Freescale Semiconductor
Application Note

Document Number: AN3997
Rev. 0, 03/2010

Building a Simple 3D Media Player for the i.MX31 in Linux

by *Multimedia Application Division*
Freescale Semiconductor, Inc.
Austin, TX

This application note describes the basic steps to build a simple media player for Linux® in the i.MX31 Freescale Microprocessor using the following application:

- Hardware (HW) overlay
- GStreamer
- tslib
- OpenGL® ES—PVRShell
- GTK threads
- Freescale Codecs

This application note provides a guideline to build a 3D media player on the i.MX31 using Linux. Additional programming such as error checking, thread handling, and memory handling are needed for the development of this application.

This application note also helps the developer to program a media player with a 3D interface. See Section 10, "References," for more information on how to develop a 3D media player with the API's presented in this document. For any issue during the development of a 3D media player, contact Freescale support at http://www.freescale.com/support.

Contents

freescale semiconductor

NOTE

The term **3D media player** in this application note refers to a media player with a 3D interface.

1 3D Media Player Overview

Two ways to implement media players that use a 3D interface are as follows:

- 3D frame rate dependent—Uses video as textures in a 3D surface
- Non-3D frame rate dependent—Plays video directly from a media API

1.1 3D Frame Rate Dependent Media Player

As the name implies, the media played is dependent on the frame rate of the 3D interface.

To implement a frame rate dependent media player, each frame on a video to be reproduced is converted to a texture and mapped to a 3D surface. The conversion can be done with OpenGL ES in combination with a media API.

Figure 1 shows a 3D frame rate dependent media player.

Figure 1. 3D Frame Rate Dependent Media Player

1.2 Non-3D Frame Rate Dependent Media Player

As the name implies, the frame rate from the media being played and the 3D interface are not dependent on each other.

This application note explains how this kind of application can be built in the i.MX31 and how it is implemented with API's and packages which are provided in Freescale Board Support Packages (BSP).

1.3 Advantages, Disadvantages, and When to Use

This section describes the advantages and disadvantages of 3D frame rate dependent media player and non-3D frame rate dependent media player. This section also includes important remarks on when to implement each of these media players.

Advantages

The advantages of 3D frame rate dependent media player and non-3D frame rate dependent media player are as follows:

- 3D frame rate dependent
  - — Better looks

    Since video is attached to a 3D surface, it has all the properties of an object in 3D space and can be moved in 3D fashion—rotate the video, translate the video, scale the video or a combination of those to achieve effects (for example, a flag effect).
- Non-3D frame rate dependent media player
  - — Better frame rates in video

    Since video is reproduced by a media API the video frame rate is not dependent on the frame rate of the 3D interface.
  - — Easy to implement

    Less coding and less experienced programmers in both graphics and streaming media API's are required.

Disadvantages

The disadvantages of 3D frame rate dependent media player and non-3D frame rate dependent media player are as follows:

- 3D frame rate dependent
  - — May be slow

    If the 3D interface has large amount of geometry or complex animation, the frame rate decreases. If the programmer is not careful with 3D graphics, the video may have undesirable frame rates since the 3D graphics and the video being played share frame rates.
- Non-3D frame rate dependent media player
  - — Not as good looking

    Since the video is not attached to a 3D surface it cannot be moved as a 3D object.

2 HW Overlays

The Image Processing Unit (IPU) performs HW overlay which allows the i.MX31 to run two applications that take over the frame buffer, such as OpenGL and GStreamer at the same time. This section gives an overview of how the IPU makes this function possible. See the *MCIMX31 and MCIMX31L Applications Processors Reference Manual* (MCIMX31RM) for more information on the IPU and its modules.

2.1 Synchronous Display Controller

The synchronous display controller (SDC) module of the IPU features HW overlays. This section describes the data flow of a 3D media player through the SDC.

The SDC module is composed of the Synchronous Display Buffer Memory, the Combining Unit, the Cursor Generator and the Synchronous Display Adapter. Figure 2 shows the data flow through the SDC in a 3D media player.

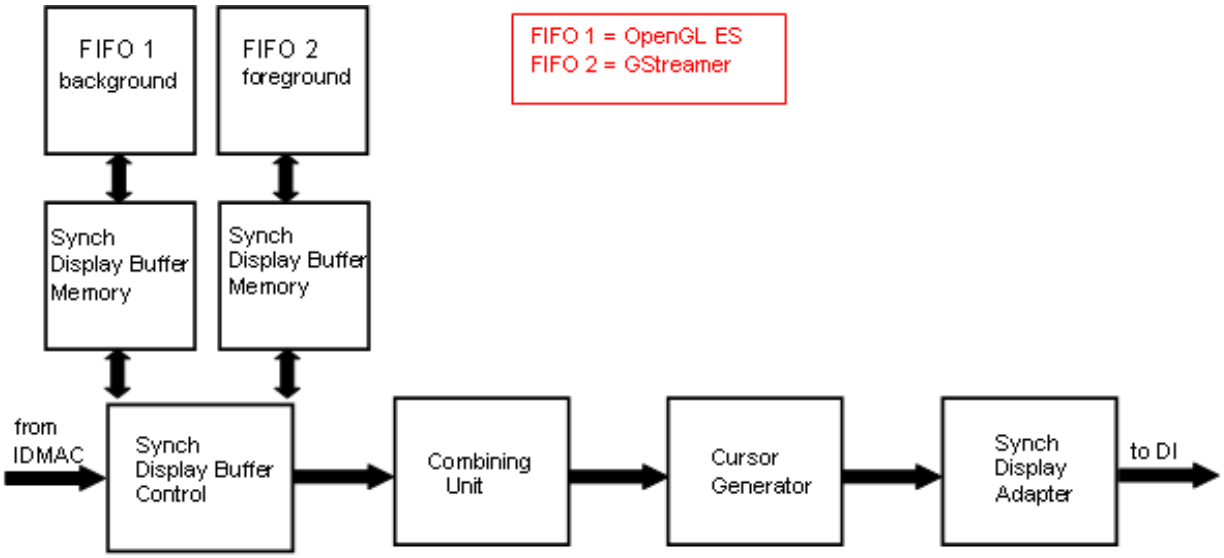

Figure 2. Data Flow in a 3D Media Player

The flow of a 3D media player using GStreamer and OpenGL ES through the SDC controller is as follows:

1. OpenGL ES and GStreamer sends pixel data to be displayed to Synchronous Display Buffer through two FIFOs corresponding to background and foreground planes.
2. The Synchronous Display Buffer Control assigns the positions for foreground and background planes.
3. The Combining Unit combines foreground and background planes and does blending operations such as local alpha, global alpha or key color. Displaying 3D graphics over a video is possible using the Combining Unit functions.
4. The Cursor Generator creates the cursor and set its size and position in the screen.
5. The Synchronous Display Adapter sends the pixels to the display and proper Synchronous signals.

Figure 3 shows how a frame is composed by the SDC.

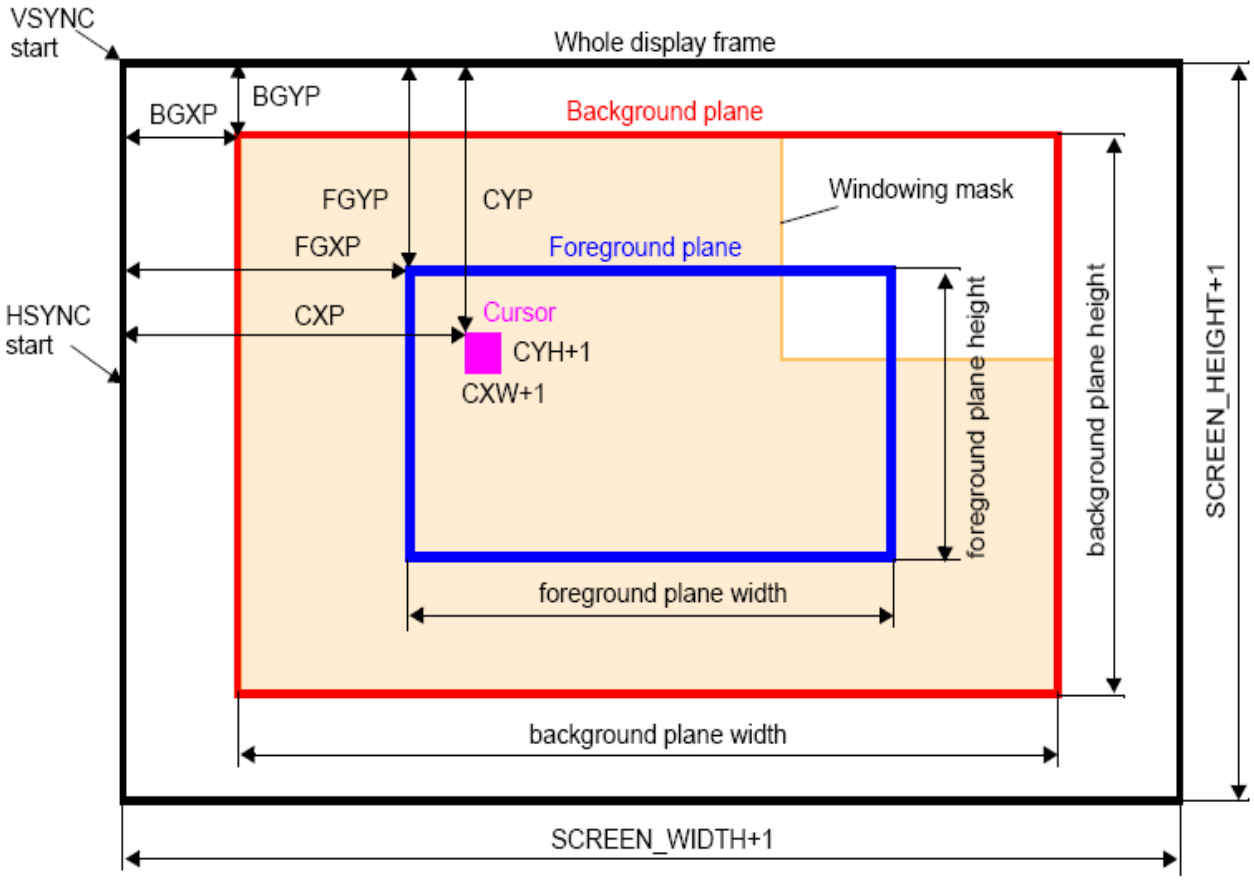

Figure 3. Display Frame Composed by the SDC

The description of the values referred in Figure 3 areas are as follows:

HSYNC	Horizontal Synchronization signal (HSYNC), also known as FPLINE or LP indicates the LCD that a line has ended and the valid pixels to follow are part of the next line.
VSYNC	Vertical Synchronization signal (VSYNC), also known as FPFRAME, FLM, SPS or TV, when active, indicates the LCD that the current frame has ended. Both of these signals are controlled by the Synchronous display adapter in the last part of the SDC.
BGXP/BGYP	Background plane X position and Background plane Y position indicates the position of the background plane.
FGXP/FGYP	Foreground plane X position and Foreground plane Y position indicates the position of the foreground plane.
CXP/CYP	Cursor X position and Cursor Y position indicates the position of the cursor which is controlled by the Cursor Generator.

3 Linux Target Image Builder (LTIB) Configuration

This section provides a step by step guide to install and configure LTIB with the proper packages to build a 3D media player.

To develop a 3D media player, the system needs the following packages:

- For media streaming: GStreamer
- For 3D graphics: OpenGL ES
- For touch screen events: tslib
- For thread handling: Gtk

The procedure to install and configure LTIB with the proper packages to build a 3D media player is as follows:

1. Install the LTIB package.

```
$tar zxvf <ltib package>
$cd <ltib installation directory>
$./install
```

NOTE

This procedure should not be done as root.

2. Get Freescale Codecs and Freescale Plug-ins packages in the system.

   `*codecs*$VERSION.tar.gz` is the GStreamer plugin source package which contains source code for the multimedia GStreamer-based plugin for the i.MX application processor.

   `*plugins*$VERSION.tar.gz` is the codec/parser binary package which contains the multimedia core codec/parser libraries for the i.MX31 application processor.

3. Extract both packages and copy the extracted `.tar.gz` files to the LPP directory, which by default is set to `/opt/freescale/pkgs/`.
4. Replace the `.spec` files in the `<ltib directory>/dist/lfs-5.1/fsl-mm/` with the extracted `.spec` files.
5. Configure LTIB.

```
$cd <ltib directory>
$./ltib -m config
```

   The menu shown in Figure 4 is launched.

6. Select Platform of choice and then select Freescale i.MX reference boards then exit. When asked to save the new configuration, select yes.

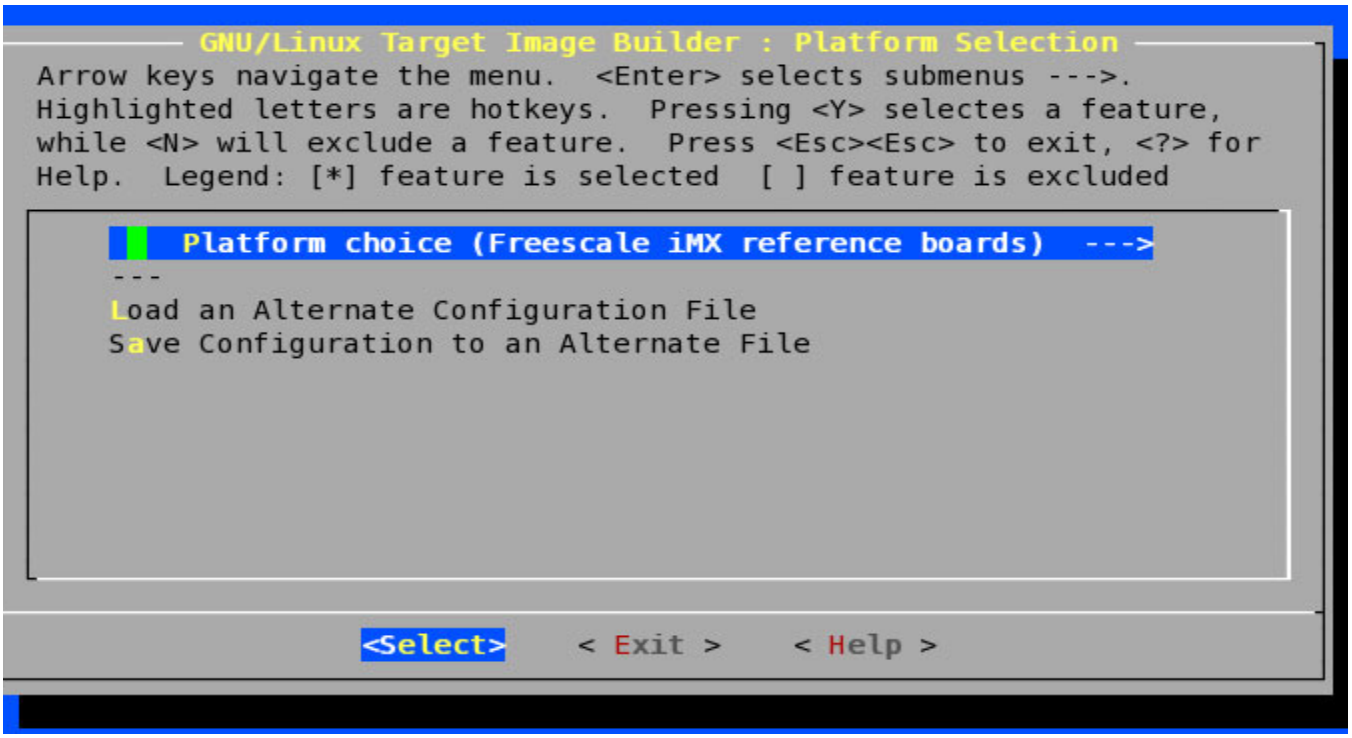

Figure 4. Platform Selection

7. In the next menu, select **Package list** as shown in Figure 5.

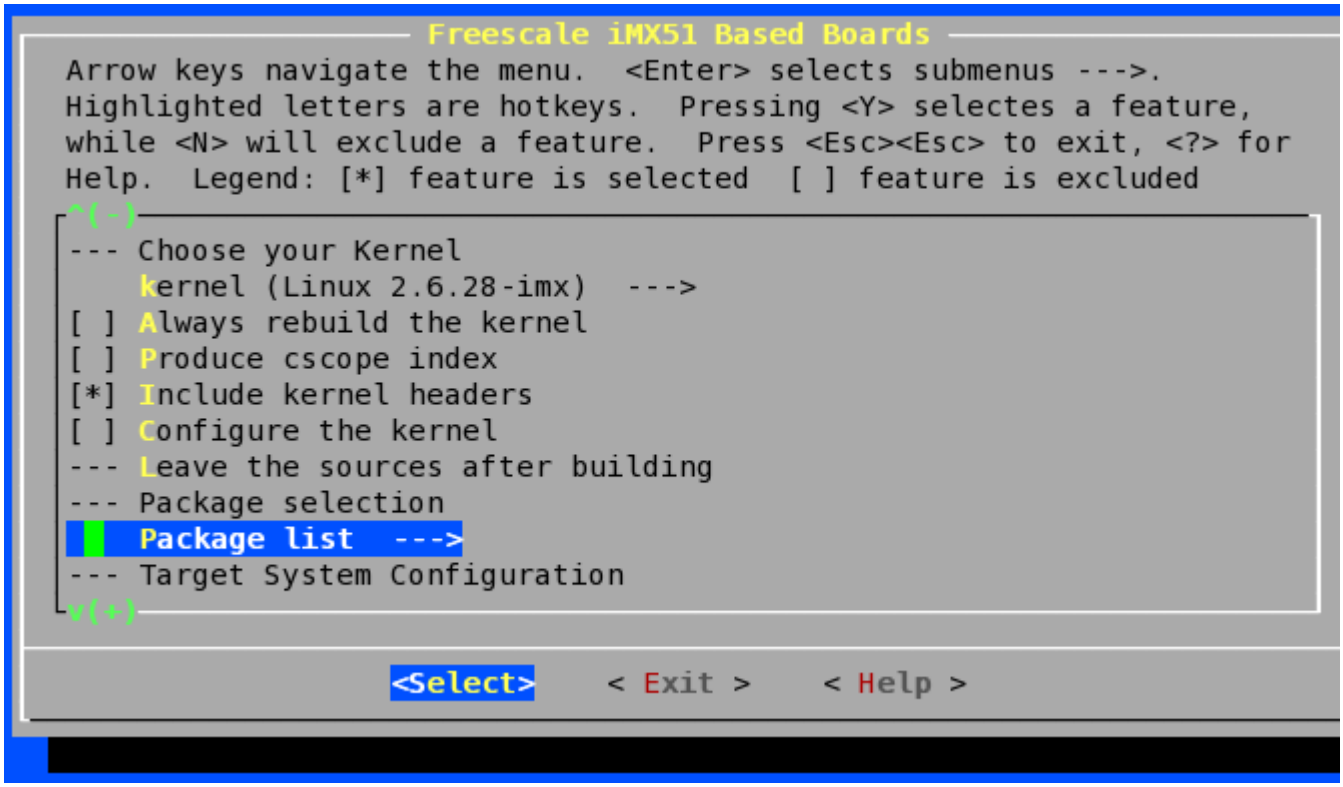

Figure 5. Package List Selection

8. Select the following packages from the package list:
   - GPU driver **mbx-bin**
   - GStreamer good plugins package **gstreamer-good-plugins**
   - GTK **gtk+**
   - Tslib **tslib**

Figure 6 shows the GPU driver **mbx-bin** package.

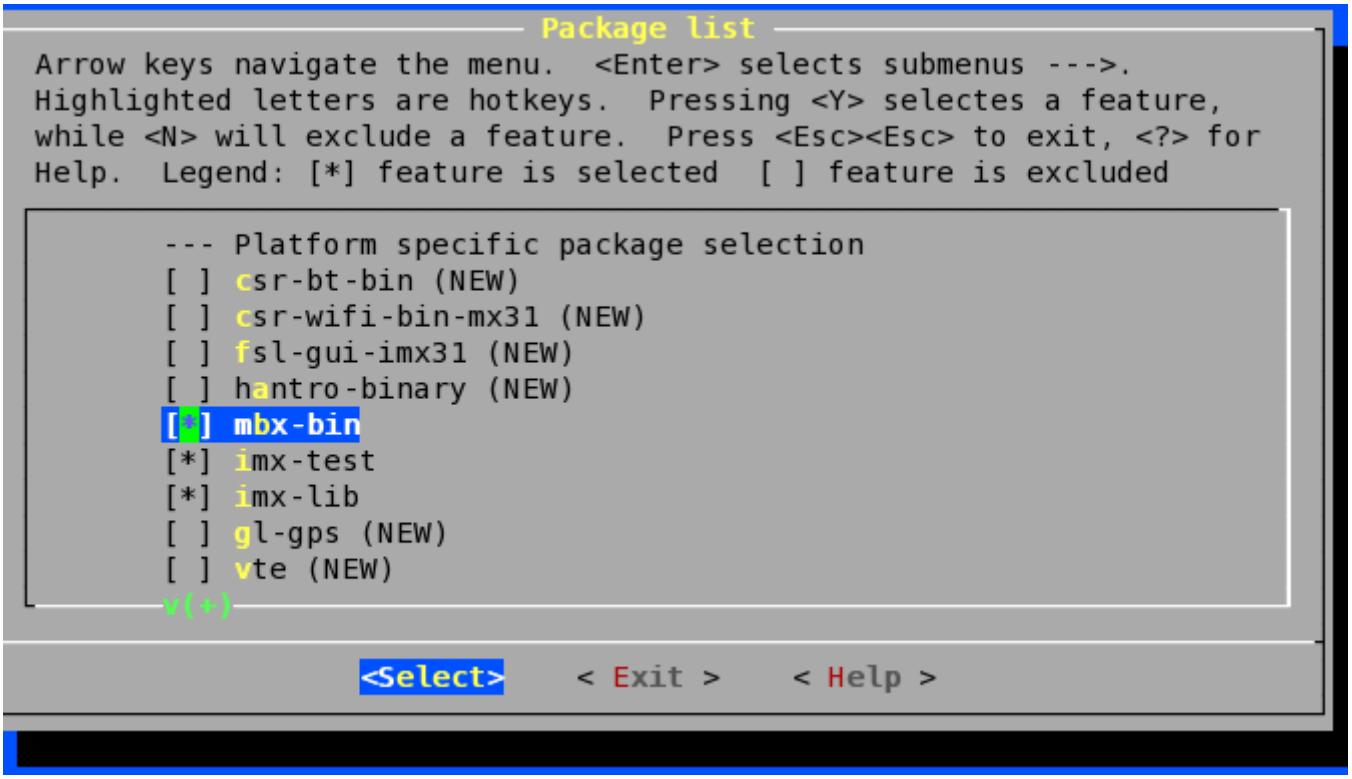

Figure 6. GPU Driver mbx-bin Package

Figure 7 shows the GStreamer good plugins package.

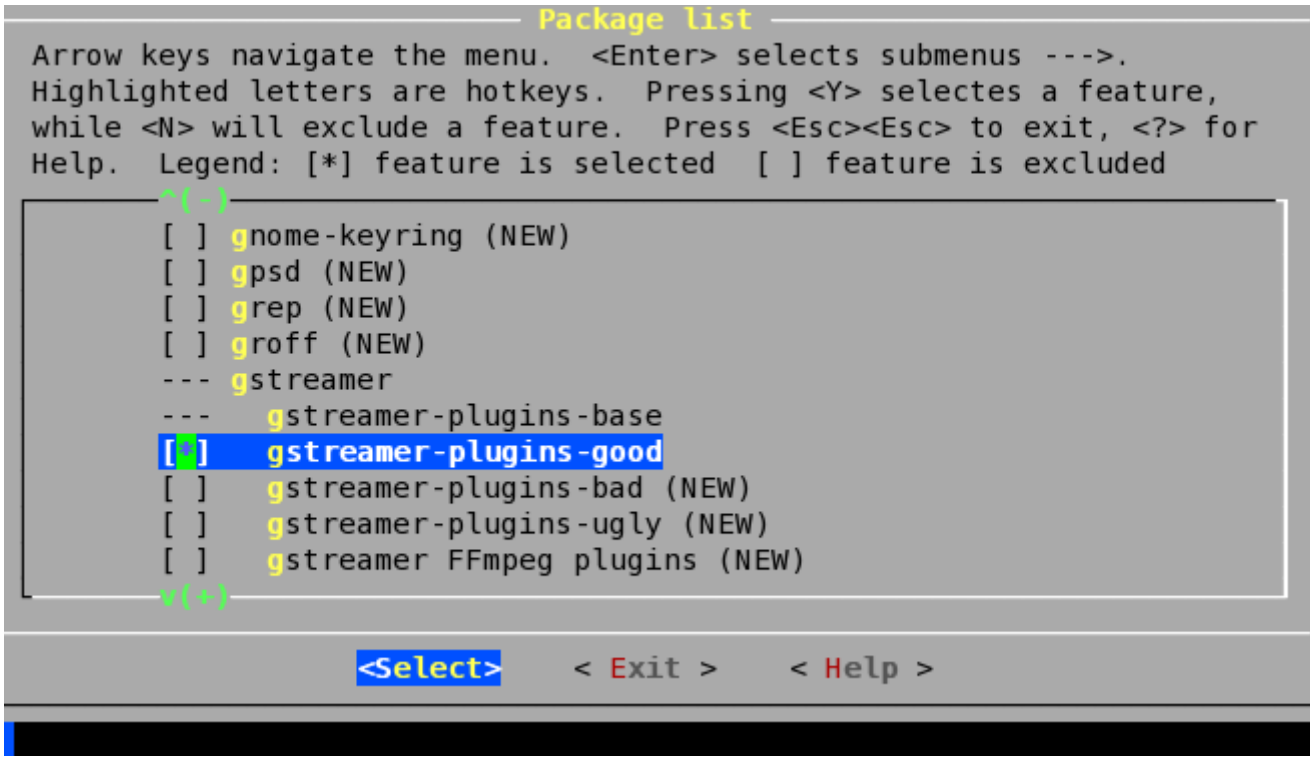

Figure 7. GStreamer Good Plugins Package Showing gstreamer-good-plugins

Figure 8 shows the GTK package.

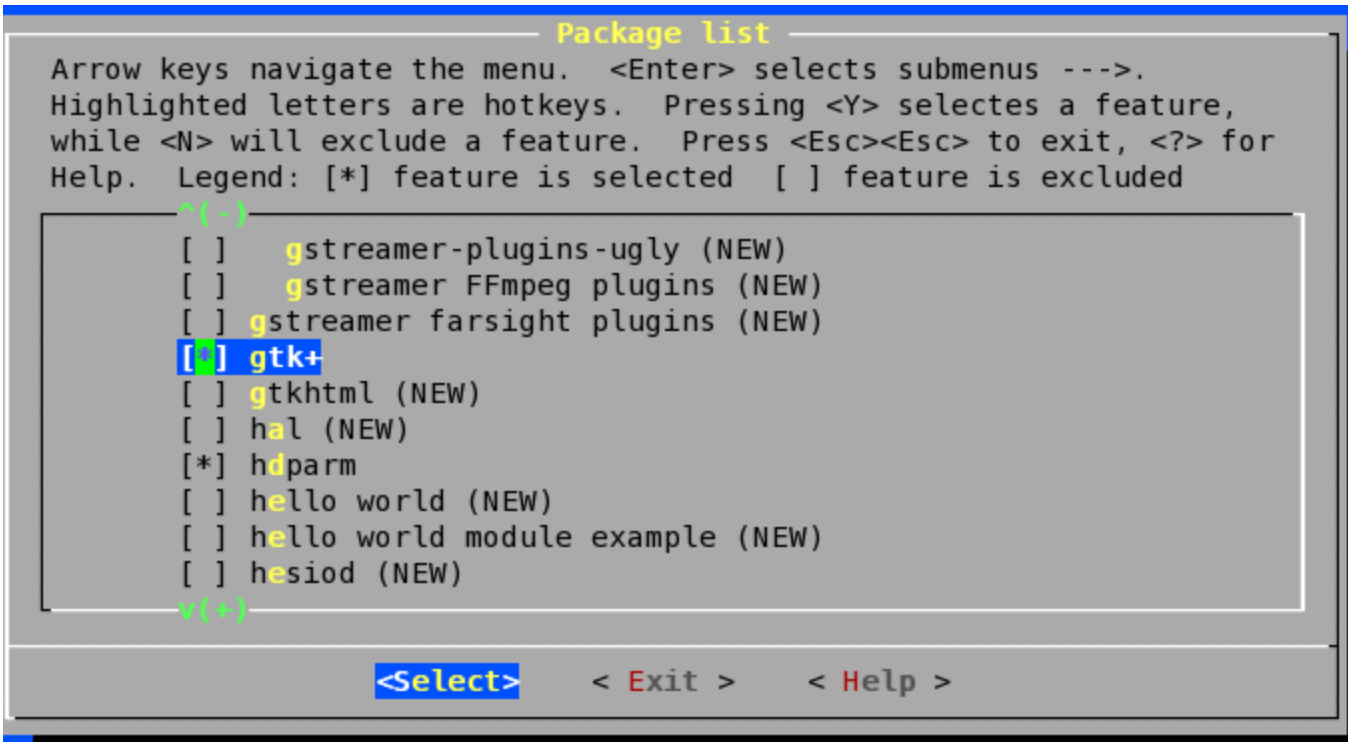

Figure 8. GTK Showing gtk+

Figure 9 shows the Tslib package.

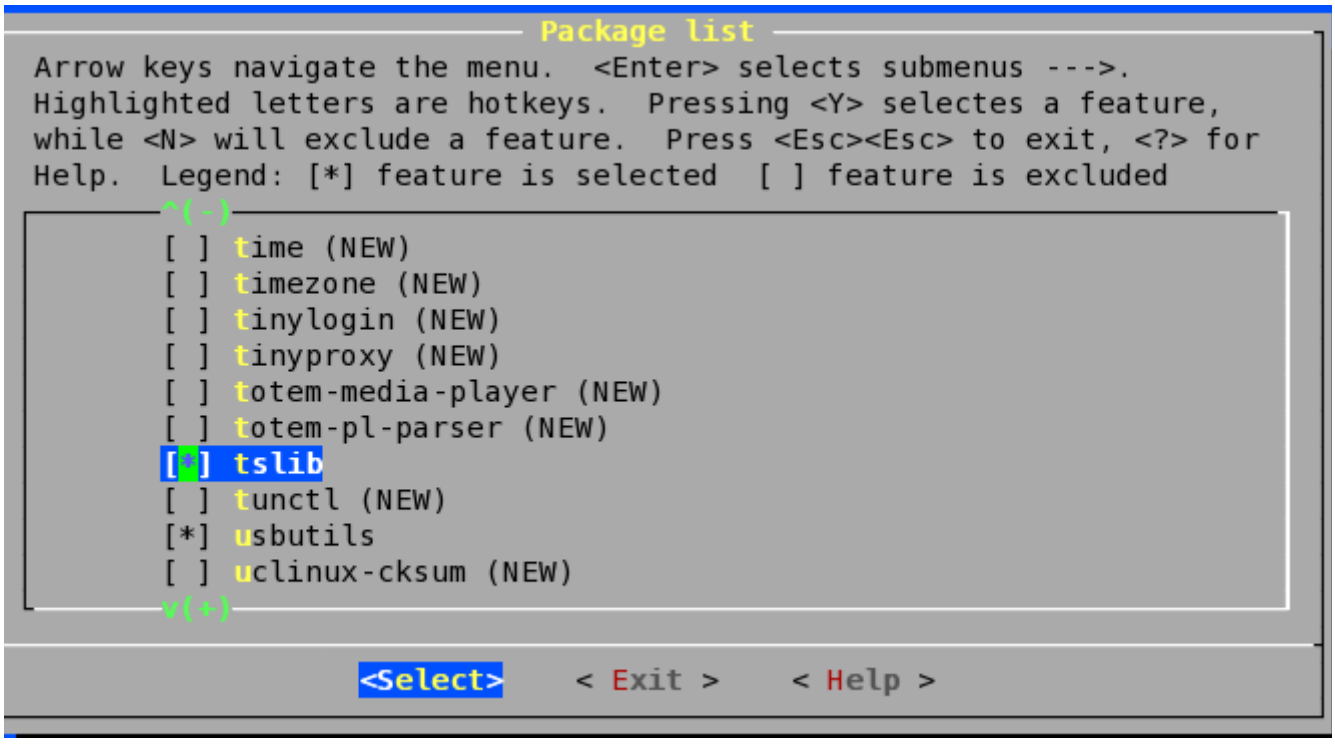

Figure 9. Tslib Showing tslib

9. Finally, select **Freescale Multimedia Plugins/Codecs** then **gstreamer-fsl-plugins** and exit ltib as shown in Figure 10 and Figure 11.

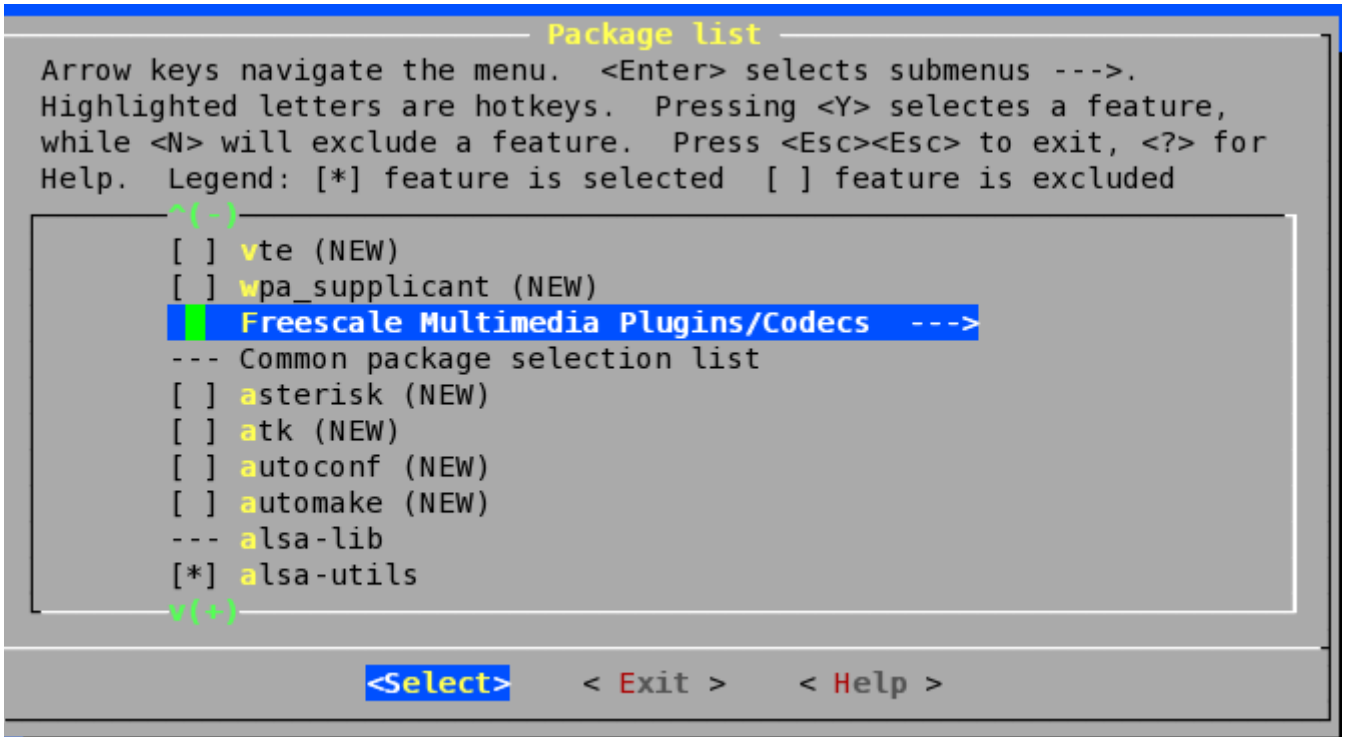

Figure 10. Freescale Multimedia Plugins/Codecs Selection

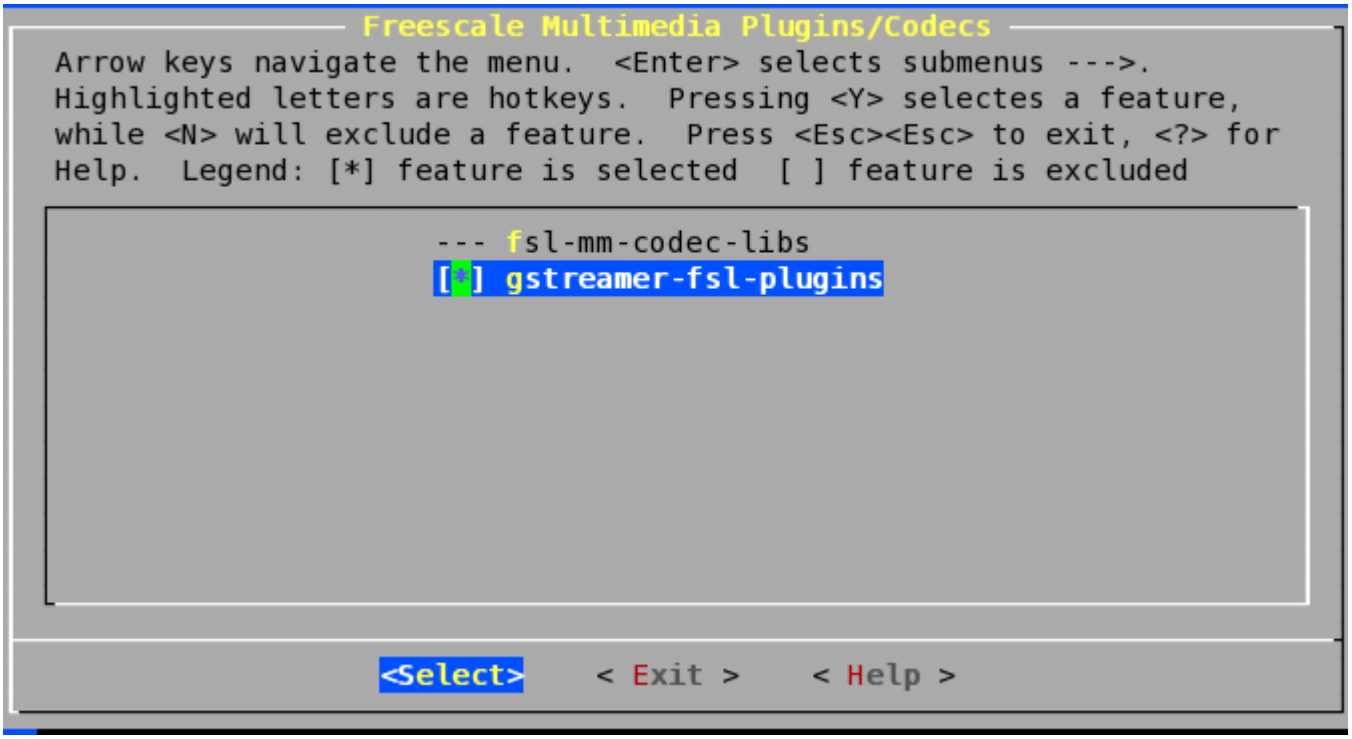

Figure 11. gstreamer-fsl-plugins Selection

NOTE

See *i.MX31 PDK 1.5 Linux User's Guide* to add configuration changes needed to build any kind of system such as NFS, JFFS, and so on.

10. Compile LTIB.

```
$cd <ltib directory>
$./ltib
```

NOTE

Some packages may not compile due to lack of packages on the host. To overcome this, install the missing package if the package already exist, install the development package counter part of the missing package then delete the directory under `<ltib directory>/rpm/BUILD/<missing package>` and compile LTIB again.

4 GStreamer

GStreamer is a framework for creating streaming media applications based on gobject. The i.MX31 supports GStreamer and Freescale codecs are GStreamer compliant. This section focuses on building a simple media application using Freescale codec's. For more information on building GStreamer application, see *GStreamer Application Development Manual*.

4.1 GStreamer Basics

The following are the basic parts of GStreamer:

Elements	Black box that can be chained to several other black boxes in order to create a data flow. For example, an element may be a file source, a demux, a decoder or a display among many others.
Bins and Pipelines	Bin is a collection of elements and a pipeline is a type of bin that has the ability to allow execution of all its elements.
Pads	Allows communication between GStreamer elements. For example, a demux and decoder should be joined by pads.
Bus	Takes messages from a pipeline. It is very useful to keep an eye on the execution of a pipeline. A bus need not be created since every pipeline has one bus. It only needs a message handler (a place to put the messages).

4.2 Initializing GStreamer

To initialize GStreamer framework, perform the following steps:

1. Include the GStreamer header file `gst/gst.h`
2. Call the `gst_init` function to perform the necessary initialization for GStreamer:

```
gst_init (NULL, NULL);
```

4.3 Using Freescale Codecs

In this section a command from the *Freescale codecs Users Guide* is emulated in the code.

```
gst-launch filesrc location=Kaleidoscope_mp4v_mp3_320x240_30fps_284kbps_a_44.1khz_160kbps.avi
! mfw_avidemuxer ! mfw_mpeg4decoder! mfw_v4lsink
```

```
void play()
{
        /*Declare pipeline and elemens*/
        GstElement *pipeline;
        GstElement *source, *demux, *decoder, *videosink, *sink;
        /*Initialize gstreamer*/
        gst_init (NULL, NULL);
        /*first we create the gstreamer main loop were the pipeline will run*/
        loop = g_main_loop_new (NULL, FALSE);
        /*create the pipeline and name it player*/
        pipeline = gst_pipeline_new ("player");
        {
                /*create the bus*/
                GstBus *bus;
                /* create the same elements as the one used in the command*/
                /*Create a filesource*/
                source  = gst_element_factory_make ("filesrc", "source");
                /*Create a mfw_avidemuxer*/
                demux   = gst_element_factory_make ("mfw_avidemuxer", "demux");
                /*Create a mfw_mpeg4decoder*/
                decoder = gst_element_factory_make ("mfw_mpeg4decoder",
                                "decoder");
                /*Create a mfw_v4lsink*/
                videosink = gst_element_factory_make ("mfw_v4lsink",
                                        "videosink");
                /*Verify all the elements were created succesfuly*/
                if (!pipeline || !source || !demux || !decoder || !videosink)
                        g_warning ("Failed to create elements!");
                /*set location property of filesource to an avi video*/
                g_object_set (G_OBJECT (source), "location",
        "Kaleidoscope_mp4v_mp3_320x240_30fps_284kbps_a_44.1khz_160kbps.avi",
                        NULL);
                /*set the created bus to the one of our pipeline*/
                bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
                /set buscall as message handler*/
                gst_bus_add_watch (bus, bus_call, NULL);
                /*Building the pipeline
        To build the pipeline you need to add the elements and link them*/
                /* must add elements to pipeline before linking them */
        gst_bin_add_many (GST_BIN (pipeline), source, demux, decoder, videosink, sink, NULL);
                /* link source and demux*/
                if (!gst_element_link_pads (source, "src", demux, "sink")) {
                        g_warning ("Failed to link elements source and demux!");
                }
                /*link decoder and videosink*/
                if (!gst_element_link_many (decoder, videosink, NULL)) {
                        g_warning ("Failed to link elements!");
                }
                /*link demux and decoder
                Since demux has an output for audio and an output for video some special
```

```
                handling needs to be done by using g_signal_conect function an our callback
                function pad added explained below*/
                if (!g_signal_connect (demux, "pad-added", (GCallback)pad_added,
                decoder))
                {
                        g_warning ("Failed to link elements demux and decoder!");
                }

}
//function to link demux and decoder
static void pad_added (GstElement* demux, GstPad* pPad, GstElement* decoder)
{
        /*declare a pad*/
        GstPad *sinkpad;
        /*get the sink pad of the decoder*/
        sinkpad = gst_element_get_static_pad (decoder, "sink");
        /*link the sink pad of the decoder with the one of the demux*/
        gst_pad_link (pPad, sinkpad);
        /*get rid of the sinkpad object*/
        gst_object_unref (sinkpad);
}
```

4.4 GStreamer Pipeline States

In the previous code, the pipeline was created. The pipeline can be set to the following states:

GST_STATE_NULL — This state deallocates all resources held by the pipe. The pipeline is set to this state when the program terminates.

GST_STATE_READY — This state resets the stream position to the beginning and allocates all resources. The pipeline is set to this state when the stop button is pressed.

GST_STATE_PAUSED — The stream stops but it retains its position and resources. The pipeline is set to this state when the pause button is pressed.

GST_STATE_PLAYING — The stream plays and the data flows through all the elements in the pipeline. The pipeline is set to this state when the play button is pressed.

4.5 GStreamer Events

Event handling for a simple GStreamer player is shown in the code below:

```
if (ExitEvent)
{
                //deallocate resources in the pipeline
                gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_NULL);
                //deallocate the pipeline itself
                gst_object_unref (GST_OBJECT (pipeline));
                //deallocate gstreamer main loop
                gst_object_unref (GST_OBJECT (loop));
                return;
}
else if (PlayEvent)
{
                //set pipeline state to PLAYING
                gst_element_set_state (GST_ELEMENT (pipeline),
                                                GST_STATE_PLAYING);
```

```
}
else if(PauseEvent)
{
                //set pipeline state to PAUSE
        gst_element_set_state (GST_ELEMENT (pipeline),
                                        GST_STATE_PAUSE);
}
else if(StopEvent)
{
                //set pipeline state to READY
                gst_element_set_state (GST_ELEMENT (pipeline),
                                                GST_STATE_READY);
}
```

5 tslib

tslib is a package that provides the functionality to capture touch screen events in the display. This section provides a basic guideline to obtain x and y coordinates from touch screen events using tslib.

5.1 tslib Basics

The following are the basic parts of tslib:

struct tsdev: Structure that needs to be initialized to work with tslib.

struct ts_sample: Structure that contains x and y coordinates for a touch screen event. It also contains pressure which is a variable that is different from 0, while the screen is still pressed and 0 if the screen is not pressed.

5.2 Initializing tslib

The following are the steps to initialize tslib:

1. Include the tslib header file `tslib.h`.
2. Initialize a `tsdev struct` with the proper tsdevice by calling the `ts_open` function
3. Finish initialization by running `ts_config`.

```
struct tsdev *ts;
ts =tsopen("/dev/input/event1",0); //touch screen device is /dev/input/event1 in the
i.mx31
ts_config(ts); //Configure ts
```

5.3 Getting x and y Values from the Screen

To get the x and y values, use the `ts_sample` structure. The following sample code shows how this is done. The code is taken from one of the tests contained in the tslib package. The code takes x and y sample and stores it in an array of `ts_sample` structures while the screen is pressed, then, it quickly sorts the results and then stores to set x and y variables.

```
void getxy(struct tsdev *ts, int *x, int *y)
{
```

```
//defining the Maximun number of samples to be took
#define MAX_SAMPLES 128
        /* array of sample structures to store x and y values took */
        struct ts_sample samp[MAX_SAMPLES];
        /* int index to store the number of samples took, middle to quick sort
        the results */
        int index, middle;
        /* Get ts ready to read */
        do {
                if (ts_read_raw(ts, &samp[0], 1) < 0) {
                        printf("ts_read");
                        exit(1);
                }

        } while (samp[0].pressure == 0);


        /* Now collect up to MAX_SAMPLES touches into the samp array. */
        index = 0;
        do {
/* if  max samples numbers of samples hasn't been took increment index */
                if (index < MAX_SAMPLES-1)
                        index++;
                    /* Read form ts and store it in a sample structure */
                    if (ts_read_raw(ts, &samp[index], 1) < 0) {
                        printf("ts_read");
                        exit(1);
                }
        /* while there is still pressure in the screen */
        } while (samp[index].pressure > 0);
        /* Print the number of samples took */
        printf("Took %d samples...\n",index);


     /* quick sort of x and y */
        middle = index/2;
        if (x) {
                qsort(samp, index, sizeof(struct ts_sample), sort_by_x);
                if (index & 1)
                        *x = samp[middle].x;
                else
                        *x = (samp[middle-1].x + samp[middle].x) / 2;
        }
        if (y) {
                qsort(samp, index, sizeof(struct ts_sample), sort_by_y);
                if (index & 1)
                        *y = samp[middle].y;
                else
                        *y = (samp[middle-1].y + samp[middle].y) / 2;
        }
}
```

5.4 tslib Events

The following code continuously waits for tslib events and reads x and y values using the function described in the previous section.

```
//declare tsdev structure to use tslib
struct tsdev *ts;
while(1)
{
         //open touch screen device and point it to ts strcuture
         ts = ts_open("/dev/input/event1", 0);
         //make sure ts_open worked
         if (!ts)
         {
                  printf("ts_open failed");
                  return;
         }
          //run ts_config to finish initialization of ts
          if (ts_config(ts))
          {
                  printf("ts_config");
                  return;
          }
         //call our previously created function
           getxy (ts, &x, &y);
}
```

5.5 Setting tslib Events

Obtaining x and y coordinates from the touch screen events is acceptable for a simple 3D media player that does not have 3D objects that are behind each other. If recognition of 3D objects in perspective is required, ray tracing should be implemented. In this case, only x and y coordinates are obtained.

Figure 12 shows the layout of media player in the sample code.

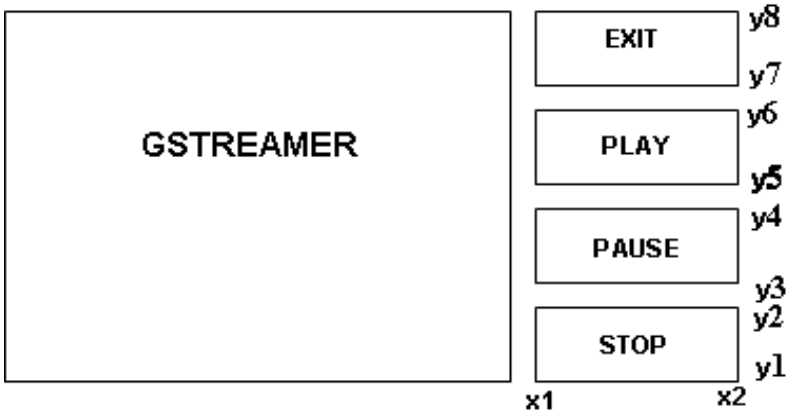

Figure 12. Layout of a Media Player

```
//Initial position of buttons in x, final position of buttons in x
if( (x>x1) && (x<x2) )
{
         //Initial position in y and final position in y for stop button
         if ( (y>y1) && (y<y2) )
         {
                  ExitEvent = False;
                  PlayEvent = False;
                  PauseEvent= False;
```

```
                StopEvent = True;
        }
        //Initial position in y and final position in y for pause button
        if ( (y>y3) && (y<y4) )
        {
                PlayEvent = False;
                ExitEvent = False;
                PauseEvent = True;
                StopEvent = False;
        }
        //Initial position in y and final position in y for play button
        if ( (y>y5) && (y<y6) )
        {
                PauseEvent= False;
                PlayEvent = True;
                ExitEvent = False;
                StopEvent = False;
        }
        //Initial position in y and final position in y for exit button
        if ( (y>y7) && (y<y8) )
        {
                StopEvent = False;
                PauseEvent= False;
                PlayEvent = False;
                ExitEvent = True;
        }
}
```

6 OpenGL ES—PVRShell

OpenGL ES is a royalty-free, cross-platform API for 3D graphics on embedded systems. OpenGL ES provides an efficient way of interfacing software and hardware acceleration through a graphics processing unit (GPU). The i.MX31 GPU-MBX supports OpenGL ES 1.1. For more information on OpenGL ES, see the following link:

http://www.khronos.org/opengles/sdk/docs/man/

Imagination Technologies provides an OpenGL ES based SDK for 3D graphics. This SDK contains demos, trainings, and a shell that features functions such as loading 3D objects as .POD files.

For more information on how to get, install, and use the PVR SDK, contact Freescale support at: http://www.freescale.com/support.

6.1 PVRShell Introduction

PVRShell offers several advantages such as encapsulation of EGL and platform abstraction so that application can be easily developed, ported or reused. PVRShell also features built-in command line options such as specifying the number of frames before quitting the application or the backbuffer size. PVRShell is object oriented and easy to use.

To use the PVRShell, the application must be defined as a class that inherits from the PVRShell, and must implement the following five methods:

- InitApplication—Used to initialize application data on the 3D graphics data.
- InitView—Used to initialize 3D data.
- RenderScene—This Render loop is called repeatedly to draw geometry.
- ReleaseView—Function called to release any 3D data memory.
- QuitApplication—Function called to release the memory of non 3D data.

See lesson 3 of *PVR SDK 03_Introducing PVRShell* for examples on how to use the PVRShell.

6.2 POD and PVR Files

A 3D object is composed of an array of triangles and an array of normals to the triangle surface (for light refraction calculations) and texture coordinates. Triangles are composed of three vertices. Each vertex is composed of a position in 3D space (X, Y, Z component).

Building complex 3D object in programming code is an impossible task. Programs such as lightwave, 3D Max or Maya are used to build complex 3D objects. Figure 13 shows a sample 3D object.

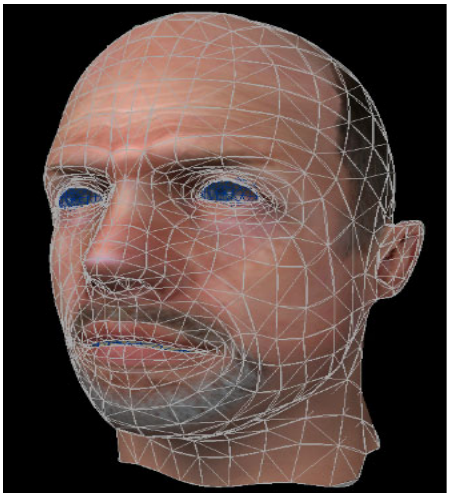

Figure 13. 3D Object

PVR SDK also contains a plug-in for Autodesk 3D Studio Max that converts 3D objects to .POD format. This format can be easily loaded by the PVR Shell. The plug-in performs optimization of geometry reducing the number of triangles in the object and orders the triangles so, it is correctly drawn in OpenGL.

In 3D graphics, PVRShell can perform texture handling in an easy and efficient way. The PVRTexTool tool provided in the PVRSDK can convert images to .PVR extension.

To easily load the texture of the created 3D objects, name textures the same way as in graphics design tool (3D Studio Max, Maya) and in the program.

The following code illustrates how the POD objects for exit, play, pause, and stop should be handled inside PVRShell. All the OpenGL ES setup for rendering was omitted. For the details on implementation of a full 3D program using POD files, see lesson 07 of the *PVRSDK trainings 07_IntroducingPOD*.

```
//declare our CPVRTPODScene objects as members of the class
CPVRTPODSceneexit,play,stop,pause;
//declare texture handles for each of the textures of our objects
```

```
GLuint            tex_exit, tex_play, tex_stop, tex_pause;


bool 3DPlayerInterface::InitApplication()
{
        //Load POD files
        exit.ReadFromFile("/data/exit.POD");
        play.ReadFromFile("/data/play.POD");
        stop.ReadFromFile("/data/stop.POD");
        pause.ReadFromFile("/data/pause.POD");
}
bool 3DPlayerInterface::QuitApplication()
{
        //Destroy CPVRTPODScene objecs
        exit.Destroy();
        play.Destroy();
        stop.Destroy();
        pause.Destroy();
}
bool 3DPlayerInterface::InitView()
{
        //load textures form .PVR files
if(!PVRTLoadTextureFromPVR("/data/exit.pvr", &tex_exit))
{
                PVRShellOutputDebug("**ERROR** Failed to load texture \n");
        }
if(!PVRTLoadTextureFromPVR("/data/play.pvr", &tex_play))
{
                PVRShellOutputDebug("**ERROR** Failed to load texture \n");
        }
if(!PVRTLoadTextureFromPVR("/data/stop.pvr", &tex_stop))
{
                PVRShellOutputDebug("**ERROR** Failed to load texture \n");
        }
if(!PVRTLoadTextureFromPVR("/data/pause.pvr", &tex_pause))
{
                PVRShellOutputDebug("**ERROR** Failed to load texture \n");
        }
}
bool 3DPlayerInterface::ReleaseView()
{
        // Free the textures
        PVRTReleaseTexture(tex_exit);
        PVRTReleaseTexture(tex_stop);
        PVRTReleaseTexture(tex_pause);
        PVRTReleaseTexture(tex_play);
        return true;
}
bool 3DPlayerInterface::RenderScene()
{
/*The drawing code of your buttons should be here is blank in this code because it depends
heavily on the animations of the buttons*/
}
void 3DPlayerInterface::DrawScene(CPVRTPODScene* scene)
{
        glPushMatrix();
                //Draw each mesh in the 3D object
                for (int i=0; i<(int)scene->nNumMeshNode; i++)
```

```
                {
                        SPODNode* pNode = &scene->pNode[i];


                        // Gets pMesh referenced by the pNode
                        SPODMesh* pMesh = &scene->pMesh[pNode->nIdx];


                        // Loads the correct texture using our texture lookup table
                        if (pNode->nIdxMaterial == -1)
                        {
                                // It has no pMaterial defined. Use blank texture (0)
                                glBindTexture(GL_TEXTURE_2D, 0);
                        }
                        else
                        {
                                glBindTexture(GL_TEXTURE_2D, m_uiTex_head);
                        }

                        //Get vertex data
                        glVertexPointer(3, GL_FLOAT,
scene->pMesh[i].sVertex.nStride,
                                                        scene->pMesh[i].sVertex.pData);
                        //Get normal data
glNormalPointer(GL_FLOAT, scene->pMesh[i].sNormals.nStride,
scene->pMesh[i].pInterleaved +
(size_t)scene->pMesh[i].sNormals.pData);
                //Get UV text cords
                        glTexCoordPointer(2, GL_FLOAT,
scene->pMesh[i].psUVW[0].nStride,
scene->pMesh[i].pInterleaved +
(size_t)scene->pMesh[i].psUVW[0].pData);
                        //Draw elements
                        if(!scene->pMesh[i].nNumStrips)
                        {
                                if(scene->pMesh[i].sFaces.pData)
                                {
                                        glDrawElements(GL_TRIANGLES,
scene->pMesh[i].nNumFaces*3, GL_UNSIGNED_SHORT,
scene->pMesh[i].sFaces.pData);
                                }
                        }
                }

        glPopMatrix();
}
//Requiered by PVR Shell to launch the 3D app
PVRShell* NewDemo()
{
        return new Morphing();
```

6.3 Animation Basics

There are three main animation techniques used in 3D graphics:

- Morphing—Act of changing an object into another. Morphing is used as an animation technique widely to animate cloth, skin, faces and non solid bodies.

- Hierarchical Animation—Animation techniques based in moving through an array of pivots in a hierarchical fashion and applying the basic operations of OpenGL ES.
- Skeletal Animation—Also called skinning because it emulates the way the skin and muscles react to their attached bones. This technique is used in humanoids, birds and most vertebrates.

See application note, *3D Animation Techniques on the i.MX31 PDK* (AN4044) for a detailed description on how to implement these techniques.

6.4 Events for 3D Player Interface

The following code shows the events for 3D player interface:

```
if (ExitEvent)
{
                  //Animation for exit button
                  exitAnimation();
                  //Release 3D related memory
                  ReleaseView();
                  //Release the rest of memory
                  QuitApplication();
                  return;
}
else if (PlayEvent)
{
                  //Animation for play button
                  playAnimation();
}
else if(PauseEvent)
{
                  //Animation for pause button
         pauseAnimation();
}
else if(StopEvent)
{
                  //Animation for stop button
                  stopAnimation();
}
```

7 GIMP Drawing Kit (GDK) Threads

GDK threads are a component of the GNOME application development environment. GDK threads are supported in the i.MX31 and can be run in parallel which is the desired effect in a 3D media player.

7.1 Initializing GDK Threads

The procedure to initialize the GDK threads is as follows:

1. Include the GStreamer header file `gtk/gtk.h`.
2. Call the function `gdk_threads_init` to initialize GDK threads internals.

7.2 Creating GDK Threads

The following are the functions to create GDK threads:

- g_thread_create function—Creates a thread with default priority. The arguments of g_thread_create are as follows:
  - GThreadFunc—Function to be spawn in the thread
  - gpointer data—Argument for the function
  - gboolean joinable—Shows if the thread is joinable or not
  - GError **error—Return location for error

7.3 Sample Media Player Threads

For a 3D media player application, GStreamer for streaming media should run in one thread. The OpenGL ES using the PVRShell and tslib for touch screen events should run in two separate threads. The only variables shared between the three threads are the boolean values corresponding to the touch screen events for play, stop, pause, and exit.

But the Boolean values for events only need to be read by GStreamer and the OpenGL ES threads. The only thread writing Boolean values for events is the tslib thread therefore, there is no need to synchronize those variables.

The following code shows how the threads are implemented. This code should go on PVRShellOS.cpp due to the nature of the PVRShell. In order to encapsulate 3DPlayerInterface in a thread this code needs to be added on the main of the Linux implementation of the PVR shell.

```
int main(int argc, char **argv)
{
//declare 3 Threads for GStreamer, OpenGL ES and tslib
GThread          *ThreadGS, *ThreadOGL, *ThreadTS;
         //declare error handlers for the threads
         GError           *err1 = NULL ;
         GError           *err2 = NULL ;
         GError           *err3 = NULL ;
         //check if GDK threads are supported
         if( g_thread_supported() )
                  // Called to initialize GDK threads internals
                  gdk_threads_init();
         else
                  printf("g_thread NOT supported\n");
         //Create each thread
if( (ThreadGS = g_thread_create((GThreadFunc)play, (void *)NULL, TRUE,
&err1)) == NULL)
         {
                  printf("Thread create failed: %s!!\n", err1->message );
                  g_error_free ( err1 ) ;
         }


         if( (ThreadOGL = g_thread_create((GThreadFunc)tsevent, (void *)NULL,
            TRUE, &err2)) == NULL)
         {
                  printf("Thread create failed: %s!!\n", err2->message );
```

```
                g_error_free ( err2 ) ;
        }


        if( (ThreadTS = g_thread_create((GThreadFunc)render, (void *)NULL,
                TRUE, &err3)) == NULL)
        {
                printf("Thread create failed: %s!!\n", err3->message );
                g_error_free ( err3 ) ;
        }
//wait for each thread to finish it's complition
        g_thread_join(ThreadGS);
        g_thread_join(ThreadOGL);
        g_thread_join(ThreadTS);
        //exit
        return 0;
}


//Code to contain 3D in a Gthread
void render(){
        PVRShellInit init;
/* Create the demo, create the OS initialiser.*/
        PVRShell *pDemo = NewDemo();
        //make sure the demo was created successfully
        if(!pDemo)
                printf("error code : %i \n", EXIT_ERR_CODE);
        init.Init(*pDemo);      //run the demo
        while(init.Run());
        //release memory
        delete pDemo;
        return;
}
```

8 Bringing Everything Together

OpenGL ES Thread and tslib thread should share the events variables. This can be done in various ways:

- Modify PVR Shell and set the code in PVRShellOS.cpp as suggested above.
- Include the `ts_lib` code in the events section of PVRShell and do the proper handling.

Regular memcopies and memreads of the events variables also add a mutex for synchronization.

9 Conclusion

This application note states the basics of building a simple non 3D frame dependent 3D media player using the advantages of the i.MX31 hardware and BSP. See Section 10, “References,” for more information on 3D and streaming media develop. For any question or remark on this application note contact Freescale Support at: http://www.freescale.com/support.

10 References

The following references give more information on 3D media player:

- *MCIMX31 and MCIMX31L Applications Processors Reference Manual* (MCIMX31RM)
- *i.MX Linux Multimedia Framework User's Guide*
- *GStreamer Application Development Manual (0.10.14)*
- OpenGL Programming Guide—The Red Book
- i.MX31_MBX_OGLES_1_1_SDK_for_Linux tutorials
- *3D Animation Techniques in the i.MX31 PDK* Application Note
- http://library.gnome.org/devel/glib/stable/glib-Threads.html

11 Revision History

Table 1 shows a revision history for this application note.

Table 1. Document Revision History

Rev. Number	Date	Substantive Change(s)
0	03/2010	Initial release

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3997
Rev. 0
03/2010