

Using dynamic loading to efficiently reconfigure DSP systems

A new market for mobile communications has raised a new set of DSP technical challenges. Use dynamic loading to effectively meet these challenges.

By Leland Szewerenko and Denise Ombres, Texas Instruments

MULTIFUNCTIONAL SYSTEMS BASED on DSPs (digital-signal processors) are becoming increasingly common, especially in wireless communications. The latest generation of ultra-high-performance DSPs is enabling base stations to carry more channels of voice, data, and video information, and advanced low-power DSPs are bringing Web browsing and other forms of multimedia to handheld systems. More than any previous DSP systems, these systems accommodate multifunctional software flexibility, not only because the communications standards are so diverse, but also because no one knows yet which of the many emerging applications will be successful.

Not surprisingly, the wide-open nature of the new market for mobile communications has raised a new set of DSP technical challenges. Among the most important of these is how to load and configure resource-intensive multimedia applications, as well as the many alternative communications algorithms, onto the widely varying types of systems that are coming into use. This issue is often complicated by the need to reconfigure software while the system is in operation. The success of many new applications and the DSP-based systems they operate on depend on developing software configuration techniques that meet this challenge.

Today, developers are discovering that dynamically loading application modules is the most effective way to reconfigure a system during operation to alter or expand its capabilities. Developers cannot statically configure many DSP systems before runtime, owing to real-time requirements that drive execution. However, dynamic loading permits a system to reconfigure itself on demand. For example, a base-station system that supports multiple modem protocols can use dynamic loading to appropriately reconfigure itself for the current service request. The download of Web-based content to a wireless handset offers another example. You cannot link a static program image that could support all of the possible Web navigation paths that a user might choose. But, with dynamic loading, you can download the support for a function when a user needs it. This runtime flexibility makes dynamic loading a key technology for reconfigurable-DSP systems.

Dynamic loading is familiar in the PC world, where users frequently employ it to assemble applications from independently upgradeable modules. However, developers in the past have not used dynamic loading much with DSP systems, which have traditionally been dedicated to single applications with real-time determinacy requirements.

Failure of alternative techniques

Developers have tried using static techniques for reconfiguring, but these techniques are less successful than dynamic loading. One such technique is building multiple program images, each with different combinations of algorithms. At runtime, when the

developer can determine the actual configuration, he or she can download the appropriate image. In a wireless link with limited bandwidth, it may take a long time to download a complete application, and if the developer has to reconfigure the system during operation by repetitively downloading full images, this time is likely to become excessive. In addition, when developers reconfigure the application, they interrupt system use. For developers, this approach requires full knowledge ahead of time to build all the images that an application may need.

In a system with multiple independent functional variations, the number of images developers must build grows combinatorially. As systems and software applications become more complex, it eventually becomes impossible to predetermine all possible requirements, so the scheme is no longer feasible.

As a partial solution to these problems, developers have introduced overlay techniques, in which alternative code or data modules reside in the same memory space. Reconfiguration takes place when a new module is written to that space at runtime, overlaying the previous module. This technique reduces loading time and does not necessarily interrupt the application for reconfiguration. If a developer can restrict the system to choosing only one module from a set of choices, then the system can fairly efficiently use overlays. But if the system requires multiple choices, it will probably become impossible to predetermine the memory assignment for each overlay. This circumstance may force the building of multiple overlay images, one for each possible memory binding. This approach re-creates the combinatorial number of image problem.

Advantages of dynamic loading

Dynamic loading avoids these problems by delaying binding the module to the system's physical memory until runtime. As a result, the code can run on different system setups, thus becoming more flexible and reusable. In most embedded systems, on-chip memory is at a premium, and the system must efficiently use it, but deciding what portion of a system will reside in on-chip memory can be a severe limitation in development. Dynamic loading allows developers to delay that decision until runtime when they can use real-time conditions to determine what algorithm will reside in memory at a given time. Developers can repeatedly replace or interchange algorithms if necessary with less impact on the ongoing use of the application. In the new wireless systems, multichannel applications can interchange codec algorithms on demand, and wireless personal communicators can download DSP content from remote servers.

Dynamic loading can also make it easier to upgrade a system. Usually, to upgrade, developers must overwrite the entire system, including valuable user and configuration data. But, with dynamic loading, they can limit an upgrade to one or more portions of a system, such as an algorithm or a data table. Further, a dynamic module upgrade depends only on the functional APIs (application-programming interface) that the base system provides; it does not depend on the static addresses of the base system. This fact means that one dynamic module can support multiple product releases, as long as the APIs that all releases provide are the same.

Table 1 summarizes some of the key concerns of DSP system reconfiguration. It shows a comparison of the properties of dynamic loading against overlays and static loading. As this **table** illustrates, dynamic loading has a powerful advantage over static images and overlays for the reconfiguration of DSP-based systems.

The following discussion of the components of dynamic loading and how it functions is based on the implementation in TI's TMS320 DSPs, though details may differ depending on the implementation. To view a demonstration of the dynamic loader demo, please see www.ti.com/dynamicloadingarticle.

A self-loading DSP application

In a typical self-loading DSP application, a single DSP is running a main control program that includes the dynamic loader library (**Figure 1**). This approach reserves some DSP memory for use by dynamic images. When an application requires a dynamic module or a mix of dynamic modules, the control program invokes the dynamic loader specifying the module images to be loaded. Module images may reside in flash or in secondary storage, or the system may read these images from some device. The loader allocates memory for each module, relocates the image for the chosen memory, fixes any references to the main program, and copies the modified image into the dynamic memory. When loading completes, the dynamic module becomes seamlessly integrated with the application and appears as if it existed in the system from initialization.

A typical use of self-loading is a DSP-enabled media player device that supports multiple media formats. It expresses each possible input source, media format, audio effect, and output process as a dynamic module. When a user selects a media, the system loads the appropriate modules, and playback begins.

MCU-controlled loading

In a typical MCU-controlled DSP application, a microcontroller manages one or more DSP processors (**Figure 2**). The microcontroller selects which DSP module or set of modules is to run on each DSP and uses the dynamic loader to load the appropriate modules. This usage differs from the self-loading application in several important ways. For one thing, the dynamic loader runs on the microcontroller, rather than on the processor being loaded. Also, the microcontroller on behalf of the DSP usually manages dynamic memory, because the DSP requires an indirect-memory-allocation algorithm. In addition, in microcomputer-controlled loading, loading of the dynamic image may require output through an interface peripheral because the microcontroller may not be able to directly address the DSP memory.

Master-slave applications occur in base stations, central offices, and wireless terminals. In base stations and central offices, a single microcontroller manages multiple DSPs, each of which handles multiple channels. TI's OMAP1610 platform typifies a wireless terminal. In the device, targeting handheld-multimedia applications, the system master is an ARM MCU that includes a DSP-bridge function to control a TMS320C55x's loading, initiation, and execution of code.

A field-testing application

In a typical field-test application, the service technician has a set of test applications expressed as dynamic-load modules. Typically, a vendor over time increases and enhances this test. Spare memory for test modules' use fielded the serviced product and is enabled for dynamic loading. The test equipment comprises the test modules and the dynamic loader and test-control application. (The loader in this case may be in the test-control application or built into the fielded product.) The test equipment also includes

definitions for the entry points built into the product. (These symbol definitions may be built into the product, or the test set may carry them as a symbol module.) Diagnostic testing proceeds by loading individual tests. Each test is dynamically connected into the services of the product software it requires, such as device drivers and state variables.

In a field-test usage, the major advantage of dynamic loading over an overlay scheme is that developers need not build the tests themselves to match the built version of the system under test. An overlay scheme would require a library of test images equal to the number of tests times the number of product versions, whereas the dynamically loaded scheme requires the number of tests plus the symbols for each product version.

The advantages of linear versus quadratic complexity of the test set are profound. If the product has built-in symbol information, then it can eliminate the last term. This reduction of configuration complexity is the primary advantage of dynamic loading in the field-test usage.

Dynamic-loader functions and requirements

The dynamic loader is a configurable library or API that the user application calls, using C for easy interfacing. In addition to writing the program image into DSP memory, the dynamic loader performs the necessary linking, so that the module, the main application, and other loaded modules can interoperate. To do so, the dynamic loader resolves references to symbols that are external to the module, and it also records any global definitions within the module for external use. When the loader unloads a module, the dynamic loader performs memory recovery so that the system appears as if the dynamic module never existed.

The dynamic loader requires a set of four support classes. The application passes objects that implement these classes and configure the dynamic loader as the first four parameters in a load request. The first class, the image source, permits the system integrator to define a mechanism for streaming the input. The source can be external memory, an external device or peripheral, or even an embedded structure within the application. The second class, the symbol handler, includes functions that map symbols to addresses, as well as memory management and error reporting that relates to symbols. The dynamic loader uses the third class, the DSP-memory allocator, to request memory for the dynamic module. Once the dynamic loader determines the contents of the memory, the dynamic loader uses the fourth class, the DSP-memory initializer, to request the system to update the memory address that the DSP-memory allocator returns.

Because the dynamic loader library links to a user's application, the code that implements it must be fast, small, and robust. Programmers coded the C library with algorithms that they optimized for speed. The host's image reformatter, another component of dynamic reloading, performs any calculations that it can perform offline. To efficiently use the DSP's memory, the dynamic loader has a minimal footprint. Finally, the vendor has exhaustively tested the dynamic loader, because runtime errors in this code are unacceptable.

Developing dynamic modules

Code development for dynamic modules is almost identical to standard development. **Figure 3** shows a representative development flow for the creation and debugging of dynamic modules. The three dynamic loading components—the dynamic loader, the

image reformatter and the debugger plug-in—are shaded. Initially, the main application links in the dynamic-loader library and sets aside memory for the dynamic modules. Then, it builds executables of the modules in the normal way, except that image addressing is relocatable, not static, through the use of symbols that the dynamic loader manages during execution.

This system feeds the relocatable image as input to the image reformatter, which optimizes it for downloading. The reformatter removes portions of the object code that are not necessary during the loading to reduce the image size. In addition, the reformatter reorders the image for I/O-streamed loading, formats the image for inclusion in the application, adds checksums for validating the input, and provides an interface for controlling the module's exported symbols.

When the system needs a module, the system streams it to the dynamic loader, which processes it and writes it into the reserved dynamic-module memory. Because the debugger must also be able to link to the module, a debugger plug-in detects the existence of the dynamic module and then locates the original object file that corresponds to the dynamic module. The plug-in then updates the development environment with all the information necessary to enable full debugging.

Importance of limiting symbol exports

The image-reformatting tool provides an interface for controlling the symbols a module exports. This feature gives a developer a way of hiding some symbols while making others visible for other modules' reference. The system uses this capability for both controlling access and minimizing load time and memory footprint. Each visible symbol requires several bytes of memory on the processor running the dynamic loader. If all link-time symbols were visible, this memory usage could become an issue in small embedded systems. When the reformatter limits symbol exports, the symbol memory usage is generally not an issue.

Another benefit of limiting symbol exports is faster loading. A performance analysis based on OMAP1610 platform testing produces a statistical model of the following load time: $4250 + 373 * \text{Symbols} + 273 * \text{Relocations} + 6.5 * \text{Image_Bytes}$.

This analysis indicates that the number of exported symbols in a module is by far the single most important factor under the control of the developer for improving downloading speeds. The overall module size is of minor importance by comparison, having less than 2% of the effect of reducing the number of symbols. Although the exact weighting of these factors and others that are not developer-controlled vary with DSPs and system configurations, the overriding rule to minimize external symbols remains the same.

Opening possibilities

Dynamic loading promises to open new application capabilities for DSP systems—a development that the coming of next-generation wireless technology truly needs. High-performance DSPs can load new communications algorithms, and DSP-based handheld systems can download Web content and multimedia application modules on an as-needed basis without halting the system. More flexible and efficient than multiple static program images or overlays, dynamic loading provides a means for the seamless modular reconfiguration of DSP systems. Although dynamic loading is still a novelty in the DSP

world, it will quickly become an enabling factor for many new, multifunctional DSP applications. Soon, as with so many other DSP innovations, developers will be asking how they ever got along without it.□

Authors' biographies

Leland Szewerenco is a senior member of the technical staff at Texas Instruments, where he is responsible for the OMAP and multiprocessor-tool strategy, RTDX architecture, and XDS 560 software architecture and implementation. He received a bachelor's of science degree in physics and math from Knox College (Galesburg, IL) and a master's degree in computer science from Carnegie-Mellon University (Pittsburgh).

Denise Ombres is a systems software developer for Texas Instruments. As senior developer of software for embedded systems, she provides compiler and runtime support for TI DSPs and OMAP platforms. Ombres received her bachelor's and master's degrees in computer science from the University of Pittsburgh.

Table 1—Key issues of DSP-system reconfiguration

	Static images	Overlays	Dynamic loading
Load Time	Size of system	Size of overlay	Complexity of overlay
Availability	Down during reload	Up	Up
Multifunction complexity	Combinatorial by function	Combinatorial by memory binding	Linear
Code size	System*number of images	Base+overlays* bindings	Base+overlays +loader
Upgrade dependencies	Entire system image	Base-image addresses	Base-image APIs

Figure 1. Dynamic Self-Loading by the DSP

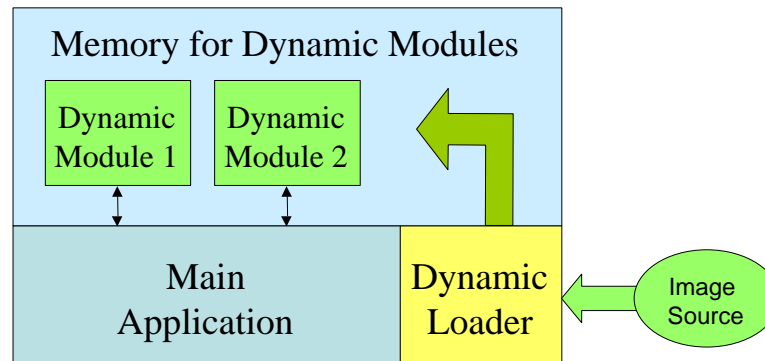


Figure 2. MCU-Controlled Dynamic Loading

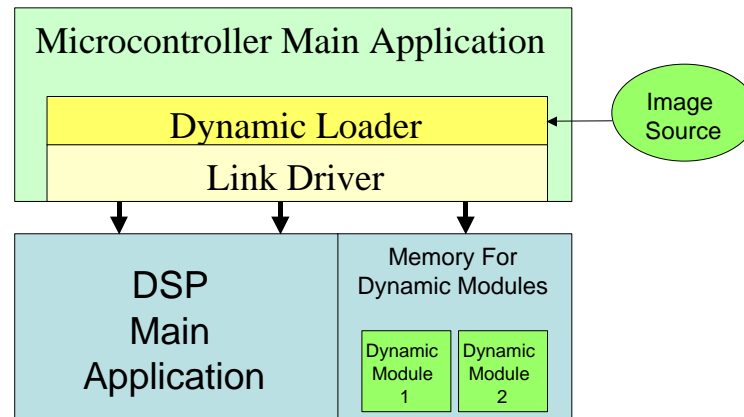


Figure 3. Application Development Flow for Dynamic Loading

