WHITE PAPER

Jason Kridner,
software architecture manager

Nick Lethaby,
software manager

Steven Magee,
software architect

Erik Welsh,
senior applications software architect

Henry Wiechman,
software product manager

Texas Instruments

TEXAS INSTRUMENTS

Embedded software development is a multi-dimensional effort

Overview

Designing, developing, integrating and deploying a complete software system for an embedded application involves a large number of interrelated questions and optimization tradeoffs. With proper planning, such a challenge can be efficiently mastered resulting in a highly differentiated solution.

Making any decision without regard to the effects it will have on the rest of the development project is too often inefficient, potentially causing adjustments to previously completed work or outright re-work. At the most fundamental level, a very basic question typically drives many of the decisions that will be made over the course of an embedded software development project. That is: What is the goal of the project? The simplicity and obvious nature of this question belies the complexity of a complete response. A firm grasp on the goals of the project implies a complete understanding of the factors that will make the end product a success in the marketplace. And it is these factors that will drive many of the decisions made by the software development team.

So, for example, the amount of product differentiation required by a project will have an impact on the product's time to market and price point. And both of these factors will affect the decisions made by the software development team regarding the type of integrated development environment (IDE) adopted for the project, whether open source or commercially available software resources will be integrated into the system, the kind of operating system (high-level general-purpose or real-time?) which will be deployed and many others. Without a thorough understanding of the project's goals to guide the process, the considerations that must be addressed by the embedded software development team can be overwhelming.

To help navigate your embedded software design, there are several topics to help with your project:

- Operating systems
- Graphic programming and user interface development
- Security
- Vendor tools: IDEs, EVMs, and SDKs
- Open source
- Code structure: Reuse, customization and debugging
- Middleware frameworks and application-specific tools
- Multimedia programming
- Embedded multiprocessing: Multi-processor, multi-core and multi-threaded programming

Operating systems

One of the first questions to come up for the software design team will be whether an operating system (OS) is needed by the application. Some embedded applications are quite simple and limited in the functionality they system provides. This type of application may not need an OS at all.

Some manufacturers offer software not requiring an OS for easy programming. Microcontrollers do not use a high-level operating system and have a very simple programming interface that works with third-party real-time operating systems (RTOS). TI offers such

a tool for its ARM® microprocessors and DSP-based processors than enables programming similar to a microcontroller. This ARM and DSP StarterWare provides C-based, no-OS platform support for the ARM and DSP platforms of devices. StarterWare provides device abstraction layer libraries, peripheral programming examples such as Ethernet, graphics and USB, and board-level example applications. StarterWare can be used stand-alone or with an RTOS.

Should an OS be required, today's alternatives are much richer than they have been in the past. An OS like Windows® CE might be purchased or an open source OS such as Linux® could be implemented. Recently, royalty-free but closed OSs have been introduced. An example of this latter type is the Android™ OS from Google. Although Android is available without cost from Google, its content and architecture are controlled by that company.

Decisions by a design team concerning an OS are of paramount importance because the OS will dictate the general outline that software development will follow. For example, if a commercially available OS is implemented, developers will be dealing with a vendor or third-party vendors for tools, software modules and ready-to-integrate subsystems. If an open-source OS is chosen, the design team will interact with the OS's open source community, often without the support of a vendor. Closed royalty-free OSs like Android offer a little of both alternatives.

Commercial OSs vs. open source vs. royalty-free

Commercial OSs like Windows CE and others are purchased on a royalty basis from vendors such as Microsoft. Developers will choose a commercial OS for a number of reasons. Technical support and training from a large and well-established corporation can shorten the learning curve for a design team and speed the development process when those inevitable difficulties and setbacks are encountered along the way. Because a commercial OS is controlled by one company, third-party tools, software cores and plug-ins are typically evaluated and certified by the OS company, easing and simplifying the integration of third-party software into the larger software system.

An open-source OS will be obtained free of charge from the community responsible for it. Linux is the most prominent example of an open-source OS. The community sustains the OS and contributes plug-ins and add-ons that can be integrated with the OS. When Linux is adopted by a software design team, the team takes on greater responsibility for evaluating and integrating third-party sources of software into the system than it would with a commercial OS. One benefit of an open source OS like Linux is the availability without a procurement cost of creative and innovative software modules from the community. Many creative software developers have been attracted to communities like the Linux community, and they freely and regularly contribute their work to the rest of the community. Of course, the software development team that has adopted an open-source OS ultimately becomes responsible for integrating all of the open-source code and ensuring it will work properly together.

Some OS vendors such as MontaVista, Red Hat, RidgeRun, Timesys, Wind River and others have made

commercial versions of Linux® available. This type of OS provides the software team with an OS that is open to third-party code and supported by a software vendor.

The Android™ OS, which is available royalty-free from Google, is built on Linux but its content and how various plug-ins are integrated into it, such as its various multimedia processing capabilities, is controlled by Google. This restricts the flexibility of developing software for Android, but it can accelerate the development process as a whole by simplifying and streamlining the integration of third-party code when compared with the effort that is required for this under an open-source OS like Linux.

Real-world software development projects will typically incorporate a mixture of several types of OS-related code. Deciding on a commercial OS does not rule out the implementation of open-source third-party plug-ins. Nor does deploying an open-source OS like Linux eliminate the possibility of a commercially available software module.

Real-time vs. high-level OSs

Another important issue which affects the type of OS chosen will be whether the embedded processing application will be a real-time system or a general-purpose application. Some OSs have been designed and developed specifically for real-time systems where predictable and pre-determined response times are critical. In fact, the deterministic responsiveness of some real-time systems can be a matter of life or death. For example, an embedded braking system in an automobile must be able to respond immediately when the brake is applied. A general-purpose OS would typically place a command in a queue and service it in due course, but a real-time OS must be able to recognize the urgency of certain commands and process them immediately.

Because real-time OSs (RTOSs) place a premium on responsiveness, they typically do not support the wide range of tasks that general-purpose high-level OSs (HLOS) like Windows® do. The tasks supported by RTOSs are typically limited to a narrow range of time-sensitive tasks that dominate the content of real-time systems. A HLOS fulfills a different set of requirements.

Ease-of-use is critical for systems that employ a HLOS. As a result, HLOSs have a wider and deeper abstraction layer than RTOSs between the user and the computing or communications system itself. A HLOS automatically performs for the user many more tasks than the relatively few tasks performed by a RTOS. This expands significantly the code footprint of a HLOS, making it much more complex than a RTOS. In turn, these characteristics make a HLOS poorly suited to real-time processing applications where predictable response times is a requirement.

High-level OS	Real-time OS
Windows CE (Microsoft)	VXWorks® (WindRiver)
Android (Google)	Nucleus® (Mentor Graphics)
Linux (open source and commercial suppliers)	QNX®/Rim
	Green Hills

Graphic programming and User Interface Development

When you begin your design project, the user interface is a very important aspect. Considering how the graphical user interface (GUI) will appear is key in the beginning stages. Most likely you will be utilizing some graphics for this development. Graphics processing has become just as important in embedded systems as it is in consumer applications. The human machine interfaces (HMI) or GUIs that are prevalent on consumer devices have conditioned the expectations of users of embedded systems to the point where any interface short of a very graphical and intuitive UI will certainly fail to engage users effectively and this will impact the system's ease-of-use and overall operational effectiveness. In the end, the system's adoption rate in the marketplace can be adversely affected by an inadequate UI.

Graphic processing for embedded systems consists of three distinct tasks: creation, composition and display. That is: the creation of the graphical elements such as windows, icons and others; designing the composition of the various windows, desktops and other graphical elements (i.e., what will they look like on the screen and how the images will change when the system is operating); and actually displaying these elements on a display of some sort (i.e., moving the graphics from memory to the display).

Graphic design and processing can be a specialized art. It certainly has become more complicated as UIs have become more sophisticated. How the software development team will develop the UI for a particular embedded system will depend greatly on the hardware capabilities of the system. For example, a great number of consumer systems feature a dedicated graphics/multimedia processor, while many, if not most, embedded system will not. Embedded systems without a dedicated graphics processor likely will require acceleration in hardware to provide a high level of perceived responsiveness. Accelerators are able to offload the graphic processing load that might otherwise burden the system's main CPU and thereby improve the graphic responsiveness of the system. After all, the responsiveness of the system's UI can dictate the perception of responsiveness on the part of the user for the entire system. That is, an unresponsive UI can impart the impression that the entire system is operating poorly.

A critical consideration for the design team will be whether a differentiated HMI/GUI will be absolutely necessary for the success of the application in the marketplace. Of course, the UI on many consumer-oriented devices and systems will play a huge role in a product's success, but short of this level of importance, many embedded UI designers will be faced with tradeoffs involving the cost implications of various graphics and display options, as well as identifying the optimum combination when it comes to system cost, functionality and performance goals.

Other issues that the embedded software development team should take into consideration will involve whether the software team itself or professional UI designers will develop the UI, and how much memory will be available in the embedded system.

Graphic programming tools

A wide range of both open source and commercial development frameworks and tools is available for HMI/GUI projects.

OpenGL® is a widely adopted cross-platform graphics framework that is supported by the Khronos Group (**khronos.org**). It features an extensive number of both 2D and 3D graphics routines and is independent of any one OS.

Several graphic programming frameworks are also available for the Linux® OS, including Qt, X-11, GTK, and DirectFB. Each framework enables a different set of implementation choices and level of abstraction from the hardware. However, developers should note that the Linux kernel itself does not contain any of these frameworks. As a result, integrating any of these frameworks, building components and executing on a certain embedded processor can be time-consuming for inexperienced developers.

In addition to open source software, several commercial embedded operating systems, such as Microsoft Windows® Compact Embedded, Wind River VxWorks® and QNX® can include a GUI framework. Moreover, OS-independent GUI frameworks like Mentor Graphics' Inflexion can run on both open source and proprietary operating systems.

Security

How secure are you aiming to make your embedded application? Is security needed at all? The susceptibility of all electronic devices to hacking, viruses and all sorts of malware – as well as the subsequent liabilities of equipment manufacturers – will continue to be a major concern afflicting the industry for the foreseeable future. Embedded systems that store consumers' private information certainly are not immune to identity theft, tampering, malevolent attacks and annoying pranks. The reliability of other types of embedded systems, such as certain medical systems and communications for emergency responder, are certainly critical to personal and public safety. As a consequence, embedded software developers must plan to protect their systems by safeguarding their operations and information their store by designing in appropriate security measures from the very beginning of a design project.

Understanding the goals of hackers is imperative to protecting embedded devices, many of which are placed in very vulnerable locations far removed from centralized security facilities. This only increases the need for strong and effective security measures. By asking the following questions the embedded software development team can focus its efforts on deploying and developing the appropriate means to adequately protect where protection is most critical. Of course, the answers to these questions will vary considerably, depending on the end-user application. For example, point-of-sale terminals will protect credit card information stored on the terminal and communicated over the communications channels connected to the terminal. Media players must ensure the security of digital copyrighted content from pirating. And the end-user voice and data communications infrastructure must protect communications channels from identity theft and unlawful intrusions on privacy.

- What are you trying to protect?
- Who are you trying to protect against?
- What will be the cost if the information you are trying to protect is compromised?

In-depth answers to these questions will enable informed and effective decisions. The development team will discover what data must be kept secret and which communications channels should be secured. The presence of third-party applications in the system must also be addressed and its security implications fully discussed.

Since all embedded systems do not require the same level of security, different techniques are often employed in some applications but not in others. For mass-market consumer applications, cryptographic acceleration, securing the boot process and protecting the system's debug channels comprise a minimum level of protection. Beyond these techniques, other embedded applications may require additional measures for securing their run-time environment, to discourage tampering with voltages and clocks and to safeguard various communications channels.

Vendor tools: IDEs, EVMs and SDKs

As you begin your development, there are several software development tools and resources available from technology suppliers, including integrated development environments (IDEs), evaluation modules (EVM) and software development kits (SDK) to assist with software development. While these vendor tools are certainly options, they can drastically reduce development time and provide a test platform from which to begin.

Integrated development environments (IDE) have become quite prevalent in the industry. Open source, commercial, and vendor-supported IDEs are available in today's marketplace. An IDE gathers a number of tools into one environment. Learning the multiple tools that will be required by a typical embedded software development project is simplified since the tools in an IDE usually share the same user interface, naming and command conventions, and other aspects of their overall operations. Moving data from one tool to another within an IDE should be easier and more intuitive.

Unfortunately, not all IDEs are created equally. Most probably have the basic code generation tools for assembling and compiling software, but the tools in some IDEs may not be as tightly integrated with each other. If the transition from one tool to the next is very time consuming, the efficiency of the entire software development process can be adversely affected.

The operational performance of the IDE should also be a concern. Certain benchmark programs can provide a basis for comparison in this regard. Alternatively, a quantity of existing code could be run through the various IDEs under consideration to determine the best performing environment. Moreover, the visibility into code under development can differ widely among IDEs. Greater visibility enables faster and more efficient code development by programmers.

An IDE developed and supported by a single vendor may be limited by the amount of support the vendor devotes to it. Some vendors have chosen to base their IDEs on an open source, royalty-free environment such as Eclipse, which has been developed and is supported by the non-profit Eclipse Foundation (eclipse.org). Funded by membership dues, the Eclipse Foundation oversees the evolution of the environment and its ecosystem of open source resources.

Eclipse is a multi-language IDE capable of supporting software development in several languages, including Java, Ada, C, C++, COBOL, Python, Ruby and others. Google has based its Android™ development tools on Eclipse as has Texas Instruments its Code Composer Studio™ IDE. As a result, a software development project targeting an Android platform would be able to take advantage of tools from Google, TI or both.

Most EVMs are comprised of a small printed circuit board (PCB) incorporating the device or devices that are being evaluated and a modicum of software resources. The software resources supporting an EVM can vary greatly. EVMs with more comprehensive software support are able to be set up quickly so that designers can experience a demonstration soon after the EVM has been removed from its box. The more complete EVMs come with SDKs containing IDEs and access to code libraries and support for high-level operating systems (HLOS) such as Linux®, Android or Windows® CE.

SDKs are often offered free of charge by the chip vendor or they can be purchased from the supplier. The baseline components in an SDK are usually certain code generation tools, while the more extensive SDKs will resemble a complete software reference design for a certain application. With this type of SDK, many of the constituent parts of system software will already be in place, such as a multimedia framework with its codecs, and the input/output (I/O) stacks of code which control I/O ports like USB, Ethernet and others. In this type of scenario, the software development team is able to focus on the top-level application software for the system.

The downside of an extensive SDK like this is the constraints it often places on developing innovative functionality intended to differentiate the product in the marketplace. This type of extensive SDK will be composed mostly of object code, and developers will have little access to the underlying source code to make modifications and customized improvements. In contrast, an SDK made up of more source code can afford developers the opportunity to modify and customize the code, as long as the licensing agreement grants developers the right to do so.

An SDK that would enable easy customization should also include structured and well-documented application programming interfaces (APIs) at each level of the system software. An effective API will facilitate and simplify the programmability and re-configurability of the supporting software architecture. The API should provide “hooks” at each layer of the system’s software so that developers can easily plug software modules into the system without having to write entirely new modules. This reduces software development time significantly and accelerates a new product’s time to market.

Vendor support for open source software development

The choice of open-source software development will require some porting by the software team. First, the tools and frameworks must be ported to the particular processor chips deployed in the target system or the processor vendor must have already completed such a port. In the case of TI, a wide range of open source middleware frameworks and tools have already been ported to its devices. This allows development to begin immediately. The following are some of the open source software resources ported to TI microcontrollers and microprocessors.

- OpenMax
- GStreamer
- Eclipse IDE (TI's Code Composer Studio™ IDE is based on Eclipse)
- OpenGL®
- OpenCV
- Yocto
- Linaro

TI's arowboat.org

For embedded development efforts that have implemented the Android™ OS on TI ARM® processors, TI maintains a central repository of Android software, including the OS itself as well as middleware frameworks and other tools that already have been ported to its chips for embedded systems, including the TI ARM microprocessor, DSP+ARM processor, DSPs, digital media processor and OMAP applications processor platforms. This portal is located at **arowboat.org**. It highlights a number of useful features for software developers, such as a community wiki, forums for technical discussions and a centralized repository for source code relating to Android deployments on TI's ARM-based processors.

Tool chains for embedded systems: Yocto and Linaro

A number of open-source toolsets or "tools chains," as they are often referred to, have been assembled for embedded system development.

The Yocto project (**yocto.org**) is an open-source collaboration that provides tools and methods for developing Linux-based embedded systems regardless of the hardware architecture. The Yocto project is a function of the Linux Foundation. Some of its participating organizations include TI, Dell, Mentor Graphics, MontaVista Software, Panasonic and others. Yocto strives to provide a complete embedded Linux environment that can carry forward as product platforms evolve in the future.

Linaro (**linaro.org**) is a non-profit open software engineering/open source enterprise that was founded by leading technology providers, including TI, ARM, IBM and others. The organization is dedicated to developing Linux software development tools and foundation software for ARM-based systems-on-a-chip (SoC), many of which are deployed in embedded systems.

Code structure: Reuse, customization and debugging

As you begin programming your embedded application, code structure is an important element. Considering how you structure your code in the beginning can influence future software investments, the customization of your embedded design and ability to easy debug possible software issues in your design.

Re-usable code

The return-on-investment (ROI) on software development increases as code, modules and even entire software systems are re-used in multiple applications. For example, manufacturers of embedded processing systems usually adopt a product line strategy for the systems they bring to market. This would involve offering multiple systems covering a wide range of price points with varying degrees of functionality, processing power and overall capabilities. What might be considered a less expensive entry-level system would likely require much less processing power than the top-of-the-line high-end product offering. To maximize its ROI on software development, the manufacturer would want to re-use as much software as possible across all of the multiple products that make up its product line strategy. As a consequence, code re-use is typically a high priority and this raises several issues for software development teams.

Theoretically, each step up in system capabilities in a product line likely would involve a step up in processing power and, consequently, a different, more powerful processor. To ensure optimum software re-use, the manufacturer's software teams would be concerned about software compatibility among the various processors in the product line. Ideally, software developed for one processor would port to and be re-usable on all of the other systems in the product line. To approach this goal, a system manufacturer must carefully consider the breadth and depth as well as the software compatibility of the processors offered by its processor supplier. Software must be portable across processors so that the system manufacturer can achieve a satisfactory ROI on its software development.

Tools also play a role in re-using software across multiple processor platforms. For example, compilers for relatively high-level languages like C will allow a degree of code re-use. Often, an equipment manufacturer will alter its software systems in order to optimize a product for a certain market segment. Well architected and fully documented APIs also ensure the scalability of software over multiple platforms and processors. Unfortunately, some APIs can become quite large and bloated in terms of their code footprint and this can adversely affect system performance.

In the end, a high level of software re-use will be achieved when the system manufacturer's software development teams have a full range of tools, software components, wrappers and other capabilities at their disposal. Then they will be able to easily mix and match previously developed code, adding or subtracting product features and functionality to quickly meet the requirements of a particular product offering.

Code customization

One of the major goals of many software development projects is to provide distinctive capabilities or differentiated features for the end product. Then, once the system reaches the marketplace, it will stand out from its competitors and achieve greater success. Software development teams must be cautious when they are developing new differentiating capabilities and integrating these features with open-source software. Unless the software system is properly partitioned and structured to meet certain requirements of some open-source licenses, any new code, including the differentiating capabilities developed by a manufacturer's

software teams, may be considered open-source code. As such, the system manufacturer may be obliged to contribute the new code back to the open-source community. Carefully partitioning the system's software architecture can reduce the risks of this happening.

Some technology companies have taken steps to avoid these types of unpleasant surprises altogether. Software from TI, for example, comes with a "software manifest" which fully explains and delineates the sources of the code and any third-party or open-source licenses or obligations associated with it. Furthermore, these software manifests and the software itself are reviewed by the company's Open Source Review Board to ensure the code is properly structured and architected for the protection of the system manufacturer.

Debugging

Very few – if any – software projects function properly without some amount of debugging to track down and eliminate unforeseen faults and failures. In fact, fully functional software will be achieved faster if it is designed in such a way that it facilitates the debugging process. The architecture of the software system and the composition of individual processing nodes can be structured to give debugging developers as much visibility into the code as possible. Developers can then isolate faults more or less precisely and quickly fix the questionable segment of code.

Processor suppliers can also provide emulation-based debugging tools. TI, for example, offers a full line of emulators which relate the state of the processor to the software it is processing. In addition, these emulators function in concert with TI's IDE, Code Composer Studio™, to quickly identify the source and location of bugs, and develop fixes.

Middleware frameworks and application-specific tools

A great wealth of resources is available to assist embedded software developers. Of course, the particular OS and the type of OS chosen will have a bearing on the software resources that will be deployed throughout the project, including various middleware frameworks and other kinds of tools. For example, resources relating to a commercial OS like Windows® CE typically will be limited to the supplier of the OS or authorized third-parties. In contrast, developers working with an open-source OS like Linux® will have to evaluate and choose from the universe of open-source frameworks and tools that are available from the Linux community. In addition, the project team may have to port open-source resources to its hardware, although some hardware vendors such as TI have undertaken this task on behalf of its users. Since the royalty-free Android™ OS is based on Linux, most open-source resources also can be applied to Android-based projects. In addition, proprietary software development resources are available from chip vendors like TI, as well as other technology suppliers.

The use of open-source middleware frameworks is becoming increasingly popular among software developers. Several middleware frameworks such as GStreamer, Open CV and OpenMax have become somewhat pervasive in the industry. Open tool chains, like Yocto and Linaro, have also emerged.

A middleware framework is a set of predefined and fully developed software functions or tasks. Developers of a larger software system are able to select tasks from a middleware framework to deploy in their system.

The framework comes with a set of rules for integrating each task with the others that are also compatible with the framework so the time and effort devoted to integration is minimized.

The typical middleware framework is specialized to a certain type of processing. For example, GStreamer is focused on multimedia processing. A development team that is deploying a multimedia processing stack can choose from the various modules in the framework, such as a video decoding module or a plug-in for audio processing.

Many of the software development kits that accompany TI's microcontrollers and microprocessors have already been ported to various middleware frameworks. This eliminates one step in the development project, allowing software teams to begin integrating the tasks from the most popular middleware frameworks immediately without porting the framework to their hardware platforms.

Multimedia programming

Generally speaking, multimedia programming will involve processing of video, audio, imaging and even graphics. Some of the overriding issues to consider will include which codecs (coders/decoders) to support, what I/O interfaces are required, the system's real-time performance requirements, the need for concurrency in the system, the number of channels, bit rates, frame rates, display resolution, quality of the multimedia content and, lastly, memory size and bandwidth will have an effect on all of these aspects of multimedia programming.

Many systems process video and audio content and output it to audio/video (A/V) I/O ports. Audio and video drivers may be provided by the processor vendor, but the embedded programmer still should be familiar with the various A/V interfaces in the event a problem arises that requires debugging. Audio I/O interfaces comes in many analog and digital forms. For analog audio, analog-to-digital (A/D) converters, digital-to-analog (D/A) converters or some combination of these in the form of a codec may be part of the system. For digital audio, SPDIF (Sony/Philips Digital Interconnect Format) and HDMI (High-Definition Multimedia Interface) are common interfaces. Analog video I/O interfaces include RF (radio frequency), Composite, S-Video and Component. These analog video streams are converted to a digital form by video decoders that are made up of video A/D converters. For monitor displays, video encoders convert digital video to an analog stream for the display. The most common digital video interfaces are HDMI, DVI (Digital Video Interface), DisplayPort, and LCD (Liquid Crystal Diode). Some systems require a camera interface. This could be a CMOS sensor, smart sensor or USB camera interface. These interfaces sometimes are integrated into a system-on-a-chip (SoC) device, but not always. When they are not, the system designer must choose from several components to implement the interfaces required. Then, the embedded programmer must ensure that the proper drivers have been included for the components chosen.

Many embedded systems must be capable of processing compressed audio and video data. If this is the case, system designers and programmers should consider how this will be accomplished. Various sources in the system can provide the compressed multimedia content, including the system's file system itself, where compressed content can come from the hard disk drive, a USB flash device, or a Secure Digital/Multimedia

Card (SD/MMC). Or, the compressed bit stream can be sourced from a network connection, such as a wired Ethernet interface or a wireless LAN connection like WiFi™. Typically, the system's software must be able to parse a particular 'container' format, which is associated with a certain multimedia compression algorithms. Some of the more prominent container formats in use today are Transport Stream (TS), Program Stream (PS), Audio/Video Interleave (AVI), Motion Pictures Experts Group-4 (MPEG-4), Advanced System Format (ASF) and others. Each of the various containers store compressed audio, video and meta data in its own particular format. The software modules that process these containers usually are referred to as parsers or demuxers because one of the primary functions they perform is demuxing the audio and video data. Once the bit stream is demuxed, the resulting compressed data buffers are passed to audio and video decoders for decompression.

There are a number of prominent multimedia codecs that programmers should be aware of (See Table 1 below.)

Table 1. Prominent multimedia codecs

Multimedia type	Codecs
Video	H.264 (AVC), MPEG-4, MPEG-2, VC1, VP6/7/8, RealVideo
Audio	AAC, MP3, WMA, MLP, AC3, Vorbis
Imaging	JPEG, PNG, GIF, TIFF
Speech	G.7xx (There are several G.7xx codecs that are deployed in voice and voice-controlled applications)

Memory issues

Many embedded applications today implement a type of Double Data Rate (DDR) memory. To optimize the cost of the system, the designer is faced with a tradeoff between memory cost and bandwidth. That is, the designer will want to implement the least costly memory that provides adequate bandwidth to meet the multimedia processing requirements of the system. As a result, the designer must fully understand the system's DDR bandwidth requirements.

Video processing typically consumes much of the DDR bandwidth while audio processing consumes relatively little. Transferring a video stream to an I/O interface, decoding, encoding, post processing and displaying video all require large amounts of DDR bandwidth. Programmers should be aware that at high definition resolutions buffer copies should be avoided because these might force the system to exceed its DDR bandwidth limits. This could result in lower video frame rates or video display anomalies that contribute to a poor user experience.

Graphics/Video blending

In many applications, graphics and video must be blended or combined for presentation on the user display. Some embedded processors have a display subsystem capable of performing this blending in hardware, but when this is not the case, a 2D graphics engine often blends the video and graphics. Still other applications

implement video as a 3D texture. For this, the video is transformed by a 3D graphics engine along with the 3D graphics. Various graphical frameworks can help with graphics/video blending. (See the section entitled "Middleware frameworks and application-specific tools" earlier in this paper.)

Latency

In some end-user applications the amount of latency in the system – or how long it will take the system to respond – is critical. An example of a low-latency application is video conferencing, where a delayed video image or audio channel would be very disruptive to conference participants. One way to limit latency is to minimize the number of buffers in the processing pipeline. In addition, every step in the pipeline should be critically examined and optimized to reach the latency requirements of the application. This may involve placing processing time constraints on each stage in the pipeline to ensure that the total elapsed time for the entire pipeline meets requirements.

AV sync

Synchronizing audio and video streams is required when the A/V content is decoded. For example, achieving a basic lip-sync is very common. More complicated synchronization is often required if the system's A/V decode processes must be synchronized with a source encoder, such as a broadcast video source. Synchronization with a source encoder is frequently necessary to ensure frames are not dropped or repeated, and to eliminate bit-stream buffer under runs and overflows.

Quality

Applications require varying degrees of multimedia quality. Achieving a high level of audio and video quality often involves software algorithms or accelerators, which typically consume processing resources. As a result, allocating sufficient processing resources for multimedia processing tasks must be taken into consideration by the system designers. Moreover, programmers have found that embedding instrumentation at strategic points in application code can help debug quality issues when problems arise.

Multimedia frameworks: OpenMAX and GStreamer

Both OpenMax (**khronos.org**) and GStreamer (**gstreamer.freedesktop.org**) are royalty-free, cross-platform multimedia frameworks which allow software developers to quickly plug media-processing modules into their systems. The capabilities of the multimedia modules that populate these frameworks include audio, video, still image and other types of multimedia processing routines. The routines are typically written in a high-level language such as C or C++ and feature a programming interface that provides an abstraction layer for the application programmer so that routines can be rapidly integrated into larger systems.

OpenMAX originated with the Khronos Group, a non-profit consortium of companies involved in multimedia processing, including Evans & Sutherland, ARM, TI, NVIDIA, SGI, Oracle/Sun and others. GStreamer is a traditional open-source project with contributions from its supporting community. The GStreamer project

originated at the Oregon Graduate Institute in 1999. The GStreamer framework has been adopted by a number of commercial companies, including TI, Nokia and others.

Embedded multiprocessing: Multi-processor, multi-core and multi-threaded programming

Generally speaking, multiprocessing (MP) refers to a system design approach that features multiple processors or multiple processing cores running multi-threaded software. The reasons for the contemporary prevalence of MP as a basic tenet of architecting computing and communications systems involve overcoming the limitations of single-processor architectures. Single-processor systems are limited to the speed of the processor, whereas MP systems can achieve higher system throughput by deploying multiple processors or processing engines to accomplish the same set of tasks. In addition to performance considerations, power consumption is often improved with an MP architecture in comparison to systems based on a single complex and large processor. This can be particularly critical in battery-operated mobile systems.

Given the significant advantages of MP, many embedded systems have migrated to this approach, but embedded applications typically have certain characteristics, such as the real-time nature of their processing, that may be not found in a majority of general-purpose systems. As a result, embedded systems place special requirements on the software environment and tools implemented to support an embedded multiprocessing system.

By its very nature, an MP architecture will require significant communications and coordination among the processing elements and the various threads that comprise the multi-threaded software of the system. On a hardware level, an interconnect methodology must be adopted and this will drive the system's communications model, which will feature either message-passing or shared-memory communications. Message-passing communications involves distributed processing where processing elements are configured with their own private memory space which is accessible by all processors. Data is then passed among processing elements in the form of packet messages which are placed in a processor's private memory. Shared-memory systems usually have a large amount of memory that is shared by all processors. Data is communicated among the processors by placing it in the shared memory space. This type of architecture raises issues concerning cache coherency, a memory consistency model and synchronization primitives.

From the perspective of embedded software development, an MP architecture will raise questions concerning which processing loads to place on which processors, how to coordinate the processing that is taking place in the system, how to divide multi-threaded software among the processing elements and many other such issues. Fortunately, there are tools and toolsets which can help.

OpenMP

OpenMP (Open Multiprocessing) is a de facto standard tools environment for shared memory parallel programming. OpenMP, which is supported by TI and has been ported to many TI processors, is a high-level programming construct that automates or significantly simplifies many of the issues involved in developing multi-threaded software for MP systems. With OpenMP, a user specifies the parallelization strategy for a

program at a high level by annotating the program code with compiler directives that specify how a region of code is executed by a team of threads. The compiler works out the detailed mapping of the processing to the processing elements. OpenMP can execute on embedded real-time OSs.

EZ tools

A significant number of embedded systems involve a heterogeneous MP architecture featuring one or more general-purpose processing elements and DSPs. DSP processing elements often accelerate certain computationally intense tasks like codec functionality for multimedia processing. TI's suite of EZ tools simplifies for embedded software developers the tasks that are needed to partition code among general-purpose processors and TI's C6000™ DSPs.

The EZ suite consists of three distinct tools: EZFlo, EZRun and EZAccel. Each tool addresses distinct developer needs. EZFlo targets DSP developers. It's intuitive, graphical UI gives developers the ability to drag, drop and connect functional blocks for rapid code generation and application prototyping. EZRun allows ARM/Linux developers to migrate C code into ARM-executable code or an ARM library which leverages the DSP for accelerated execution. EZAccel is intended for system developers who have implemented TI's codec engine framework. It accelerates adding DSP functionality along with codec operation on the DSP.

Conclusions

There are many factors for embedded software programming. No matter the application for which you're developing, there are a myriad of considerations. Our hope is that the topics and considerations discussed above have helped you in your embedded design. If you need more support of information, please visit our 24/7 support community at **www.ti.com/e2e**.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
	TI E2E Community Home Page		e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265