Linux For Space Constrained Small Embedded Devices

A considerable segment of embedded systems are often found in mass-market products and are therefore subjected to hard economic constraints. The basic nature of these systems mandates further constraints on physical size and power consumption. These in turn give rise to resource constraints on the computing platform level, e.g., constraints on computing speed, memory size, and communication bandwidth etc. In spite of the rapid development of computer hardware these constraints are true due to the economic overheads. In most cases it is not economically justified to use a processor with more capacity due to the overall product's cost limits.

Now coming to the software that goes into these space crunched hardware?

A recent survey conducted on small embedded device manufacturers revealed the following facts.

Most of these vendors avoided using embedded operating systems and the response was a straightforward, "we don't need one." More than half of these vendors mentioned a simple lack of need as the sole reason for skipping the operating-system. It's easy to think of many products that might not need an OS so this response isn't surprising. Out of those about 30% said an OS will strain their system's processor and/or memory. Some said an OS was too expensive and the remaining felt that operating systems are too complicated. However the need for an embedded OS was felt considering the fast changing demands of the consumers, who are looking for multi-purpose small embedded devices, which mandates the use of an OS

Now the question, what is the most economical and easy OS solution?

This brings up Linux as a viable option. Let's explore the reasons behind this choice and most importantly understand how it can be used on these space constrained devices?

•Linux is open source kernel backed by a strong community of developers; members of the community provide bug fixes, enhancements, and even support old devices long after the manufacture has ceased support.

•Linux offers a large library of functionality that can be leveraged to accelerate development. Linux is easy to customize and port even on to MMU less processors. With Linux community's commitment to support embedded systems and various embedded hardware vendors contributing development tools and support.Linux is the best bet to migrate from hardware specific code to richer software.

Linux kernel is massive collection of sources targeting various architectures and platforms. Linux kernel currently contains around 9.2 million lines of code with a 10% increase every year. With the current version, pattern new stable kernel is available every 3 to 4 months. Linux Currently supports 25 different architectures, 60 Filesystems, 20 net protocols(L2), 50+ ATA Drivers, 300+ pci Drivers, 200+ Ethernet Drivers

"Linux supports more individual types of devices overall than any other kernel" -Greg KH

Default vanilla kernel configuration includes full network support, support for wide variety of input devices, and does not offer fixed boot time (can afford the time delays related to device discovery).

So with all these code buried within Linux, how can we fit it into these small devices?

since Linux kernel sources are available, it allows embedded engineers to customize kernel build by integrating various optimizations that help us keep check on kernel foot-print and reduce boot up-time.

Following is a list of few tweaks , that we could consider.

1 Disabling unrequired Device drivers and file systems like Ext2, Ext3 (not required on mass storage less embedded devices)

2 Disabling unrequired network protocols and other network services like Net filters, packet forwarding etc…

3 Disable SMP Support when not running on dual CPU boards. This reduces the kernel image size and also results in better performance on uniprocessor machines

4 Disabling support for other CPU variants for the architecture we have selected would help us reduce kernel image size by stripping out unrequired start up code.

5 Disable support for Dynamic configuration of kernel parameters through sysctl's.

6 Reduce the kernel log buffer to 4k (default 16k).

7 Support for hotplug events / user space notifications are often not required in Embedded systems kernel so choosing to disable them would help reduce kernel image size

8 Enable CONFIG_EMBEDDED which enables a group option to further trim down the kernel. Notable options enabled are the ability to select a different allocator and a selection to optimize for size (CONFIG_CC_OPTIMIZE_FOR_SIZE).

9 **Enable SLOB Allocator**
Current kernels are by default set to SLAB allocator which is more efficient on the large memory desktop machines. So enable SLOB allocator which perfectly suits smaller, resource limited devices.

10 **Disable Module Support**
Modules under Linux allow kernel code to be dynamically installed or removed. However, to accomplish this, each module is loaded onto a fresh page. This has two consequences. On architectures that support large memory pages, modules do not share the large page typically used by the primary kernel code. Consequences of this is another page table entry is needed and page table caches may get thrashed more which may be a potential performance impact. A second consequence is if a module is the page size plus one byte, the total memory consumed is 2 entire pages. This is almost 50% waste. Or in another case if the module is small, a full page has to be allocated. If the page size is 4K and the module is only 1K, this is a 300% waste. If many such modules are loaded, there can be considerable waste of memory.

11 Consider UClinux when there are no stringent MMU requirements

12 Other issues apart from normal selections include appropriate tagging of disposable code. Custom drivers or even existing drivers should have the initialization code tagged appropriately so memory occupied by them can be freed. This can be done by the use of the GCC __init attribute.

13 Developers also need to be aware of performance constraints with XIP(Execute in place) NOR flashes on some custom hardware. XIP allows the kernel to run form the same place that it is stored. This requires the flash to appear as linear memory. On custom hardware (such as certain ARM based systems) the bus for interfacing to NOR flash may be smaller than the native size of the processors for ex: 16bit NOR memory interface requiring two access cycles to load a native size piece of data, 32bits wide. This would result in massive performance bottleneck. In addition, XIP prevents the use of compressed kernel images.