Beyond Platformization: Using Mobile Software Management to Achieve Feature Customization of Mobile Phones

By John Purcell, Red Bend Software

"Platformization" of mobile phone software architectures promises to greatly speed the process of designing and decrease the cost of developing wireless devices. By leveraging a common software base, wireless designers and developers are able to more cost effectively build devices that meet manufacturer and operator

[1]

requirements for different market segments. In this approach, every phone in a market segment has the same software assets when it ships. However, consumer demand for personalization of mobile phones is taking software customization requirements further. Consumers currently personalize their phones with ringtones, games and wallpapers. As consumers grow more sophisticated in their use of mobile phones, they will want to perform greater personalization by selecting the software features, applications and services that fit their unique lifestyles. Potentially, no two mobile phones will be alike.

Selling new software to consumers post-sale generates new revenue opportunities across the mobile value chain, but it also creates a new set of challenges for the manufacturers of mobile devices and platforms. It is time to rethink the best way to bring to market devices that remain cost-effective but that meet the new software customization requirements that the market is increasingly demanding.

To achieve feature customization of mobile phones, wireless designers and developers must look beyond simply storing programs in a User File System, and instead focus on the embedded software layer &#151 those software elements residing in Read-Only Memory (ROM). To do this, manufacturers will need to reexamine current methods used to build embedded software in order to enable pre- and post-sale software customization down to the individual component level. Enhanced mobile software management capabilities are being developed that promise to help manufacturers maximize the true potential of platformization and meet the growing demand for feature customization of mobile phones beyond

applications in read/write user file systems.

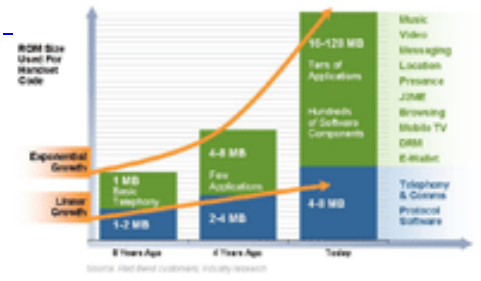
Architectural Suitability

There are two primary methods for structuring mobile software in the Read-Only flash memory space: store software in a modular fashion over an “open” operating system such as Symbian or Linux, or store it in a monolithic, statically-linked image, typically over a proprietary RTOS. Each has its advantages in terms of physical and logical architecture, but they pose very different challenges to remote management tasks such as installation, replacement and removal of embedded software.

There are inherent advantages in execution speed and platform security to mobile device software comprised of statically-linked binary objects that form a monolithic image in flash memory. In allowing software code to be executed in-place in NOR flash, these advantages are enhanced further by reducing RAM requirements. However, once the code is compiled, linked and burned to flash, efficient modular management of the embedded software image becomes difficult. The slightest modification of any of its constituent components will affect almost the entire image. Changes to the static references alone make the modular management of individual embedded components impossible. In other words, once the large binary image is created, it must be managed and manipulated as a single component and can not be managed as a collection of individual software elements.

On the other hand, storing embedded software components in a more modular fashion requires an on-device loader mechanism to execute the code. For example, to execute a software element or collection of elements associated with an application, the elements must first be loaded to RAM and have their dependencies and references dynamically resolved before the functionality can be made available to the consumer. This apparently makes the software more suitable for remote management. However, in reality there are inefficiencies associated with modular storage and dynamic linking of software that often offset the advantages inherent in the software modularity. Inevitably, there is a load-time penalty associated with the need to load the software components into RAM, process all their symbolic information and modify their actual content in order to execute them. The resultant delay becomes somewhat of a nuisance especially in today's complex devices.

To solve this side effect of dynamic linking, architects of modular mobile devices often turn to the practice of "prelinking." By resolving all dynamic information at build-time and storing the components in ROM exactly as the on-device loader requires, load efficiencies result at execution time. But "prelinking" the code before it is burned into flash also prevents the software



[2]

elements from easily being replaced in a modular manner, as any changes in component size and location may affect prelinked references in other modules. The result turns the stored firmware in ROM back again to the 'good old' monolithic form. To add to the difficulty, the file system in which most device software files are commonly stored is of a Read-Only type like ROFS (Symbian), cramFS or squashFS (both Linux) as well as other RTOS based ones. These read-only file systems are designed for efficient retrieval and storage, not for updating, deleting or adding individual files. In addition, these file systems allow storing their content in compressed form, which adds yet another complexity layer for on-device updates.

Both methods of software structuring (static and dynamic linking) mean that, once the device has shipped, changing anything in the embedded software has traditionally only been possible using Firmware Over-the-Air (FOTA) updating. This technology uses a known target software version, already fully built and prepared for burning to flash, and computes the differences that exist between it and the source software version that should exist on the device. There are several approaches to FOTA. The computational approach is considered the most practical and is powerful enough to compute the new and potentially shifted references in the device and therefore construct the new software version over the old version. However, FOTA technology updates the entire firmware image as one single component (monolith) and does not support software customization by individually managing software components. A true solution for embedded software management is required that can be scaled across multiple device models, operator variants, components and component versions.

Enabling the Vision

In general terms, managing embedded software involves the ability to install, upgrade, replace and potentially remove software components from the Read-Only memory space. As discussed, these management tasks