**WHITE PAPER**

***ESC Boston 2008***
***Class: ESC-321***

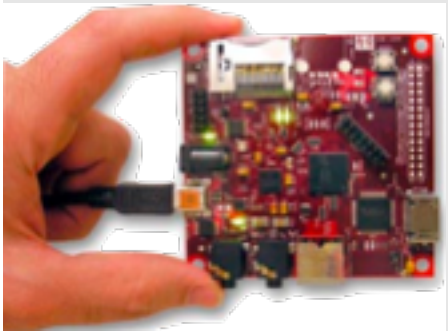
By **Todd Fischer**
*R&D Manager*
RidgeRun

# Embedded Streaming Media with GStreamer

## Executive Summary

Beagleboard has many features supporting streaming audio and video thanks to the OMAP™ 3 architecture. Developing a multimedia application that utilizes all of the hardware accelerators that are available is a daunting task unless a streaming media framework is available. GStreamer is such a framework, available for OMAP3, that simultaneously simplifies application development and utilizes the hardware accelerators.

This document provides an overview of GStreamer, how using GStreamer simplifies multimedia application development, and compares GStreamer to other multimedia frameworks like Texas Instrument's DMAI and OpenMax.



## Introduction

BeagleBoard features powerful streaming audio and video capabilities thanks to the use of the Texas Instruments OMAP™ 3530 application processor with an Cortex A-8 ARM processor supporting the NEON instruction set and an integrated C64 DSP with a video hardware accelerator. GStreamer makes multimedia easy on the BeagleBoard.

## GStreamer

GStreamer uses the notion of sources, filters, and sinks connected in a pipeline to handle streaming audio and video data. The GStreamer framework has been around 9 years and has wide adoption on GNOME desktop, with a rich set of media players, recorders, audio and video editors, among other uses. There are over 200 GStreamer plug-ins available. Much of this GStreamer infrastructure can be used directly in embedded multimedia devices. Multimedia frameworks options for embedded devices is expanding with the emerging support for OpenMAX and development of hardware specific solutions like Texas Instruments DMAI. This paper presents examples on using GStreamer to cement the key concepts, discusses potential challenges when GStreamer is used in embedded devices, and compares GStreamer to other multimedia frameworks for embedded devices.

# Pipeline Creation

GStreamer elements consist of sources, filters, and sinks. A group of elements is also an element called a bin. The top-level bin is called the pipeline. The pipeline can be controlled by setting the state, to play or pause, for example. The other bit of GStreamer terminology that is used frequently are pads. Elements have source pads and sink pads. The pipeline is connection of source pads to sink pads. Figure 1 Simple mp3 player shows the filesrc source element which reads data from a file named music.mp3, a mad filter element that converts MP3 encoded data to binary, and an alsasink sink element that passes the data to ALSA audio output[1].

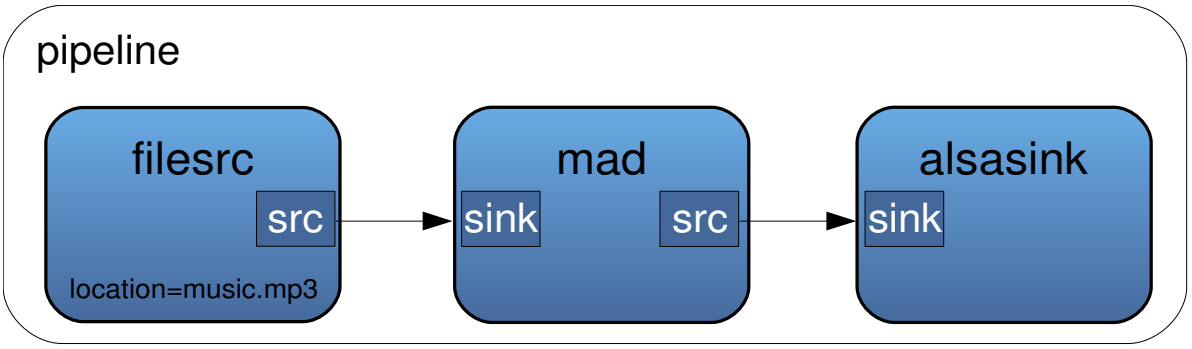



*Figure 1: Simple mp3 player*

The following C source code example is a modified version from the GStreamer Application Development Manual hello world example. In addition to the main() function, a bus_call() function is used to handle end of stream, as shown below:

```
static gboolean bus_call (GstBus *bus, GstMessage *msg, gpointer data)
{
  GMainLoop *loop = (GMainLoop *) data;

  switch (GST_MESSAGE_TYPE (msg)) {

    case GST_MESSAGE_EOS:
      g_print ("End of stream\n");
      g_main_loop_quit (loop);
      break;

    case GST_MESSAGE_ERROR:
      g_print ("Error\n");
      g_main_loop_quit (loop);
      break;

    default:
      break;
  }

  return TRUE;
}
```

1 The pipeline can be created using the `gst-launch` program which can build and run a GStreamer pipeline from the command line.

gst-launch filesrc location=music.mp3 ! mad ! alsasink

*Code Sample 1: MP3 player pipeline bus monitor*

The example main() has been simplified and made to align with the vala example given later. The key aspects of the example are the various elements and how they are connected in a controllable pipeline. Each of the four elements are created, the file location is set for the filesrc file ready source, the source, filter, and sink are connected added to the pipeline. The src and sink pads of the elements are connected together in the pipeline. The bus_call() end-of-stream handler connected to watch for bus events, the pipeline state is set to play, and then we turn it loose with a call to g_main_loop_run() to activate the pipeline.

```
#include <gst/gst.h>
#include <glib.h>

int main (int   argc, char *argv[])
{
  GMainLoop *loop;
  GstBus *bus;
  GstElement *source;
  GstElement *filter;
  GstElement *sink;
  GstElement *pipeline;

  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  pipeline = gst_pipeline_new ("mp3 player");
  source   = gst_element_factory_make ("filesrc",  "file reader");
  filter   = gst_element_factory_make ("mad", "MP3 decoder");
  sink     = gst_element_factory_make ("alsasink", "ALSA output");

  g_object_set (G_OBJECT (source), "location", “music.mp3”, NULL);

  gst_bin_add_many (GST_BIN (pipeline), source, filter, sink, NULL);
  gst_element_link_many (source, filter, sink, NULL);

  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, bus_call, loop);
  gst_object_unref (bus);

  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  g_main_loop_run (loop);

  return 0;
}
```

*Code Sample 2: MP3 player main logic*

There are many of resources describing GStreamer details. See the *References* section for a list.

# Vala Pipeline Example

GStreamer is based on GObject, the object model in GLib. The wide acceptance of GObject, driven by GNOME, brought about the development of Vala, a C# like programming language. Vala supports objects, which are compatible with GObject, thus providing programmers with syntax support for the GObject type system. Vala compiles a vala source file into a C source file.

```
using GLib;
using Gst;

public class ValaExample : GLib.Object {

  public void run (string[] args) {
    MainLoop loop;
    Element src;
    Element filter;
    Element sink;
    Pipeline pipeline;

    loop = new MainLoop (null, false);

    Gst.init (ref args);

    pipeline = (Pipeline) new Pipeline ("mp3 player");
    src = ElementFactory.make ("filesrc", "file reader");
    filter = ElementFactory.make ("mad", "MP3 decoder");
    sink = ElementFactory.make ("alsasink", "ALSA output");

    pipeline.add_many (src, filter, sink);
    src.link (filter);
    filter.link (sink);

    src.set ("location", "music.mp3");

    pipeline.set_state (State.PLAYING);

    loop.run ();
  }

  public static int main (string[] args) {

    var example = new ValaExample ();
    example.run (args);
    return 0;
  }
}
```

*Code Sample 3: Vala MP3 player*

If you are developing an object oriented application that uses GStreamer, you might consider using the vala programming language.

# Performance Concerns

The example so far has been trivial, just enough to give you the flavor of GStreamer sources, filters, sinks connected in a pipeline. Let's say you are developing a video phone with audio and video streaming in both directions. You can create two pipelines for mux-ing and dex-ing the audio and video streams, running each of the streams (now up to 4) through compressors / decompressors, all in a just a few hundred lines of code. If you have a big enough CPU, maybe it will keep up, but not likely. To overcome the performance issues, processors for embedded multimedia devices, like Texas Instrument's OMAP3, have a variety of hardware to improve performance.

## Using On-Chip Hardware Accelerators

When running GStreamer on an OMAP3430 the CPU load and overall system performance is dependent on how effectively the GStreamer elements utilize the OMAP3430 hardware capabilities. Using general purpose GStreamer filters allows any data conversion available in a GStreamer plug-in to be usable in an OMAP3. All the encoding / decoding is being done by Cortex A8 processor. Any A8 optimizations supported by the GNU toolchain will help performance. This is the baseline performance. We can improve on the baseline by taking advantage of the on-chip hardware accelerators.

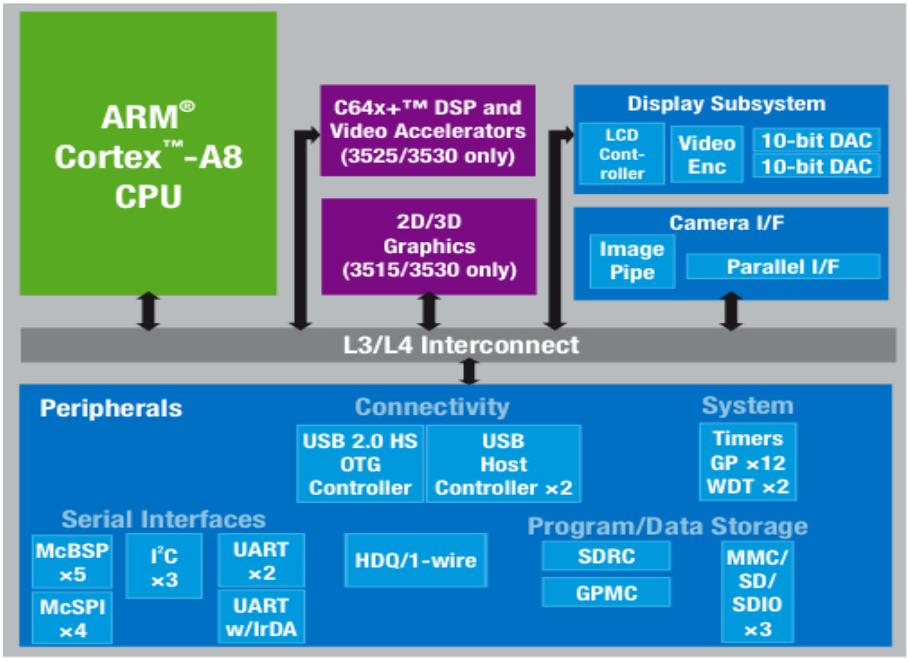



*Figure 2: OMAP3 Architecture*

Some of the GStreamer codecs are being tuned to use the NEON Single Instruction Multiple Data (SIMD) instruction set. NEON is designed specifically for media and signal processing. The ffmeg library, available as a GStreamer plug-in, can be built to use the NEON instruction set. Utilizing the Cortex A8 NEON hardware can lower CPU requirements while leaving the other OMAP3430 hardware accelerators available for other tasks.

In another incantation, the encode / decode can be done using the C64 digital signal processor (DSP). The data is routed though the C64 for conversion, again freeing up the A8 for other tasks. A variation on this approach is to have the algorithm running on the C64 DSP take advantage of the video hardware accelerator that is part of the OMAP IVA2 macrocell. Using the video hardware accelerator increases the video image size the C64 is able to process in real-time. The application software complexity to directly use the C64 and video hardware accelerator is significant. In the past most companies purchased proprietary software in order to utilize the DSP for multimedia data processing.

The above simplified description of one way to take advantage of on-chip hardware accelerators uncovers the many challenges involved. We went from using a simple GStreamer pipeline created using general purpose GStreamer plug-ins, to needing plug-ins customized to use the OMAP3430 hardware. How is bus contention in the OMAP3 handled? How do you minimize moving the stream data between the different OMAP3 subsystems? How is cache coherency maintained? How do the different hardware accelerators work together when they may be in different elements in the GStreamer pipeline? As you will see shortly, a software framework is now available that addresses these needs, thus simplifying the use of the DSP by Linux applications.

### Minimizing Data Copies

As the audio and video data streams move from source to sink, going though various filters in the GStreamer pipeline, general-purpose GStreamer filters will often read data from a GstBuffer, process the data, and store the results in another GstBuffer. So far, so good. But what happens when a hardware accelerator is used? The `gst_buffer_alloc()` method for GstBuffer creation simply `malloc()`s space in virtual memory. The hardware accelerator need the data in contiguous physical memory, thus requiring a data copy if the buffer is in virtual memory.. Another case is when multiple filters are running on the C64, like a decoder filter and an equalizer filter. If these are two filters work completely independently, the data will be stored in GstBuffer as the data moves between the two filters. However, an optimization is possible where the stream data is kept local to the C64 and each data chuck is processed by both filters before being stored back into a GstBuffer.

GStreamer designers recognized the performance issues with unnecessary data copying and endeavored to create a streaming multimedia framework that does as little data copying as a highly tuned targeted application. One improvement is supporting the `gst_pad_alloc_buffer()` mechanism to allocate a buffer in addition to the more general `malloc()` based `gst_buffer_alloc()` function. `gst_pad_alloc_buffer()` allows the sink pad to allocate a buffer in a manner optimal for the sink, and only if that is not available uses `malloc()`. Since the sink pad knows what type of memory is needed for optimal performance, this provides a simple mechanism to get the steam data in the right type of memory the first time, thus avoiding data copies. Minimizing data copies is key performance requirement.

## GStreamer and Texas Instruments DMAI

In one multimedia application development approach there is a simple extensible framework called GStreamer that makes it easy to create pipelines for processing streaming media and another approach is a what appears to be a highly coupled, processor specific tuned solution taking advantage of all the on-chip hardware accelerators. To simplify multimedia application development that takes advantage of available hardware accelerators, Texas Instruments developed DMAI, the Davinci Multimedia Application Interface.

Texas Instruments created a GStreamer plug-in to maintain the easy-to-understand and widely used GStreamer pipeline model utilizing the high performance DMAI sub-system. DMAI, a library (including source code) is available now from Texas Instruments and the GStreamer DMAI plug-in is in develop-

ment at the time of this writing. Check the Texas Instruments website for availability.

For chips containing both an ARM processor and a DSP, the software components used are shown in Figure 3 *GStreamer with DMAI*. The DMAI layer understands Codec Engine and the multimedia related OMAP3 device drivers. Codec Engine provides a consistent API to audio and video codecs independent of what hardware transforms the data. For the case of a video codec running on the C64 DSP using the video hardware accelerator, Codec Engine uses DSPLink to exchange data with the codec algorithm running on the C64 DSP. The algorithm uses the DSP BIOS operating system and can directly control the video hardware accelerator. If an application writer attempted to use the C64 based video hardware accelerator directly, you can see many, many hardware details that would need to be taken into account, including cache coherency, DMA, C64 data exchange, and video hardware accelerator usage.. Using the appropriate level of abstraction, either at the DSPLink, Codec Engine, or DMAI API allows the right balance between code portability and hardware acceleration. With the GStreamer DMAI plug-in, the hardware accelerators can be utilized by a GStreamer aware application without the application having to directly control the hardware.

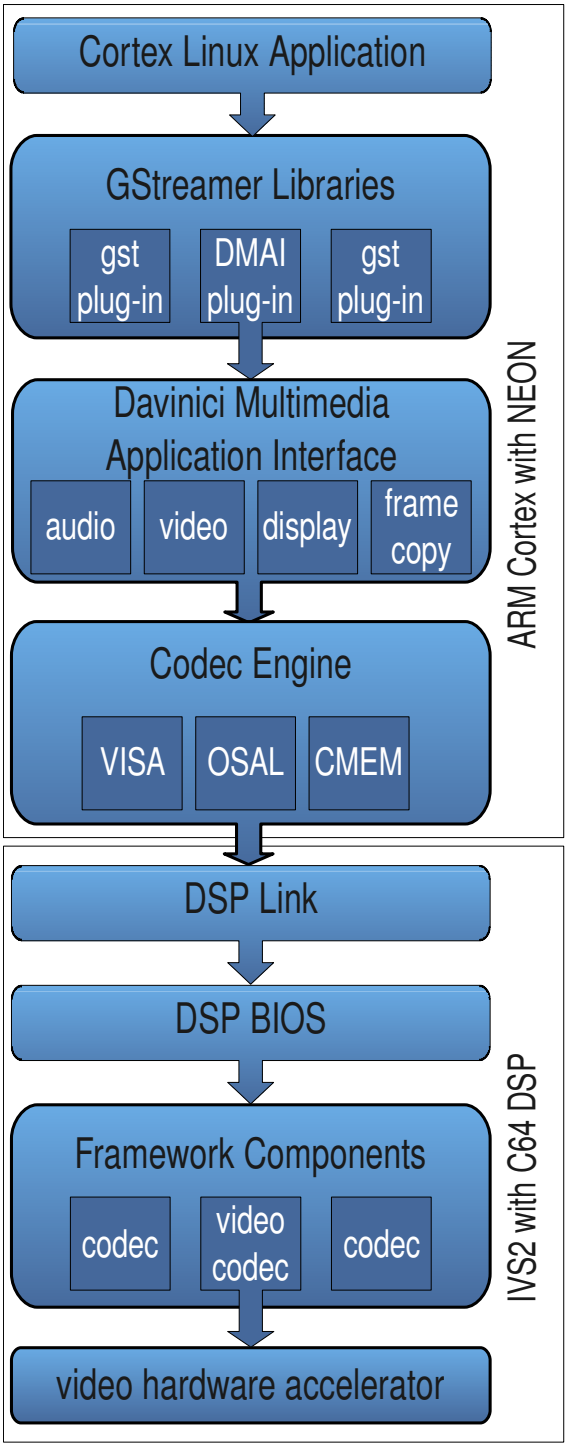



*Figure 3: GStreamer with DMAI*

DMAI introduces a buffer abstraction that allows a reference to a data buffer to moved from one multimedia processing subsystem to another, minimizing the need for a data copy. If a data copy is required, the DMAI `Framecopy` module will perform the copy in an optimal manner for the hardware available at the time the copy occurs. DMAI coupled with Codec Engine and Linux device drivers allows arbitrary processing pipelines to be created in an efficient manner. The GStreamer plug-in allows this underlying high performance flexible multimedia pipeline to fit within the standard GStreamer framework. Applications using the DMAI enabled plug-in get the best of both worlds – GStreamer pipelines and fast data processing.

# GStreamer and OpenMAX

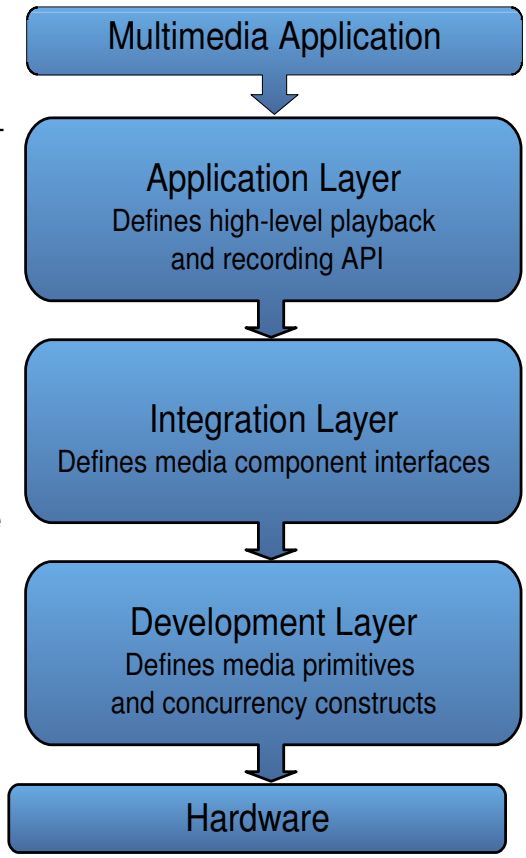



*Figure 4: OpenMAX Architecture*

OpenMAX was developed for embedded systems as a way to provide key abstractions to the three important aspects of multimedia data handling. Using GStreamer terminology, the 3 layers are application, data filters, and hardware acceleration. Using OpenMAX terminology, the 3 layers are application (AL), integration (IL), and development (DL). The application layer exposes the multimedia framework to applications. The integration layer has the pipeline building blocks including sources, sinks, and filters. The development layer provides the low-level building blocks allowing, for example, codec filters to used optimized routines tuned for a particular hardware platform. OpenMAX enables vendors of codecs to use hardware optimized routines without knowing anything about the hardware.

At first glance, OpenMAX, like DMAI, appears to offer an API similar to GStreamer in that all 3 frameworks allow arbitrary multimedia pipelines to be created dynamically. Are OpenMAX, DMAI and GStreamer competing technologies? In some sense, yes. If you analyze OpenMAX (or DMAI), and identify all the multimedia functionality your application requires is available, GStreamer may not add much value. However, if you need to break apart MPEG packaged audio and video streams, transfer multimedia data over the network, or perform some other function supported by the over 200 GStreamer plug-ins, then you may want to base your application on the GStreamer framework.

There is a `gst-openmax` GStreamer plug-in for hardware with OpenMAX support. The OpenMAX plug-in maps integration layer standard components to GStreamer elements. OpenMAX is in development and is usable, but not production ready. The current OpenMAX focus is on encoding and decoding filters. In the Bellagio integration layer effort, many of these filters are based on the same open source libraries used by GStreamer elements. The difference is the heavy lifting is passed across the OpenMAX development layer to use hardware optimized routines.

There is work underway at the development layer as well. For ARM processors, ARM Inc. published a development layer implementation that takes advantage of the NEON instruction set. Texas Instruments has an OpenMAX development layer implementation that uses algorithms running on the C64 in the OMAP3 processor. At the time of this writing, AAC decode, MPEG4 decode, and an A/V player with MPEG4 AAC decoder are implemented. Source code is available from the OMAP zoom website.

# References

GStreamer – http://www.gstreamer.org
OMAP3 – http://www.ti.com/omap3
Vala – GStreamer support: http://www.vala-project.org
DMAI – http://wiki.davincidsp.com/index.php?title=Davinci_Multimedia_Application_Interface
OpenMAX – http://www.khronos.org/openmax
GStreamer and OpenMax – http://freedesktop.org/wiki/GstOpenMAX
OMAP3 and OpenMax – https://omapzoom.org/gf/project/openmax
Presentation Examples with Makefile: http://tfischer.public.ridgerun.net/rr-gst-examples.tar.gz

# About the Author

Todd Fischer has focused on embedded Linux since 2000, specializing in developing USB, SD, audio and video drivers for custom consumer electronics hardware. Prior to 2000, Todd was with Hewlett-Packard working on embedded software for LaserJet printers. Todd has 20 years experience defining system architectures for embedded devices, and holds a BS in Electrical Engineering / Computer Science from the University of Colorado, Boulder and a MS in Electrical Engineering / Computer Science from University of Minnesota, Minneapolis. Todd can be contacted at todd.fischer (at) ridgerun.com.