Lowering the Cost of Linux

You can automate the process of porting an OS using the MLD format.

by Derek Palmer
Embedded Marketing Specialist
Xilinx, Inc.
derek.palmer@xilinx.com

The growth of Linux-based embedded systems has raised many questions about the real cost of a free open-source operating system (OS) like Linux. The primary cost is labor. Porting a Linux kernel to a custom embedded processing system requires a thorough understanding of what kernel features your product requires, what dependencies these features have on other kernel services, and the skills of your engineering team. Even more important is understanding what new technologies are available to help automate these tasks and save money.

Open-Source Versus Commercial OSs

Many of us have read the GNU General Public License (GPL) that governs how to use Linux sources. Understanding which parts of Linux are truly free and unencumbered by legal restrictions and which are carefully protected is usually the first step in determining what the cost will be to use Linux in your system. This is similar to evaluating commercial OS or RTOS vendors to determine what features and fees are included with their products.

Clearly, one decision is whether to buy a commercial Linux distribution or tackle the porting of a free Linux distribution. Many engineers have turned to Linux to utilize only a small fraction of the available services, such as file systems, TCP/IP, and web services like HTTP or THTTP. If you only need a few services, porting your own Linux kernel may not be that time consuming or costly.

Commercial OSs have ported their products to many of the most popular processors and offer board support packages (BSPs) to run on a variety of standard hardware platforms. This is where a commercial OS has an advantage. Unfortunately, if you are designing a custom embedded processing system, a commercial OS may incur additional fees for services to create a custom BSP for your embedded hardware system – unless you can automate the BSP generation. This is one of the many features of the award-winning Xilinx® Platform Studio (XPS) tool suite: BSP and library generation are automated for Xilinx embedded processing designs.

To better understand this process, let's take a look at the various layers of software and hardware needed to build a complete OS. Figure 1 is a diagram representing the many layers of a Linux system. At the lowest level is the microprocessor and its peripherals; this represents the hardware layer. To communicate with these peripherals, you need Linux kernel-compatible software drivers. The kernel also includes very important services like boot functions and interrupt handling. These functions are highly dependent on your system architecture and are usually customized for specific processor and system architectures.

When the underlying hardware platform changes – often to accommodate new application features – the kernel must also change, which takes time and raises design costs. This is common in highly customized processing platforms tailored to a particular kind of application. Designers often use FPGA-based embedded systems so that you can tune system architectures to find the optimal balance of hardware performance and software flexibility. Additionally, if you are designing your own custom embedded platform, you must also consider the time involved in writing custom drivers for new peripherals, determining the proper settings for your kernel configuration file and the verification time to fully test the new kernel.

Automating OS Setup

XPS allows embedded system designers to build and optimize systems and automate the tasks of assigning address maps, configuring drivers for peripherals, and linking all of the necessary libraries to make your system ready for writing your first application. But what happens when your application is an entire OS? This is where Xilinx offers an innovative way to help automate the generation of customized OSs like Linux.

XPS provides a way to export information about the microprocessor, peripherals, and system architecture through the

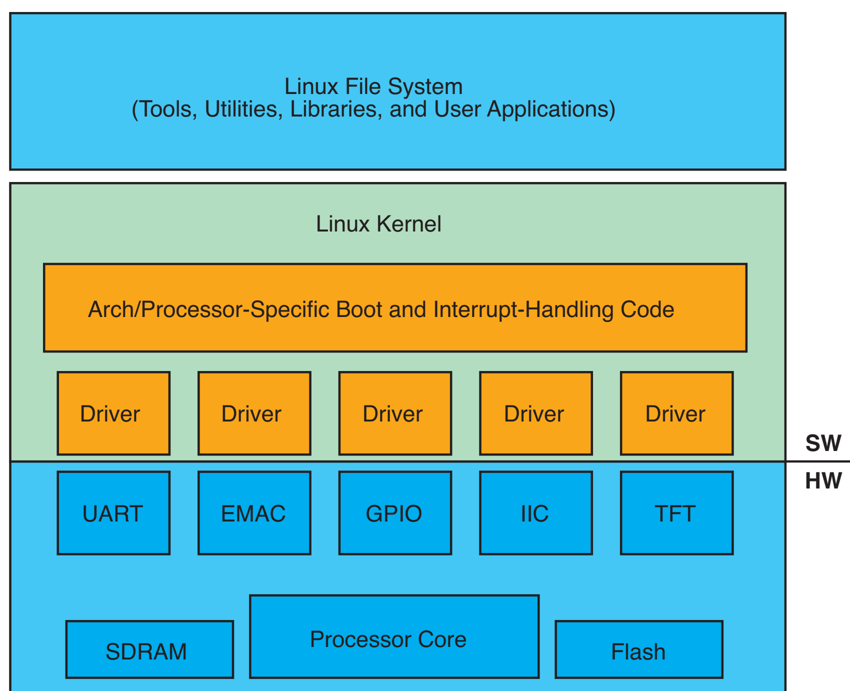


Figure 1 – Linux hardware and software layers

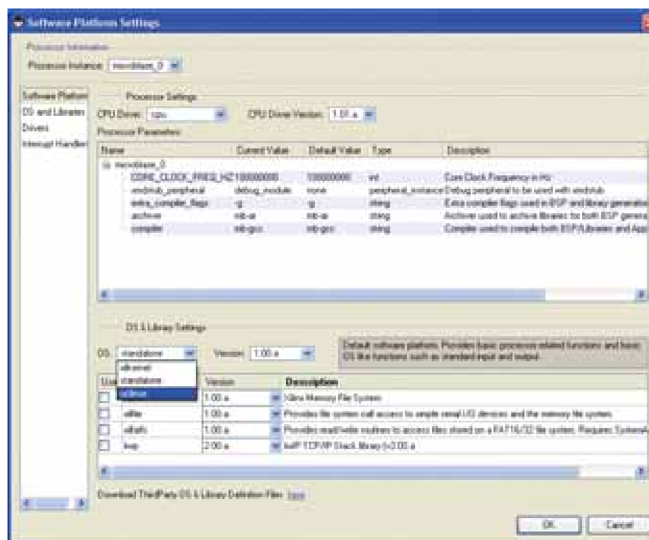


Figure 2 – Setting up μClinux in the XPS software platform settings window

microprocessor library description (MLD) file. The MLD file contains directives for customizing software libraries and generating BSPs for OSs.

Each OS/library has an MLD file and a Tcl (tool command language) file associated with it. The Tcl file uses the MLD file to customize the OS/library depending on the embedded design system architecture. These files already exist for many Xilinx OS partners. For example, for µClinux, you can simply select it from a menu, as shown in Figure 2. Changes to your embedded system hardware will automatically update the libraries and regenerate a kernel configuration file that will control how the kernel is built.

XPS also generates an "auto-conf.in" file that the auto configuration process uses to build the makefile for the µClinux kernel. You can automatically rebuild the kernel without having to redesign your kernel configuration files, a valuable time-saving feature when you consider that there are about 380 line items in the "auto-config.in" file. Manually updating each of these files could be error-prone and time consuming. A small portion of this file is shown in Figure 3.

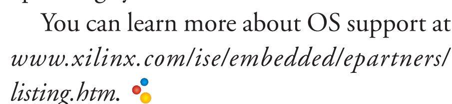
The MLD technology is generic and can be applied to other OSs. The Integrity operating system from Green Hills, for example, also uses this technology.

Conclusion

As demands for higher performance embedded systems increase, we will continue to see new embedded processing system architectures that contain specialized hardware customized for specific markets. Developing these systems requires an iterative process that combines compiling and profiling software applications as well as hardware co-processing units and specialized peripherals.

Manually updating the Linux or other OS kernel every time you iterate your embedded processing hardware platform wastes engineering time. Automating these tasks significantly reduces the cost of developing embedded systems that require an operating system.

You can learn more about OS support at *www.xilinx.com/ise/embedded/epartners/listing.htm*.



```
##############################################################
# CAUTION: This file is automatically generated by libgen.
# Version: Xilinx EDK 8.2.01 EDK_Im_Sp1.3
# Description: µClinux Configuration File
##############################################################

# MAIN_MEMORY Settings
define_hex CONFIG_XILINX_ERAM_START 0x22000000
define_hex CONFIG_XILINX_ERAM_SIZE 0x02000000

# FLASH_MEMORY Settings
define_hex CONFIG_XILINX_FLASH_START 0x21000000
define_hex CONFIG_XILINX_FLASH_SIZE 0x01000000

# LMB_MEMORY Settings
define_hex CONFIG_XILINX_LMB_START 0x00000000
define_hex CONFIG_XILINX_LMB_SIZE 0x00002000

# System Clock Frequency
define_int CONFIG_XILINX_CPU_CLOCK_FREQ 100000000

# Definitions for MICROBLAZE0
define_string CONFIG_XILINX_MICROBLAZE0_INSTANCE microblaze_0
define_string CONFIG_XILINX_MICROBLAZE0_FAMILY spartan3e
define_string CONFIG_XILINX_MICROBLAZE0_INSTANCE microblaze_0
define_int CONFIG_XILINX_MICROBLAZE0_D_OPB 1
define_int CONFIG_XILINX_MICROBLAZE0_D_LMB 1
define_int CONFIG_XILINX_MICROBLAZE0_I_OPB 1
define_int CONFIG_XILINX_MICROBLAZE0_I_LMB 1
define_int CONFIG_XILINX_MICROBLAZE0_USE_BARREL 1
define_int CONFIG_XILINX_MICROBLAZE0_USE_DIV 1
define_int CONFIG_XILINX_MICROBLAZE0_USE_HW_MUL 1
define_int CONFIG_XILINX_MICROBLAZE0_USE_FPU 0
define_int CONFIG_XILINX_MICROBLAZE0_USE_MSR_INSTR 0
define_int CONFIG_XILINX_MICROBLAZE0_USE_PCMP_INSTR 1
define_int CONFIG_XILINX_MICROBLAZE0_UNALIGNED_EXCEPTIONS 0
define_int CONFIG_XILINX_MICROBLAZE0_ILL_OPCODE_EXCEPTION 0
define_int CONFIG_XILINX_MICROBLAZE0_IOPB_BUS_EXCEPTION 0
define_int CONFIG_XILINX_MICROBLAZE0_DOPB_BUS_EXCEPTION 0
define_int CONFIG_XILINX_MICROBLAZE0_DIV_ZERO_EXCEPTION 0
define_int CONFIG_XILINX_MICROBLAZE0_FPU_EXCEPTION 0
define_int CONFIG_XILINX_MICROBLAZE0_DEBUG_ENABLED 1
define_int CONFIG_XILINX_MICROBLAZE0_NUMBER_OF_PBRK 2
define_int CONFIG_XILINX_MICROBLAZE0_NUMBER_OF_RD_ADDR_BRK 0
define_int CONFIG_XILINX_MICROBLAZE0_NUMBER_OF_WR_ADDR_BRK 0
define_int CONFIG_XILINX_MICROBLAZE0_INTERRUPT_IS_EDGE 0
define_int CONFIG_XILINX_MICROBLAZE0_EDGE_IS_POSITIVE 1
define_int CONFIG_XILINX_MICROBLAZE0_FSL_LINKS 0
define_int CONFIG_XILINX_MICROBLAZE0_FSL_DATA_SIZE 32
define_hex CONFIG_XILINX_MICROBLAZE0_ICACHE_BASEADDR 0x22000000
define_hex CONFIG_XILINX_MICROBLAZE0_ICACHE_HIGHADDR 0x23FFFFFF
define_int CONFIG_XILINX_MICROBLAZE0_USE_ICACHE 1
define_int CONFIG_XILINX_MICROBLAZE0_ALLOW_ICACHE_WR 1
define_int CONFIG_XILINX_MICROBLAZE0_ADDR_TAG_BITS 14
define_int CONFIG_XILINX_MICROBLAZE0_CACHE_BYTE_SIZE 2048
define_int CONFIG_XILINX_MICROBLAZE0_ICACHE_USE_FSL 1
define_hex CONFIG_XILINX_MICROBLAZE0_DCACHE_BASEADDR 0x22000000
define_hex CONFIG_XILINX_MICROBLAZE0_DCACHE_HIGHADDR 0x23FFFFFF
define_int CONFIG_XILINX_MICROBLAZE0_USE_DCACHE 1
define_int CONFIG_XILINX_MICROBLAZE0_ALLOW_DCACHE_WR 1
define_int CONFIG_XILINX_MICROBLAZE0_DCACHE_ADDR_TAG 12
define_int CONFIG_XILINX_MICROBLAZE0_DCACHE_BYTE_SIZE 8192
define_int CONFIG_XILINX_MICROBLAZE0_DCACHE_USE_FSL 1
define_string CONFIG_XILINX_MICROBLAZE0_INSTANCE microblaze_0
define_string CONFIG_XILINX_MICROBLAZE0_HW_VER 4.00.aFig. 3
```

Figure 3 – Example portion of the µClinux "auto-config.in" file created by XPS