void){ return value_ ; }
   unsigned      target(void){ return target_ ; }
   unsigned      velocity( void ){ return velocity_ ; }

   ...
```

The odometer is constructed using a specified set of graphics, a screen location, an initial value, and a mode[8].

The max velocity parameter, specified in pixels, places an upper limit on the amount of scrolling (in pixels) per screen refresh.

The initial value parameter is specified in units of currency (pennies for the U.S.).

Display of the odometer begins at zero and advances toward the *desired*, or *target* value accelerating to `maxVelocity` and decelerating as the target is approached. The `setTarget()` method sets the desired target value, while the `setValue()` method immediately displays the specified value.

The `advance()` method is called periodically during screen refresh to advance the current value and update the display. Actually, that's a bit of an overstatement. In reality, the `advance()` method and `setTarget()` routines both just update commands in a command-list for later execution by the `SM-501`'s command-list interpreter.

Each `odometer_t` defines a command-list chain with a number of embedded commands to display each digit and punctuation. More detail will follow in following sections on the `odomValue_t` and `odomDigit_t` classes.

The `odometer` module also defines a container for `odometer_t` named `odometerSet_t`. This class is currently a singleton, accessible through the static `odometerSet_t::get(void)` method. Odometers are added by name through the `add()` method and retrieved by name through the non-static `get(name)` method.

```
class odometerSet_t {
public:
   static odometerSet_t &get();

   inline bool isOpen( void ) const { return (0<=fdCmd_) && (0<=fdSync_); }
```

[8] alpha or graphics layer

```
void add( char const *name, odometer_t * );
odometer_t *get( char const *name );

void setValue( char const *name, unsigned );
void setTarget( char const *name, unsigned );

void stop( void );
void run( void );
bool isRunning( void ) const { return isRunning_ ; }
```

Convenience routines are present to set the value (`setValue()`) and target (`setTarget()`) of an odometer through the set, and a couple of routines are present to temporarily stop (`stop()`) and start (`run()`) the odometers. Note that the `run()` method does not actually perform any screen updates, but establishes signal handlers connected to some internal methods of the singleton.

```
void sigio(void);
void sigCmdList(void);
void sigVsync(void);
```

Finally, the vertical sync signal handler may be used as a trigger for other purposes through the `setHandler()` routine.

```
typedef void (*vsyncHandler_t)( void * );

void setHandler( vsyncHandler_t, void *opaque );
```

7.5 odometer value construction `odomValue.h/.cpp`

Most of the mechanics of implementing the odometer command list are actually performed in the `odomValue` module, and the `odomValue_t` class. Most of the interface to the class mirrors `odometer_t`, but the command-list is passed in, and the `odomValue_t` contains many of the implementation details, such as a video RAM copy of the background image and information about the digit locations and number of significant digits.

```
class odomValue_t {
public:
   odomValue_t( fbCommandList_t      &cmdList,
                odomGraphics_t const &graphics,
                unsigned              x,
                unsigned              y,
                odometerMode_e        mode );
   ~odomValue_t( void );

   enum {
```

```
    maxDigits_ = 8
};

void set( unsigned pennies );
void advance( unsigned numPixels );
unsigned value( void ) const { return pennies_ ; }

unsigned sigDigits( void ) const { return sigDigits_ ; }
rectangle_t const &getRect( void ) const { return r_ ; }
fbImage_t const &getBackground( void ) const { return background_ ; }
```

The `odomValue_t` contains code to create the command-list entries for display of the value punctuation, but passes the details for each digit on to the `odomDigit_t` class described below.

Note that the `advance()` method of `odomValue_t` works in pixel values, since `odometer_t` performs all velocity calculations.

7.6 odometer digits `odomDigit.h/.cpp`

The `odomDigit_t` class is used to create two command-list commands for a single digit, and update them as necessary. This class contains two `blt` commands for use when making a transition from nine to zero. When either a single blt or no blt is needed, the `odomDigit_t` method converts one or both of its' `blt` commands to `jump` commands, speeding execution.

```
class odomDigit_t {
public:
   odomDigit_t( fbCommandList_t  &cmdList,
                fbImage_t const  &digitStrip,
                unsigned          x,
                unsigned          y,
                odometerMode_e    mode );
   ~odomDigit_t( void );

   unsigned long advance( unsigned long howMany ); // pixels

   void     show( void );
   void     hide( void );

   inline bool isVisible( void ) const { return !hidden_ ; }
   inline bool isHidden( void ) const { return hidden_ ; }

   void set( unsigned digitVal );
```

8 Layer four: animation and video

The preceding sections covered the source code structure for odometers, and the mechanisms used to produce smoothly rotating digits.

This section will describe how animation and video are connected to the `odomTTY` program.

8.1 The playlist `odomPlaylist.h/.cpp`

A *playlist* container is defined in class `odomPlaylist_t` to control a set of operations over time. It can be thought of as a *first-in/first-out* queue of operations to be performed. Each operation may take a long time to execute (e.g. video playback), and the playlist will wait for completion of each operation before advancing to the next.

Each item in the playlist has a *repeat* flag to signal to the playlist that it should be re-inserted at the tail of the queue upon completion.

As listed in the `playlistType_t` enumeration, four types of playlist entries are currently supported:

```
typedef enum playlistType_t {
   PLAYLIST_NONE       = -1,
   PLAYLIST_STILLIMAGE = 0,
   PLAYLIST_MPEG       = 1,
   PLAYLIST_FLASH      = 2,
   PLAYLIST_CMD        = 3,
   PLAYLIST_STREAM     = 4
};
```

The *still image*, *MPEG*, and *Flash* playlist types are fairly straightforward. Each refers to something shown on the LCD panel.

The *command* entry is used to allow general-purpose commands to be executed in sequence with a playlist, and leaks into *Layer five*. Refer to the `odomCmdInterp_t` later in this document for details.

The *stream* data type allows a home-brewed form of MPEG streaming video to be displayed. It is unique in the list above because it will not complete until instructed from the outside. See the playlist `stop()` method below.

Playlist entries are added to a playlist through the use of the following structure:

```
typedef struct playlistEntry_t {
   playlistType_t    type_ ;
   bool              repeat_   ; // add to end of list on completion?
   unsigned          numTicks_ ; // duration for still images, ignored otherwise
   unsigned          x_ ;
   unsigned          y_ ;
   unsigned          w_ ;
   unsigned          h_ ;
```

```
   char              fileName_[512];
};
```

As you can see, the data provided includes the type, screen location and file name for the entry. A couple of these fields are overridden for particular playlist entry types. Notably:

- `numTicks_` is only used for still images, and
- `fileName_` contains a command string for commands, and the ASCII UDP port number for streams

The constructor for playlist class takes no parameters, and has entry points for adding entries, stopping the current entry (and advancing to the next), and purging the entire playlist. The `odomPlaylist_t` connects itself to the active odometer set through the static `odometerSet_t::get()` method.

```
class odomPlaylist_t {
public:
   odomPlaylist_t( void );
   ~odomPlaylist_t( void );

   enum {
      MAXENTRIES  = 16     // must be power of 2
   };

   void add( playlistEntry_t const &entry );
   void stop( void ); // stop current animation, move to next
   void purge( void ); // clear everything
   ...
```

In the current implementation, the playlist class is polled from the application's `main()` routine. This is done to allow lengthy operations such as `MPEG` decode to be interrupted by higher priority events such as vertical sync or command-list completion. The interface allows a vertical sync count to be passed as input, but the parameter isn't currently used. Each of the playlist types makes determinations about what to do next based on clock time instead.

```
   void play( unsigned long syncCount );
```

To provide easy glue to a command-line interpreter, the playlist class also contains a command dispatch routine to parse and execute 'playlist' commands. Again, refer to the `odomCmdInterp_t` below for a broader discussion.

```
   bool dispatch( odomCmdInterp_t   &interp,
                  char const * const params[],
                  unsigned           numParams,
                  char              *errorMsg,
                  unsigned           errorMsgLen );
```

Finally, the playlist contains a glue interface to the odometer's vertical sync handler that allows data from the current playlist entry (only used for MPEG) to be output to the screen on each vertical refresh:

```
void vsyncHandler( void );
```

The playlist types for *still images* and *commands* are implemented within the body of the `odomPlaylist` module. `MPEG` videos, whether from file or stream are separated to reduce the size of the module and allow separate reading of the code.

8.2 Playlist static video `odomVideo.h/.cpp`

The `odomVideo_t` class is used to implement the playback interface for static, or file-based MPEG videos.

```
class odomVideo_t {
public:
   odomVideo_t( odomPlaylist_t &playlist, // used to get YUV handle
                char const     *fileName,
                unsigned        outx = 0,
                unsigned        outy = 0,
                unsigned        outw = 0,
                unsigned        outh = 0 );
   ...
```

The constructor is pretty straightforward, simply reflecting the playlist entry structure, and passing a reference to the playlist itself for the sole purpose of sharing a file-handle to the `YUV` device. This is probably worth a comment or two.

> The playlist allows playback of multiple video files, and a common use is to play several videos of the same size in the same, reserved screen location.
>
> The YUV device requires a separate *set plane* call to establish the geometry of the output after each call to `open()`. In fact, it requires a `write()` operation after the *set plane* call before it will actually enable the video to prevent mis-display of any previous content with a new output geometry.
>
> To avoid blanking the YUV layer for a long period, and avoid mis-display of old data with a new geometry, ownership of the YUV layer is given to the playlist object.

Can you tell that an earlier version of this code actually did this?

The following methods are called from the playlist class's `play()` routine to decode the next frame(s) and detect end-of-file.

```
bool playback( void );
bool completed( void );
```

This method is called from the playlist's `vsyncHandler()` method to pump an output frame to the display at the right time.

```
   void doOutput( void );
```

8.3 Playlist streaming video odomStream.h/.cpp

The `odomStream_t` class bears a striking resemblance to the `odomVideo_t` class[9], although it derives from the class `mpegRxUDP_t`.

```
class odomVideoStream_t : public mpegRxUDP_t {
public:
   odomVideoStream_t( odomPlaylist_t &playlist, // used to get video fd
                      unsigned       port,
                      unsigned       outx = 0,
                      unsigned       outy = 0,
                      unsigned       outw = 0,
                      unsigned       outh = 0 );
   virtual ~odomVideoStream_t( void );

   // called to pump frame(s) to frame buffer from vsync
   void doOutput( void );
...
```

Notable differences are

- The `playback()` method is missing. Instead, the `poll()` method of the base class is called.
- There is no `completed()` method (since streams never stop).
- There are a set of event handler methods. I'll describe these below.

8.4 Streaming video base class mpegRxUDP.h/.cpp

In order to allow separate testing of the streaming video receiver, the mechanics of receiving an MPEG video stream have been separated from the playlist version into the `mpegRxUDP` module.

The interface to the module is fairly simple: it expects a UDP port number in its' constructor, a UDP socket and attempts to bind to the port.

```
class mpegRxUDP_t {
public:
   mpegRxUDP_t( unsigned short portNum );
   virtual ~mpegRxUDP_t( void );
```

[9] I probably even *cut-and-pasted* it

```
bool isBound( void ) const { return 0 <= sFd_ ; }
...
```

The class expects to be polled periodically to check for incoming UDP data, and keeps track of the state of the stream.

```
void poll( void );
```

The message format is defined in `mpegUDP.h`, and allows, but does not require each stream to contain multiple *files*, with separate geometry.

If multiple files are used, the following methods are called to indicate the start of one file and the end of a previous file.

```
virtual void onNewFile( char const *fileName,
                        unsigned    fileNameLen );
virtual void onEOF( char const   *fileName,
                    unsigned long totalBytes,
                    unsigned long videoBytes,
                    unsigned long audioBytes );
```

If an `mpegRxUDP_t` object is created mid-stream, the `onNewFile()` method will be called with an empty (“”) filename.

Once a file name has been provided, incoming data along with timing information will be passed to the object through the `onRx()` method:

```
virtual void onRx( bool                 isVideo,
                   bool                 discont,
                   unsigned char const *fData,
                   unsigned             length,
                   long long            pts,
                   long long            dts );
```

The `discont` parameter indicates that the `mpegRxUDP_t` class saw a discontinuity in the input stream. Each UDP packet is numbered, and anything other than *prevPacket+1* is considered a discontinuity.

Note that this class *does not perform any decoding.* It simply passes the encoded data to the `onRx` handler.

Also note that a companion *sender* application is available in `mpegSendUDP.cpp`.

9 Layer five: the odomTTY application

The odomTTY application consists largely of a command-interpreter, odomCmdInterp_t to control the set of classes described above and a single transport, odomTTY for reading commands.

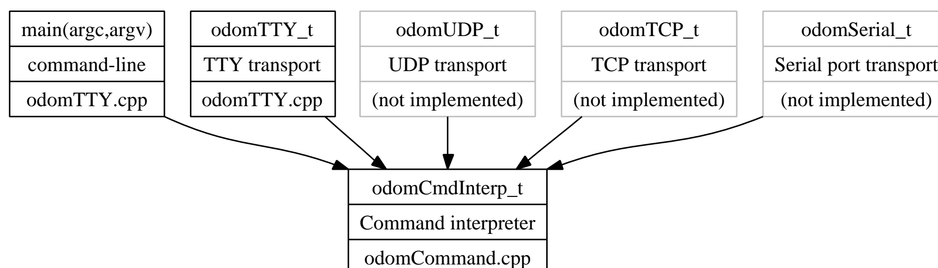

Figure 3: Application components

As shown in figure 3, the transport was created separately to allow commands to be dispatched from more than one transport.

Each of the modules and classes are pretty straightforward, so only minimal discussion is given to each.

9.1 The command list interpreter odomCommand.h/.cpp

The odomCmdInterp_t class is used to control the set of odometers and a playlist. A playlist is handed in to its' constructor, and transport classes issue commands through the dispatch() method.

```
class odomCmdInterp_t {
public:
   odomCmdInterp_t( odomPlaylist_t &playlist );
   ~odomCmdInterp_t( void );

   bool dispatch( char const *cmdline );
```

The dispatch() method returns false if an error occurs. A transport can determine the cause of the error (in human-readable form) through the getErrorMsg() method.

```
   char const *getErrorMsg( void ) const ;
```

Note that this interface prevents the odomCmdInterp_t class from being thread-safe. It is assumed that only one dispatch() call can be in progress at any one time.

The odomCmdInterp_t class also keeps track of whether application exit has been requested. This can be checked by an application through the exitRequested() method.

```
bool exitRequested( void ) const { return exit_ ; }
```

The following is a list of commands currently supported.

command	Usage and description
args	`args argument1 argument2...` Display command line arguments (to test command-line parser)
add	`add graphName odomName x y maxV initval target` Add and name an odometer.
caption	`caption fontName pointSize x y colorIdx text data...` Create an alpha-layer caption with the specified font, point size, color, and location.
clrAlpha	`clrAlpha x y w h` Clear the specified region of the alpha layer to transparent.
cls	`cls [rgb]` Clear the graphics layer to black or the specified RGB value if present.
font	`font fontName fontFile` Load and name a Freetype compatible font file.
graphics	`graphics name path` Load and name a set of graphics.
image	`image path x y` Load an image from *path* and display it at [x:y]
dump	`dump` Display information about running odometers to *stdout*.
mode	`mode 4444|565` Set the odometer mode.
playlist	`playlist subcommand params...` Issue a playlist command. Parameters are passed to `odomPlayList_t::dispatch()`
run	`run` Start the odometers.
screenShot	`screenShot path` Create a `PNG` image of the graphics layer and save it to *path*.
setTarget	`setTarget odomName targetValue` Set the target for the specified odometer to *value* in pennies.
setValue	`setValue odomName value` Set the current value of the specified odometer to *value* in pennies.
sm501reg	`sm501reg register [value]` Read specified `SM-501` register. If *value* is specified, set value.
source	`source path` Read and interpret a set of commands from the specified file.
stop	`stop` Stop the odometers.

9.2 The TTY transport `odomTTY.cpp`

The `odomTTY_t` class is used to read commands from *stdin* and issue them to a command interpreter.

```
class odomTTY_t : public ttyPollHandler_t {
public:
   odomTTY_t( pollHandlerSet_t  &set,
              odomCmdInterp_t   &interp );
   ~odomTTY_t( void );
```

Implementation makes use of a `ttyPollHandler_t` class, whose operation is outside the scope of this document.

The `odomTTY_t` class allows minimal command-line editing.

9.3 The `main()` routine `odomTTY.cpp`

The `main()` routine for the `odomTTY` application is located in `odomTTY.cpp` conditionally compiled if `MODULETEST` is defined.

It consists of only around 35 lines of code to instantiate a playlist, a command interpreter, an odometer console (`odomTTY_t`) and polls the tty and playlist until exit is requested.

```
int main( int argc, char const * const argv[] )
{
   odomPlaylist_t  playlist ;
   odomCmdInterp_t interp( playlist );
   for( int arg = 1 ; arg < argc ; arg++ ){
      char const *cmd = argv[arg];
      printf( "%s: ", cmd );
      if( interp.dispatch( cmd ) )
         printf( "success\n" );
      else
         printf( "error %s\n", interp.getErrorMsg() );
   }

   pollHandlerSet_t handlers ;
   odomTTY_t tty( handlers, interp );

   odometerSet_t &odometers = odometerSet_t::get();
   odometers.run();

   while( !interp.exitRequested() )
   {
      playlist.play( odometers.syncCount() );
      handlers.poll( -1 );
```

```
   }

   return 0 ;
}
```

10 To Do List

10.1 Pull vertical sync signal handler out of `odometer_t`

This would be better served by a general-purpose signal handler, one of whose call-outs is to the odometer signal handling code.

10.2 Check for hardcoded values

1. **max digits** - Currently 8. No more, no less (`odomValue.h`).
2. **optional pennies** - Should support dollar increments as well as pennies.

10.3 Enhance the performance of streaming

The current queuing isn't well-suited for streams and it shows at higher bit rates.

Fix bug in discontinuity test. Current implementation doesn't retain the 'discontinuity' flag until the next I-FRAME.

10.4 Multiple backgrounds

Currently, both `odomValue_t` and `odomDigit_t` store background images. `odomDigit_t` should be modified to use a piece of the larger background.

10.5 Flash performance

We are currently investigating whether or not the performance of the GPL flash player can be enhanced through either simple optimizations or the use of the SM-501 Graphics Processor.

11 Software License

The software written by Boundary Devices® is provided in source under the LGPL license, although a number of the libraries used to implement it are governed by the GPL.

Although we hope you find the software contained in this set of files is useful, no warranty is expressed or implied by Boundary Devices® .