**What is a Beagleboard?**

This is the brainchild of a small group of volunteers (many of whom are Texas Instruments employees – TI’s chips are used in the design) who were frustrated at the lack of affordable development boards, and with the kinds of concerns faced by those seeking to get involved with embedded development. Their attempt to remove these barriers led to a small, (relatively) low-cost, durable (hard to ‘brick’) ARM-based development platform that couples an open source hardware design that anyone can look at with open source software. The project is supported by a strong community of users and developers who support each other and welcome newcomers. That it also benefits TI to have developers using its parts is obvious, but that in itself isn’t necessary a bad thing as such a project does need some kind of backing.

**Porting the kernel**  
Linux has been available for the ARM architecture for many years now. The original ‘port’ was done by Russell King, and he is still the maintainer through whom all ARM kernel patches generally must pass. The work was much harder then than it would be now, since at the time the Linux kernel was still very Intel-centric. In fact, modern Linux kernels come with a handy reference example called asm-generic that shows all of the header files and kernel interfaces that a new architecture port should provide. Once the kernel has been ported to a given architecture, it is necessary to implement support for a specific platform based upon that architecture.

To understand platforms, consider that the PC is a platform originally from IBM and built upon the Intel x86 processor, while the Intel Moorestown is an x86 processor but it is used to create non-PC platforms. The difference is that ‘PC’ implies the presence of certain standard devices, ACPI and UEFI system descriptor tables, a PCIe system bus and so forth. Similarly, Texas Instruments implements the ARM architecture in its OMAP processors, while BeagleBoard is a particular variant of the OMAP3 platform. Thus Beagle uses generic ARM architecture code in the arch/arm directory, but it also makes use of platform code in arch/arm/mach-omap2/board-omap3beagle.c.

No matter the architecture, kernel code execution generally begins (after being loaded by the bootloader – more on that later) in a file written in assembly language, and known as head.S (for historical reasons). In the case of ARM, this file is arch/arm/kernel/head.S, at a location identified very plainly with the comment, ‘Kernel startup entry point’. When this code begins running, the processor has few of the required features enabled, such as virtual memory or caching of instructions and data. It may even be executing code that has been loaded at an address that differs from the one at which the code was compiled to run at. Thus, to avoid a silent crash, the first thing the kernel does is to work out what memory location it was really loaded into. It then carefully avoids making any regular function calls until the virtual memory logic has been enabled, at which point the kernel is running at the correct (virtual) location.

The first useful thing the kernel does is to (carefully, using special memory-location-accounting logic) call the functions \_\_lookup\_processor\_type and \_\_lookup\_machine\_type, which are used to perform various ‘fixups’ – literally hacks to replace kernel code or enable and disable certain features. After this, the kernel initialises a region of memory for use as a stack from C code and jumps into the first real C function, the top-level generic ‘start\_kernel’, in init/main.c.

This function does a lot of the same things on every platform, one of which is to call into a function calledsetup\_arch, which calls setup\_machine. Later, it calls a function do\_basic\_setup, which causescustomize\_machine to run by virtue of a feature known as an ‘initcall’ (the customize\_machine function is marked specially). This causes init\_machine to run, which in the case of BeagleBoard is the functionomap3\_beagle\_init, contained within the board-omap3beagle.c file mentioned previously. The setup here includes initialising various Beagle devices, but for the most part simply informs the kernel that devices exist for which standard drivers are loaded later.

You can dig deeper into the startup process by following the code using a tool such as Cscope, or [LXR](http://lxr.linux.no/). You can also purchase a special hardware debugger and monitor the startup of the BeagleBoard. This author uses an inexpensive ‘Flyswatter’ JTAG debugger from Tincantools and the open source OpenOCD utility, but this is still under development and only recommended for the brave. Much more expensive, commercial hardware debuggers are available if you have a hardware budget to play with.