Linux Port to LatticeMico32 System Reference Guide

Lattice Semiconductor Corporation
5555 NE Moore Court
Hillsboro, OR 97124
(503) 268-8000

February 2008

Copyright

Trademarks

Disclaimers

to correct any errors contained herein or to advise any user of this document of any correction if such be made. LSC recommends its customers obtain the latest version of the relevant information to establish, before ordering, that the information being relied upon is current.

Type Conventions Used in This Document

Convention	Meaning or Use
Bold	Items in the user interface that you select or click. Text that you type into the user interface.
<*Italic*>	Variables in commands, code syntax, and path names.
Ctrl+L	Press the two keys at the same time.
`Courier`	Code examples. Messages, reports, and prompts from the software.
`...`	Omitted material in a line of code.
`.` `.` `.`	Omitted lines in code and report examples.
`[ ]`	Optional items in syntax descriptions. In bus specifications, the brackets are required.
`( )`	Grouped items in syntax descriptions.
`{ }`	Repeatable items in syntax descriptions.
`|`	A choice between items in syntax descriptions.

Contents

1

Introduction

This guide explains how to configure and build the modules required to run LatticeMico32 System on the Linux operating system.

LatticeMico32 System

LatticeMico32 System is a highly configurable 32-bit Harvard architecture soft microprocessor core for Lattice Semiconductor FPGA devices. It provides 32-bit-wide instructions with 32 general-purpose registers and directly interfaces to WISHBONE-compatible peripheral components. No memory management unit (MMU) is available for the platform.

LatticeMico32 System provides a highly configurable platform, which poses some challenges for an operating system port. Operating systems are usually parameterized to support a basic architecture, specific CPU variants, and a specific board. However, LatticeMico32 System breaks most of the assumptions implicit in this structure. Since most of the CPU's basic functions can be parameterized, the configuration options for a typical operating system and the included peripheral drivers can be diverse. In addition to these challenges, the individual peripherals cannot be automatically detected, and access to peripherals that have not been configured can cause failures. The configurability and flexibility of the LatticeMico32 System platform creates complexity that must be carefully managed in the boot loader and the board support package to ensure that a system's integrator will not be slowed down by configuration mismatches and that pre-tested binary software components can be reused among multiple systems.

The Linux nommu port to LatticeMico32 System has been designed to reduce the number of configuration points in the Linux kernel that are specific to each LatticeMico32 System configuration and therefore reduce the number of potential error sources. The configuration-specific details are mainly located in the boot loader and passed to a "universal" kernel.

The result of this design is a very clean and efficient tool flow when you use the Linux port to LatticeMico32 System, accelerating the time to market for Linux LatticeMico32-based solutions. The Linux port to LatticeMico32 System assumes only a small set of prerequisites from the platform, such as a divider, and leaves almost everything else to be configured during system boot through parameters passed by the boot loader. You can therefore run a single Linux kernel configuration across a wide variety of configurations.

While simplifying the overall integration process, the port remains true to the Linux paradigm of providing a large number of configuration options. It also allows the application-specific drivers to be easily integrated, which is a task simplified by the small number of LatticeMico32 System-specific configuration options.

Design Flow

The Linux port to LatticeMico32 System consists of multiple modules that are configured and built by using the following specialized tools:

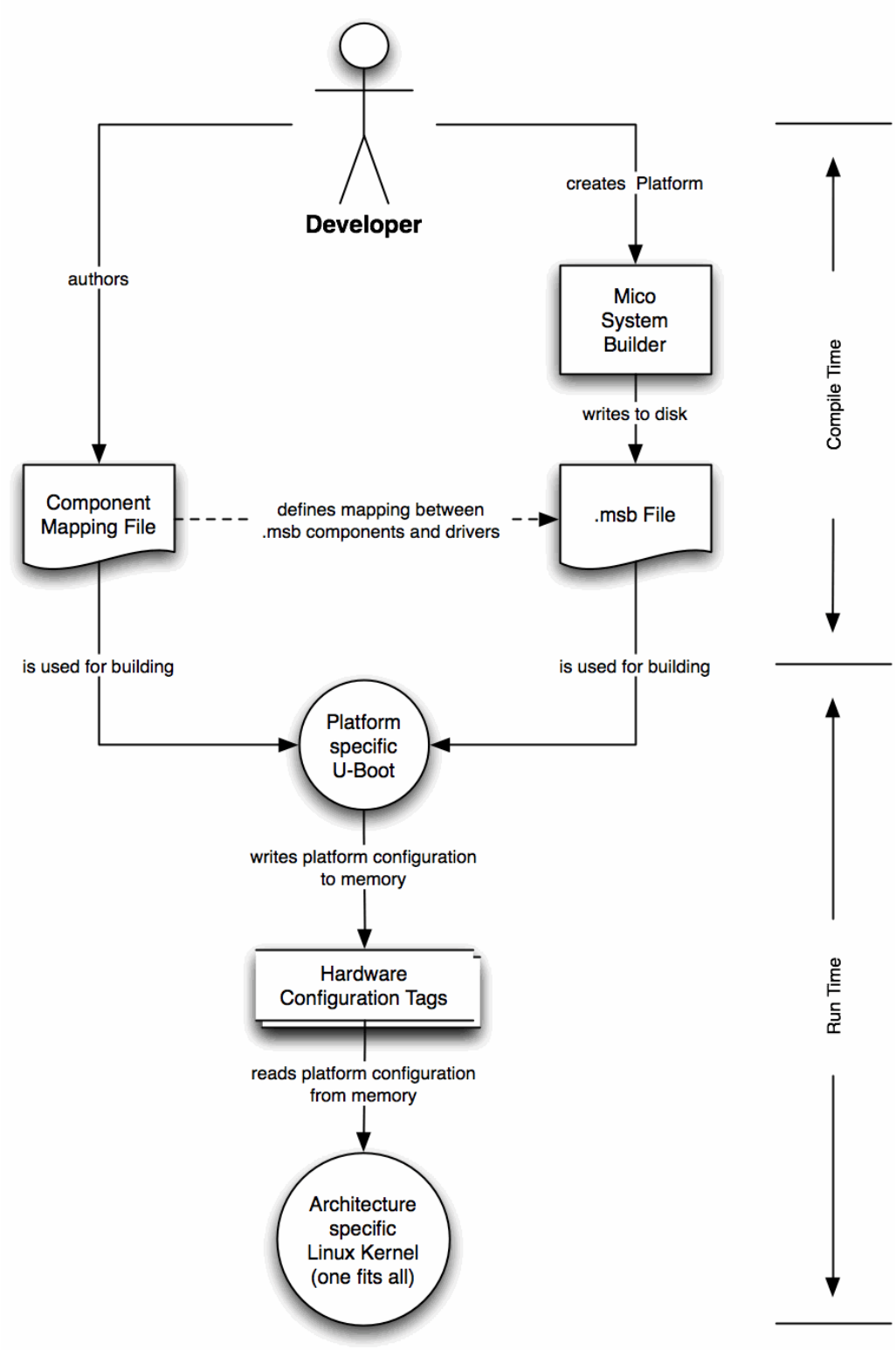
- A port from the U-Boot boot loader to the LatticeMico32 architecture, which is configured from the .xml files generated by the Mico System Builder (MSB) perspective in LatticeMico32 System, using a cross-platform tool implemented in Java (MSBConfigParser)
- A port from the Linux kernel to the LatticeMico32 System nommu architecture
- A userland built from a uClinux distribution tree

The design flow involved in configuring and building a U-Boot and a Linux kernel for a LatticeMico32-based device is as follows:

- The platform is created, and an additional mapping file tells the U-Boot build process about the components and drivers.
- The U-Boot is built from these sources.
- The Linux kernel for the LatticeMico32 System architecture is built independently of the U-Boot configuration and the platform.
- When transferring control, U-Boot tells the Linux kernel about the platform configuration, using a list of tagged structures.

Figure 1 shows how the various tools and components interact.

Figure 1: Design Flow



More Information

For more information on the GCC tool chain and the GDB debugger, go to:

http://www.gnu.org

For more information on uClinux, go to:

http://www.uclinux.org

For more information on Linux, go to:

- http://www.linux.org
- http://kernel.org

2

U-Boot

This chapter describes the U-Boot boot loader, which initializes the board and provides information about the board configuration to the Linux kernel.

U-Boot is an open-source boot loader available for a wide range of embedded processor architectures initiated by Wolfgang Denk. The official Web site of the project is http://www.denx.de/wiki/UBoot.

The Linux port to LatticeMico32 includes a U-Boot port to the LatticeMico32 architecture, which can be highly customized and is adaptable to any Mico System Builder platform generated for the LatticeECP2-50-based development board. However, pre-generated U-Boot configurations and LatticeMico32 platforms for the board are provided in the port, too.

Execution Flow

After you turn on the board, you must initialize the board's components. The U-Boot boot loader provides initialization code and initializes the board. The LatticeMico32 soft-core processor is configured to have a default address where the first instruction is fetched at start-up, so the boot loader must be stored at this address to execute it after the release of the reset.

The early U-Boot initialization assembler code copies the whole boot loader from the current start-up location (usually the flash) to a pre-configured address (usually the RAM) and executes the boot loader from there. If the boot loader already resides at its destination address, the process of copying is skipped and the boot loader is executed directly.

Hardware Configuration

This section describes the pre-generated setups included in U-Boot and explains how to customize U-Boot.

Customized U-Boot

Since the LatticeMico32 platform is highly configurable and neither the specific configuration nor the individual peripherals can be automatically detected, you must make a list of accessible components and their hardware parameters available to the U-Boot boot loader and the Linux kernel.

This information is stored in the .msb file generated by Mico System Builder. This file contains detailed information about the accessible peripherals and their configuration that is not available at run time. A customized U-Boot must be built for the configuration of the generated bitstream, which is read from the .msb platform file, for the LatticeMico32 board and must be rebuilt every time the configuration is changed.

Since the Linux kernel is built independently of the platform, the function of U-Boot in the Linux port to LatticeMico32 is not only limited to the initialization of the board but also provides the information about the board configuration to the Linux kernel. “Hardware Setup Parameters” on page 39 describes in detail how the data is transmitted between the boot loader and the kernel.

Pre-Generated Bitstream

In addition to the ability to use a custom setup, U-Boot in the Linux port to LatticeMico32 includes a pre-generated setup for testing and debugging purposes. This setup contains a prefabricated platform bitstream for a LatticeECP2-50-based development board and all files necessary to rebuild the whole environment for this configuration from scratch.

The pre-generated configuration files are located in a subdirectory of u-boot/board/lattice. Each subdirectory contains the following files:

- platform.bit, which is the platform bitstream used for configuring the board
- MSBplatform.msb, which is the platform description file created by the Mico System Builder in the /soc directory from which the bitstream was built
- MSBcomponents.cfg, which is a component-to-Linux-driver mapping file required by MSBConfigParser for custom bitstreams (see “MSB Configuration File Parser (MSBConfigParser)” on page 12)
- hwsetup_pregenerated.h, which is the header file that adapts U-Boot to the platform configuration that is generated by MSBConfigParser for custom bitsteams
- Makefile, config.mk, and an empty C file, which are files that generate a dummy library that U-Boot depends on for every configuration

The following sections describe the pre-generated bitstreams that are shipped with U-Boot in the Linux port to LatticeMico32.

LatticeECP2-50 Full Setup

The fully featured LatticeECP2-50 configuration setup is located in the ecp250full subdirectory. It includes all peripherals required to run the Linux port to LatticeMico32 and therefore enables you to run Linux on the board.

The full configuration consists of the following components: a LatticeMico32 CPU (75 MHz PLL), a flash, a DDR SDRAM, one UART, three timers, a Tri-Speed Ethernet MAC, a LED, and a 7-segment LED.

Prerequisites for Custom Bitstreams

A LatticeMico32 platform must meet a few prerequisites to be able to use the U-Boot and boot the Linux LatticeMico32 kernel. These prerequisites are mostly related to a minimum set of required components and hard-coded memory addresses.

Required Components

A LatticeMico32 platform requires a minimum set of configured components in order to use U-Boot and the Linux LatticeMico32 kernel:

- LatticeMico32 processor with the following:
  - Divide enabled
  - Multiplier enabled
  - Sign-extend enabled
  - Shifter enabled
  - Debug interface enabled
  - Location of exception handlers at 0x04000000
- 32-MB read/write memory, starting at address 0x08000000
- At least one LatticeMico32 UART instance
- At least one LatticeMico32 timer instance
- At least one LatticeMico32 Tri-Speed Ethernet MAC instance, if you plan to use the Ethernet capabilities of uClinux and U-Boot.

Note

Use of the DDR controller and Tri-Speed Ethernet MAC IP in new platforms requires recompilation in IPexpress and an IP core license. Free evaluation of the IP on an FPGA is provided with about a four-hour timeout before you purchase a license. Other IPs are free of charge.

U-Boot Location in Memory (TEXT_BASE)

Several parts of the U-Boot system depend on the TEXT_BASE constant, which defines where U-Boot locates itself in memory, that is, to which location U-Boot copies itself in the first stage. It is hard-coded in the root makefile of the U-Boot source tree (the default is u-boot/makefile) as for all other U-Boot architectures.

For custom bitstreams, the value of TEXT_BASE must be adjusted to the value of CFG_MONITOR_BASE, which is defined in the configuration header

file of the particular board, for example, u-boot/include/configs/ECP250.h. This adjustment requires knowledge of U-Boot's internal components and should be performed by experienced users only.

Primary Timer Configuration

Since the first specified (that is, primary) timer must perform certain tasks, it must provide a specific configuration (see "Special Issues Regarding U-Boot" and "Component-To-Driver Mapping" on page 13 for a description of how the first component of a driver is determined). The first timer in the LatticeMico32 system must have the following:

- A writable tick count
- A readable tick count
- A start-stop control
- A counter width of 32 bits

The timer code in u-boot/cpu/lm32/interrupts.c assumes this configuration, but the build process will fail as a precaution if the assumptions are not fulfilled.

U-Boot Configuration

This section describes how to invoke the configuration targets of the U-Boot distribution. These targets configure the build tree before the build and play an important role in the LatticeMico32 distribution because the platform is configurable through Mico System Builder.

Pre-Generated Bitstreams

When you use one of the pre-generated bitstreams, the configuration target copies the pre-generated hardware setup header from the respective board configuration directory to include/asm-lm32 in the build directory. A graphical representation of the configuration process is displayed in Figure 4 on page 41. In addition, the cpu/lm32/generate_files.sh script is executed. This script creates a U-Boot linker script called u-boot.lds from the u-boot/cpu/lm32/u-boot.lds.template with the correct TEXT_BASE value and creates a board-specific config.generated.mk makefile with the correct platform-specific compiler flags.

Custom Bitstreams

Custom bitstreams are configured through the LatticeECP250_config make target in a fashion similar to that of the pre-generated bitstream targets, except for the generated header file. This header file is not copied from an existing file of the U-Boot distribution but is generated by the MSBConfigParser tool.

The following files are required to generate the header file:

- The u-boot/cpu/lm32/MSBconfig/compiler_flags.xml file, which is provided by the U-Boot distribution, is the compiler flag extraction configuration file. It may also be shipped by the MSBConfigParser distribution.
- The u-boot/cpu/lm32/MSBConfig/driver_structs.xml file, which is provided by the U-Boot distribution, is the hardware setup payload extraction configuration file. It may also be shipped by the MSBConfigParser distribution.
- The MSBplatform.msb file is the platform file created by the Mico System Builder in the /soc directory. This file contains the components of the generated platform and must be stored in the root of the U-Boot source tree (the default is u-boot/) and renamed to MSBplatform.msb. The configuration target looks for the file in this location.
- The MSBcomponents.cfg file is a user-provided file in the root of the U-Boot source tree. It contains the mapping of the platform components to the U-Boot and Linux drivers. Its syntax is documented in detail in “MSB Configuration File Parser (MSBConfigParser)” on page 12.

Adding a New Pre-Generated Bitstream

This section describes the steps required to add a new pre-generated configuration to the distribution. After executing these steps, you should be able to build a new setup that resembles the pre-generated bitstreams shipped in the original distribution. Use the files and make targets already available as templates for the new configuration that you want to add.

Make Target

Each pre-generated bitstream requires its own make targets in the root makefile (Makefile) and the U-Boot makefile (u-boot/Makefile). The target-naming convention for the root makefile is u_boot_<*name*> and for the U-Boot makefile is <*board_type*><*setup*>_config. The name should represent the configuration setup but can be freely chosen.

The make target in the root file must set the needed environment variables, configure the U-Boot build by running the <*board_type*><*setup*>_config U-Boot make target with the correct parameters, and compile U-Boot by executing “make” in the U-Boot directory.

The purpose of this U-Boot make target is to invoke mkconfig with the appropriate parameters, copying a pre-generated hardware setup header file generated by the MSBConfigParser to u-boot/include/asm-lm32/hwsetup_generated.h, and configuring the TEXT_BASE constant, preferably by using the u-boot/cpu/lm32/generate_files.sh script.

Board Configuration Files

Create a new subdirectory in the u-boot/board/lattice/ directory that identifies the newly created bitstream.

U-Boot depends on a library called lib<*board*>.a by default. Therefore, in the pre-generated bitstream configuration directories shipped with the distribution, an “empty” dummy library is built by the makefile and config.mk from an empty C file. To do the same, copy the makefile and config.mk files to

the new directory, create an empty C file, and name this file to match the corresponding parameter of the mkconfig invocation in the U-Boot root makefile.

Use the .msb file of the platform to generate a U-Boot hardware setup header file using the MSBConfigParser (see “MSB Configuration File Parser (MSBConfigParser)” on page 12). Place this file in the generated directory and rename it to match the name of the file used in the modified U-Boot root makefile.

Adding a New Board Type

Support for a new board type is added with the support for a new pre-generated bitstream, with the exception of another prefix for the directories in u-boot/board/lattice/ and a new board type identifier for U-Boot make targets. The tasks remain the same.

Build System

This section describes the build environment provided by the LatticeMico32 U-Boot distribution.

Make Targets

The distribution provides the LatticeMico32-specific make targets shown in Table 1 for U-Boot.

Table 1: LatticeMico32-Specific U-Boot Make Targets

Target	Description
u_boot_custom	Uses a dynamic configuration from the custom platform and bitstream by using MSBConfigParser.
u_boot_min	Uses a pre-generated minimal configuration for evaluation. The pre-generated bitstream contains no PLL and no DDR RAM, so it does not allow you to run Linux.
u_boot_nonlinux	Uses a pre-generated bitstream with a 100-MHz PLL configuration without DDR RAM but includes several peripherals. You cannot run Linux by using this bitstream.
u_boot	Uses a fully featured configuration with a pre-generated bitstream and hardware setup header file. This pre-generated bitstream enables you to run Linux.

Building U-Boot for a Pre-Generated Bitstream

Use the" make u_boot" command to build U-Boot for the fully featured pre-generated bitstream or the "make u_boot_min" command for the minimal featured bitstream or the "make u_boot_nonlinux" command for the non-Linux platform configuration.

The following example configures and builds a fully featured U-Boot for LatticeMico32 while putting all generated files into u-boot/build/:

```
$ make u_boot
```

Building U-Boot for a Custom Bitstream

Building U-Boot for a custom bitstream requires a compiled and fully configured MSBConfigParser. This task is described in "MSB Configuration File Parser (MSBConfigParser)" on page 12. After compiling the MSBConfigParser and generating the required configuration files, build U-Boot with the u_boot_custom target (that is, with the "make u_boot_custom" command).

"Integrating the MSBConfigParser into the U-Boot Build" on page 14 describes how to build U-Boot for a custom platform configuration in detail and gives an example.

U-Boot Binaries

If the compilation succeeds, the unstripped U-Boot binary for JTAG debugging will reside in u-boot/build/u-boot, ready to be tested, as described in “Testing and Using U-Boot” on page 18. Additionally, a stripped U-Boot binary is generated that can be programmed into flash memory to load the boot loader automatically after the board is reset. You can find this binary in the u-boot/build/ directory. It is named u-boot.bin.

Always be sure to use the correct bitstream corresponding to the U-Boot configuration target that you used to build the boot loader.

MSB Configuration File Parser (MSBConfigParser)

As described previously, the configuration of the board cannot be automatically detected at run time, so you must make a list of accessible components and their hardware parameters available to the U-Boot boot loader at compilation time. The U-Boot boot loader gives this configuration to the Linux kernel at run time. The parameters must be compiled in a configuration header file to be included in the U-Boot boot loader.

MSBConfigParser parses a given .msb platform file, takes the information needed by U-Boot and the Linux kernel to configure all the specified components of the board, and creates a C header file from it. This header file is then used for building and configuring the U-Boot boot loader. At run time, the parameters are passed by U-Boot to a “universal” LatticeMico32 Linux kernel. Additionally, the header file generated by MSBConfigParser contains the compiler flags that are necessary for the compilation of U-Boot and the Linux kernel.

The MSBConfigParser source code is located in toolchains/ MSBConfigParser. It is the only part of the tool chain that is not compiled by the “make vendor_toolchains” target in the release package root makefile. You do not need to recompile MSBConfigParser unless you want to. You can use the pre-built MSBConfigParser.jar Java package with JRE 1.6 (Version 6) or later.

To verify the Java version number, enter the following command on the command line:

```
$ java -version
```

You now see the following response:

```
java version "1.6.0_03"
Java(TM) SE Runtime Environment (build 1.6.0_03-b05)
Java HotSpot(TM) Client VM (build 1.6.0_03-b05, mixed mode,
sharing)
```

To recompile MSBConfigParser, follow the instructions in “Building MSBConfigParser” on page 16.

For further information about the generated header file, its content, and its purpose, see “Hardware Setup Parameters” on page 39.

Component-To-Driver Mapping

The .msb file does not give much information about the purpose of the configured components. But since this information is needed by U-Boot, as well as the Linux kernel, you must provide a component-to-driver mapping file to MSBConfigParser. This mapping associates each platform component with a specific driver that the system can control. This way, U-Boot knows which component provides an essential functionality at a memory base address.

The mapping is defined in a plain text file, where each line has the following format:

`<driver_name>=<component_name>[,<component_name>[...]]`

The value on the left-hand side of the equal sign defines the name of the driver with which the components on the right-hand side are associated. The name of a component in the mapping file is the same as the name of the component in Mico System Builder.

If several components must be mapped to the same driver, write a comma-separated list of components on the right-hand side. The first component in the list is always the primary component of this type. A comment starts with a pound sign (#) and ends at the end of the line.

Here is an example of a component-to-driver mapping file:

```
# Component mapping example
CPU=LM32
ASRAM=sram
FLASH=flash
DDR_SDRAM=ddr_sdram # a comment
UART=uart
TIMER=timer0,timer1,timer2
TRISPEEDMAC=ts_mac_core
SPI_M=spi_master
SPI_S=spi_slave
I2CM=i2c,i2cm_oc
LEDS=LED
7SEG=LED_7Segs
```

Table 2 is a list of supported components and available drivers in MSBConfigParser.

Note

Not all supported drivers have a corresponding and functional driver in the Linux LatticeMico32 port.

Table 2: Supported Components and Available Drivers

Driver	Component
CPU	LatticeMico32 CPU
ASRAM	LatticeMico32 asynchronous RAM controller
FLASH	LatticeMico32 flash controller

Table 2: Supported Components and Available Drivers

Driver	Component
SDRAM	LatticeMico32 single data rate (SDR) synchronous dynamic random-access (SDRAM) controller
DDR_SDRAM	LatticeMico32 double data rate (DDR) synchronous dynamic random-access (SDRAM) controller
DDR2_SDRAM	LatticeMico32 double data rate (DDR2) synchronous dynamic random-access (SDRAM) controller
OCM	OCM memory
UART	LatticeMico32 universal asynchronous receiver-transmitter (UART)
TIMER	Timer
TRISPEEDMAC	LatticeMico32 Tri-Speed Ethernet media access controller (MAC)
GPIO	LatticeMico32 general-purpose input/output core (GPIO)
LEDS	Light-emitting diodes (LED)
7SEG	7-segment LED
I2CM	I2CM
SPI_M	LatticeMico32 serial peripheral interface (SPI) master
SPI_S	LatticeMico32 serial peripheral interface (SPI) slave

Integrating the MSBConfigParser into the U-Boot Build

The U-Boot build process requires the MSBConfigParser tool to construct the hwsetup_generated.h platform configuration header file located in u-boot/include/asm-lm32. This task is integrated in the LatticeECP250_config make target. However, you must perform a few preliminary steps to make this process work:

1. Create the dist directory in MSBConfigParser directory in the toolchains directory:

   `mkdir ./lm32linux-<yyyymmdd>/toolchains/MSBConfigParser/dist`

2. Copy MSBConfigParser.jar in the dist directory created in the previous step:

   `cp <path_to_MSBConfigParser.jar> ./lm32linux-<yyyymmdd>/toolchains/MSBConfigParser/dist/`

3. Copy your MSB platform file (*.msb) to the root of the U-Boot source tree and rename it to MSBplatform.msb.

4. Create your component-to-driver mapping file, name the file MSBcomponents.cfg, and copy it to the root of the U-Boot directory.

5. Export the CLASSPATH variable to provide all the path name information to the MSBConfigParser.jar file. If you are using csh, enter the following:

setenv CLASSPATH /<*full_path*>/lm32linux-<*yyyymmdd*>/toolchains/MSBConfigParser/dist/MSBConfigParser.jar

If you are using bsh, enter the following:

export CLASSPATH=/<*full_path*>/lm32linux-<*yyyymmdd*>/toolchains/MSBConfigParser/dist/MSBConfigParser.jar

After that, you can build U-Boot, using the u_boot_custom make target.

Note

If your CLASSPATH environment variable must contain more than the single path to the MSBConfigParser .jar file and you are building U-Boot under Windows Cygwin, the path to MSBConfigParser must be the first element of the CLASSPATH list because of Cygwin-managed mount constraints.

Following is a template showing how to integrate MSBConfigParser into the U-Boot build:

```
[Copy MSBConfigParser.jar into /dist directory]
$ mkdir /path/to/lm32linux-<yyyymmdd>/toolchains/
MSBConfigParser/dist
$ cp /path/to/MSBConfigParser.jar/MSBConfigParser.jar /path/to/
lm32linux-<yyyymmdd>/toolchains/MSBConfigParser/dist/
MSBConfigParser.jar

[Change directory to the U-Boot source]
$ cd /path/to/u-boot/

[Copy the .msb file to the U-Boot root directory]
$ cp /path/to/platform/soc/platform.msb /path/to/u-boot/
MSBplatform.msb

[Create a driver-to-component mapping]
$ vi MSBcomponents.cfg

[Change directory to Linux sources directory]
$cd /path/to/lm32linux-<yyyymmdd>

[Set CLASSPATH]
$ export CLASSPATH=/path/to/MSBConfigParser/dist/
MSBConfigParser.jar
[Build U-Boot]
$ make u_boot_custom
```

Adding Support for New Drivers and Compiler Flags

The two MSBConfigParser configuration files shown in Table 3 reside in the /config directory.

Table 3: MSBConfigParser Configuration Files

File	Description
driver_structs.xml	Describes all drivers that are supported by the port and how the U-Boot hardware setup header file is generated by using the .msb file. To add a new driver, a new <*driver*> structure must be added, as described in the .xml file in compliance with the DTD file.
compiler_flags.xml	Determines which compiler flags are used to build U-Boot and the Linux kernel. A compiler flag is used if certain configuration conditions are met in the provided .msb platform file. A new compiler flag rule is set by adding a new <*flag*> tag.

The structure of the .xml configuration files is described in the DTD language. See “MSBConfigParser Document Type Definitions” on page 65 for a listing of the DTD files.

Building MSBConfigParser

Unlike the rest of the uClinux build process, building MSBConfigParser on a Windows PC requires you to use a DOS command window, not a Cygwin shell. You must use Apache Ant™ version 1.6.*x* or later and Java Development Kit (JDK)™ 6.0 or later.

You must also copy the toolchains/MSBConfigParser tree outside of a managed mount directory and build the parser.

Note

As described in the “Building the Port in the Windows Environment” section of the *Building the Linux Port to LatticeMico32 System User Guide*, building U-Boot and the Linux LatticeMico32 kernel under Cygwin requires the distribution tarball code to be unpacked on a case-sensitive Cygwin managed mount.

The MSBConfigParser cannot be built while residing in a managed mount because the Java compiler does not have information about the naming conventions of managed mounts. To build (the .jar file of) the MSBConfigParser under Cygwin, you must first copy the toolchains/MSBConfigParser/ tree outside of a managed mount and build the parser at this outside location.

After installing and configuring Apache Ant, use the following command to run the Ant compiler in the base directory of the MSBConfigParser tool (/toolchains/MSBConfigParser/) to compile the source files:

```
ant compile
```

To build a single .jar file (recommended) of the tool, use the following command:

```
ant dist
```

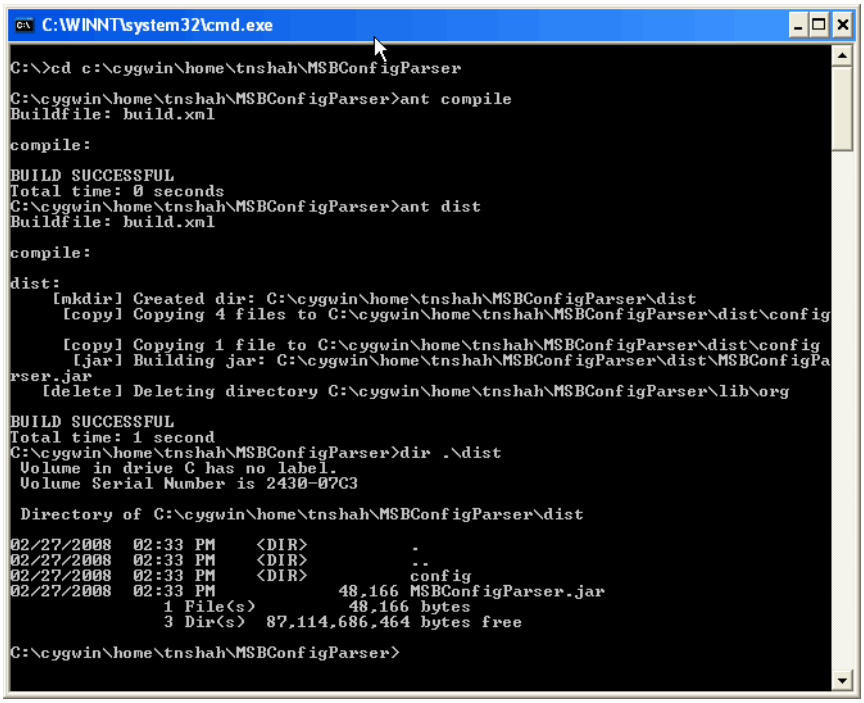
The generated .jar file is placed in the dist directory.

To remove all built files, including the dist directory, use this command:

```
ant clean
```

Figure 2 shows a sample build session for building a MSBConfigParser.jar file in a Windows DOS console window.

Figure 2: Building MSBConfigParser.jar in a DOS Console Window



Using Stand-Alone MSBConfigParser

You can use the MSBConfigParser as a stand-alone application as well. Use the following command to obtain a description of how to use MSBConfigParser:

```
java -jar MSBConfigParser -help
```

Here is an example:

```
$ java -jar MSBConfigParser.jar -help
MSBConfigParser, version 0.3.3
Usage: java -jar MSBConfigParser [Option ...] <msb_file>|
[-help|-version]
Options:
   -m <file>   Set path to the user-generated component-to-
               device-driver mapping information file
               [default: MSBcomponents.cfg]
```

```
-c <dir>    Set the directory containing the driver struct
            description file and the compiler flag
            description file [default: config/]
-report     Print a short report of the components specified
            in the MSB file
-help       This help
-version    Version information
```

MSBConfigParser requires an MSB platform description (*.msb) file to operate on. To generate the configuration header file, use -m <*file*> to specify the location of the component-to-driver mapping file and -c <*dir*> to specify the directory containing the application configuration files.

To simply parse the MSB file and print a list of all components specified in the given board configuration, use the -report flag. This feature is particularly helpful for creating the component-to-driver mapping file because it obviates the need to start the Mico System Builder to see the board configuration in a human-readable form.

Following is an example showing the steps involved in building and using stand-alone MSBConfigParser:

```
[Build MSBConfigParser]
$ cd /path/to/MSBConfigParser/
$ ant dist
[Copy the .msb file to the current directory]
$ cp /path/to/platform/soc/platform.msb .
[Create a driver-to-component mapping with support of the
-report feature]
$ java -jar dist/MSBConfigParser.jar -report platform.msb
$ nano MSBcomponents.cfg
[Create header file]
$ java -jar dist/MSBConfigParser.jar -m MSBcomponents.cfg \
-c config/ platform.msb > hwsetup_generated.h
[View result]
$ less hwsetup_generated.h
```

Testing and Using U-Boot

To test U-Boot after it has been built, you must first start the boot loader on the actual hardware. After it has been started, there are several possibilities for evaluating different parts of the LatticeMico32-specific drivers and the hardware.

The examples in this section assume that U-Boot has been successfully built already.

Prerequisites

Several steps involved in loading U-Boot to the board require an empty section in RAM that is large enough to temporarily store the U-Boot in it. The location is arbitrary, as long as the following section is unused and large enough to fit the U-Boot executable, which is approximately 150 kB.

Assumptions

The examples in the following sections assume the following:

- The TFTP server has an IP address of 192.168.0.1.
- The base address of the flash component is 0x04000000.
- The base address of the SRAM is 0x08000000.
- U-Boot copies itself to and locates itself at a small region at the end of the SRAM; that is, the region starting at the base address of the SRAM (0x08000000) is empty and big enough for copying the U-Boot and other binaries into it.

These assumptions must be customized to fit the configuration of the actual used board.

Writing U-Boot into Memory and Starting from RAM

First the U-Boot boot loader executable must be loaded into the memory of the board. If the generated CPU contains a JTAG interface, you can do this through JTAG.

1. Connect the JTAG interface and the serial (RS232) interface with the host computer, where the LatticeMico32 development tools and GDB are installed.
2. Turn on the board.
3. Start the LatticeMico32 Cygwin shell, and enter the **TCP2JTAGVC2** command in this shell.
4. Start GDB. The correct GDB to use is lm32-elf-gdb for the LatticeMico32 architecture.
5. Use the following GDB command to establish a connection to TCP2JTAGVC2:

   ```
   target remote >ipaddr<:>port<
   ```

6. Using the following GDB command to have the debugger upload the U-Boot executable to the board memory through JTAG:

   ```
   load </path/to/uboot>/u-boot
   ```

7. Once the executable has been loaded into memory, use the following command (shorthand for continue) to start U-Boot.

   ```
   c
   ```

The U-Boot startup information and a U-Boot command prompt should appear on the serial connector.

The following example assumes that the IP address of the host computer is 192.168.0.3 and that the U-Boot executable is located in /tmp/ubootbuild/:

```
$ lm32-elf-gdb
... output from gdb startup ...
(gdb) target remote 192.168.0.3:1000
... output from connecting to remote debugger ...
(gdb) load /tmp/ubootbuild/u-boot
... output from loading u-boot ...
(gdb) c
```

After these commands are executed, a prompt similar to the following should appear on the serial console:

```
U-Boot 1.2.0 (Oct 15 2007 - 12:00:00) [Theobroma Systems]

LatticeMico32 board configuration:
   Device         | Base Address | Additional information
   ---------------+--------------+--------------------------------
   CPU 0          |              | Frequency: 75000000 Hz
   Flash 0        |  0x04000000  | Size: 33554432 (32 MB)
   DDR SDRAM 0    |  0x08000000  | Size: 67108864 (64 MB)
   Timer 0        |  0x80002000  |
   Timer 1        |  0x80010000  |
   Timer 2        |  0x80012000  |
   UART 0         |  0x80000000  | Baud Rate: 115200
   LEDs 0         |  0x80004000  |
   7Segment 0     |  0x80006000  |
   TriSpeedMAC 0  |  0x80008000  |

   LM32 configuration options:
      Hardware multiplier:          enabled
      Hardware divider:             enabled
      Hardware barrel-shifer:       enabled
      Sign-extension instructions:  disabled
      Cycle counter CSR:            disabled
      Instruction cache:            enabled
      Data cache:                   enabled

   lm32mac version 0x10000 @
   0x80008000lm32MAC#0
   In:    serial
   Out:   serial
   Err:   serial
   Hit any key to stop autoboot:  0
   lm32#
```

LED Behavior at U-Boot Start-Up

You can use the LEDs connected to the GPIO pins on the LatticeECP2-50 evaluation board to diagnose problems in U-Boot. Table 4 gives the meaning of the eight LEDs.

Table 4: LED Behavior in U-Boot

LED Number	Description
7	Toggles every second. If this LED blinks continuously, the timer interrupt in U-Boot is served correctly.
6	Currently unused.
5	Currently unused.
4	Currently unused.
3	Currently unused.
2	Currently unused.
1	Currently unused.
0	Currently unused.

Useful U-Boot Built-In Tools

Some of the built-in tools in U-Boot are useful for downloading data, reading data, writing data into the memory and flash memory, starting an application, and booting an operating system:

- boot – Runs the bootcmd U-Boot environment variable (see “Setting Environment Variables in U-Boot” on page 22). It is the boot default.
- bootm – Runs the boot application from memory. Its usage is as follows:

  bootm <*start_address*> <*argument*>
- cp.b – Copies the memory. Its usage is as follows:

  cp.b <*from_address*> <*to_address*> <*length*>
- erase – Erases the flash memory. Its usage is as follows:

  erase <*start_address*> <*end_address*>
- go – Starts the application from an address. Its usage is as follows:

  go <*address*>
- tftp – Downloads data from a TFTP server to memory. Its usage is as follows:

  tftp <*destination_address*> <*file*>

  The TFTP server at the IP address stored in the serverip environment variable is used.

Type one of these commands in the U-Boot shell and press Enter to execute them. For a complete list of U-Boot commands, use the -help command.

Obtaining Board Information

To examine how the board and the memory regions are configured, you can use the following commands:

- bdinfo – Displays some board information, including the memory layout, flash configuration, Ethernet IP and MAC addresses, and the baud rate.
- flinfo – Prints information about the available flash memory.
- irqinfo – Displays information about the configured interrupts.
- coninfo – Displays console devices and information.

Setting Environment Variables in U-Boot

Environment variables in U-Boot are set with the setenv command, following this syntax:

`setenv <variable> <value>`

For example, to set the MAC address, set the ethaddr variable to 12:34:56:78:ab:cd by typing the following:

```
setenv ethaddr 12:34:56:78:ab:cd
```

You can permanently save the environment variables by using the saveenv command, which writes the configuration to a protected sector in the flash memory of the board and is loaded each time that U-Boot starts.

The printenv command prints all set environment variables.

Two environment variables, bootargs and bootcmd, are used to configure and boot the Linux kernel in U-Boot.

- The bootargs variable sets the kernel command-line arguments to provide the kernel with useful information for the boot process. “Linux LatticeMico32 Kernel Boot Options” on page 30 describes in detail which boot arguments are available and how you can set them.
- The bootcmd variable includes all U-Boot commands that should be executed in order to boot an operating system directly after finishing the load process of the boot loader. If you set this option, U-Boot will execute the content automatically after a power cycle or a board reset. Usally this consists of a set of tftp, cp.b, and bootm commands. “Deploying and Using the Linux LatticeMico32 Kernel” on page 33 provides examples showing how to set the bootcmd variable.

  You can abort the automatic boot process by pressing any key while the boot loader displays the `Hit any key to stop autoboot` message.

  The content of the bootcmd variable is also executed when the boot command is started in the U-Boot shell.

Configuring the Automatic Boot Delay

You can change the automatic boot delay at run time.

To change the automatic boot delay:

1. Specify the new automatic boot delay by entering the following:

   `setenv bootdelay <number_of_seconds>`

   Here is an example that sets the automatic boot delay to 10 seconds:

   ```
   setenv bootdelay 10
   ```

2. Save this configuration by entering the following:

   `saveenv`

3. Verify the setting by printing the environment variable:

   `printenv`

To configure the default automatic boot delay for the U-Boot build:

1. Edit the ECP250full.h header file, which is located in the u-boot/include/configs/ directory under the sources installation directory, to modify the following preprocessor definition, as in this example:

   ```
   #define CONFIG_BOOTDELAY    5
   ```

2. Rebuild U-Boot.

Configuring the Network Interface in U-Boot

The environment variables shown in Table 5 are used to configure the network interface in U-Boot.

Table 5: Networking Environment Variables

Variable	Description
ethaddr	Sets the MAC address of the Ethernet interface, for example, 12:34:56:78:ab:cd.
ipaddr	Sets the IP address of the Ethernet interface, for example, 192.168.0.100.
netmask	Sets the net mask, for example, 255.255.0.0.
serverip	Sets the IP address of the server where the TFTP server resides, that is, the server from where the binaries are fetched.
gatewayip	Sets the IP address of the gateway to use.

Changes made to the configuration are applied instantly.

The default MAC address of the network interface, which is set when U-Boot creates the write-protected U-Boot environment configuration sector in the flash memory, is 12:34:56:78:AB:CD.

Note

You can reset the ethaddr environment variable only once for each board. If the MAC address has been changed once, U-Boot will refuse further change requests. See “Changing the Ethernet MAC Address” on page 24 for instructions on changing the MAC address.

The dhcp U-Boot command for obtaining the configuration from a network DHCP server is not available in this build. The network interface must be configured manually.

Following is an example configuration with an IP address of 192.168.0.100, a net mask of 255.255.255.0, and a server and gateway IP address of 192.168.0.1:

```
lm32# setenv ipaddr 192.168.0.100
lm32# setenv netmask 255.255.255.0
lm32# setenv serverip 192.168.0.1
lm32# setenv gatewayip 192.168.0.1
lm32# saveenv
```

Changing the Ethernet MAC Address

The default Ethernet MAC address for the LatticeMico32 Tri-Speed MAC is 12:34:56:78:ab:cd. You can override this default MAC address by setting the ethaddr environment variable. For example, to set the MAC address to 02:00:01:02:03:04, enter the following command:

```
setenv ethaddr 02:00:01:02:03:04
```

Once you change the default MAC address and save the environment variables with the saveenv command, you cannot program another MAC address by simply using the setenv ethaddr command.

To change a programmed MAC address for the LatticeECP2 board:

1. Unlock the last sector (sector 127) of the CFI flash bank 1 by using the following command:

   ```
   protect off 1:127
   ```

   You can use the flinfo command in U-Boot to obtain CFI flash information for the LatticeECP2 board’s CFI flash configuration.

2. Erase the last sector using the following command:

   ```
   erase 1:127
   ```

3. Restart U-Boot, which will use the default settings. Now you can re-program the MAC address. After you erase the flash, all environment variables are erased, so you must reprogram the environment variables that you explicitly set earlier.

Testing the Network Interface in U-Boot

You can test the network interface in U-Boot by using the ping and tftpboot commands.

Ping Command

The ping command requires you to specify an IP address on the command line. This command performs an ARP request to obtain the MAC address of the target. As soon as the ARP request is answered, an ICMP ping request is sent to the target. The response of the target is then reported to you. A typical ping test from U-Boot is the following:

```
lm32# ping 192.168.0.1
Using lm32MAC#0 device
host 192.168.0.1 is alive
lm32#
```

tftpboot Command

The tftpboot command requires you to specify a load address, in hexadecimal, and a file name, including the path, on a TFTP server. This command connects to the TFTP server, retrieves the specified file, and loads it into the memory region, starting at the specified address. A typical tftpboot command invocation is the following:

```
lm32# setenv serverip 192.168.0.1
lm32# tftp 08000000 testfile
Using lm32MAC#0 device
TFTP from server 192.168.0.1; our IP address is 192.168.0.2
Filename 'testfile'.
Load address: 08000000
Loading: #
done
Bytes transferred = 400 (190 hex)
lm32#
```

Writing U-Boot into Flash Memory

Starting U-Boot from flash memory eliminates the need to use GDB and TCP2JTAGVC2 every time you cycle the power on the board. To write U-Boot to flash memory once it is running, you can use either of the following methods.

Writing U-Boot to Flash Memory by Using TFTP

To write U-Boot to flash memory by using TFTP, load a fresh U-boot binary into an empty section of SRAM with the tftp command. You must erase the flash memory and copy the binary from SRAM to the erased flash area. You can use a command sequence similar to the following, assuming that the binary is stored in the u-boot.bin file in the TFTP root on the server and that the binary is not larger than 0x80000:

```
lm32# setenv serverip 192.168.0.1
lm32# tftp 08000000 u-boot.bin
... output from tftp which loads u-boot into sram ...
lm32# erase 04000000 0407ffff
... output from erasing the flash memory ...
lm32# cp.b 08000000 04000000 00080000
```

```
... output from copying from sram to flash memory ...
```

After you issue these commands, you can reset the board, and the new U-Boot will immediately boot from flash memory.

Writing U-Boot to Flash Memory by Using GDB

To write U-Boot to flash memory by using only GDB, load a fresh U-boot binary into SRAM memory through GDB. You must erase the flash memory and copy the binary to the erased flash area. You can use a command sequence similar to the following, assuming that the binary is stored in the U-boot file in the current directory and that a GDB session connected to the target board through JTAG has already been established, as described in “Writing U-Boot into Memory and Starting from RAM” on page 19.

First the U-Boot is loaded into SRAM twice by using GDB:

```
(gdb) load u-boot 0x70000
Loading section .text, size 0x14598 lma 0x8070000
Loading section .rodata, size 0x984 lma 0x8084598
Loading section .rodata.str1.4, size 0x3c16 lma 0x8084f1c
Loading section .data, size 0xbac lma 0x8088b34
Loading section .u_boot_cmd, size 0x438 lma 0x80896e0
Start address 0x8000000, load size 105238
Transfer rate: 8562 bits/sec, 505 bytes/write.(gdb) load u-boot
Loading section .text, size 0x14598 lma 0x8000000
Loading section .rodata, size 0x984 lma 0x8014598
Loading section .rodata.str1.4, size 0x3c16 lma 0x8014f1c
Loading section .data, size 0xbac lma 0x8018b34
Loading section .u_boot_cmd, size 0x438 lma 0x80196e0
Start address 0x8000000, load size 105238
Transfer rate: 6195 bits/sec, 505 bytes/write.
(gdb) c
```

Then U-Boot starts, and you can enter the following commands on the serial console:

```
lm32# erase 04000000 0407ffff
... output from erasing the flash memory ...
lm32# cp.b 08000000 04000000 00080000
... output from copying from sram to flash memory ...
```

After you issue these commands, you can reset the board, and the new U-Boot will immediately boot from flash memory.

3

Linux Port to LatticeMico32 System

This chapter describes the Linux port to LatticeMico32 System.

The Linux kernel is ported to the LatticeMico32 System architecture. Because the LatticeMico32 platform has no MMU and is highly configurable, the assumptions made for Linux ports for other architectures do not apply to this port. The ported Linux kernel is based on the kernel of the uClinux project (http://www.uclinux.org/).

To make the Linux kernel as independent as possible of the present LatticeMico32 platform configuration, almost all configuration-specific details are located only in the U-Boot boot loader and are passed to the “universal” Linux LatticeMico32 kernel at run time.

In association with the port, the Linux device driver for the LatticeECP2-50-based boards are implemented to support a wide range of peripherals. In accordance with the Linux paradigm, the Linux port to LatticeMico32 provides a large number of configuration options that allow easy integration, activation, and exclusion of specific drivers and kernel modules.

An extensive set of userland applications, such as common GNU command-line tools, network daemons, or demonstration applications, from the uClinux distribution tree (http://www.uclinux.org/pub/uClinux/dist/) is supported and shipped with the distribution of the Linux port to LatticeMico32. “Configuring Userland Applications” on page 29 provides instructions on how to extend or shorten the list of built and included applications.

Selecting a Board

Currently, the Linux port to LatticeMico32 is only supported for LatticeECP2-50-based development boards. The “make menuconfig” command enables you to select the targeted product. The default selection is LatticeECP2-50, which includes all platforms based on the LatticeECP2-50 FPGA.

To select another target board:

1. Type **`make menuconfig`** in the root directory of the distribution.
2. In the menu, select **Vendor/Product Selection** and press **Enter**.
3. Select **(...) Lattice Products** and press **Enter**.
4. Select the target board and press **Space**.
5. Press the right arrow key to select **Exit** at the bottom of the screen and press **Enter**.
6. Back in the top-level menu, select **Exit**.
7. Save the new configuration by answering the prompt with **Yes**.

   The dialog box now closes.

Configuring the Linux LatticeMico32 Kernel

The platform-specific configuration is mainly handled in U-Boot. The Linux LatticeMico32 kernel remains almost unaffected. The port provides a default kernel configuration that should be suitable for nearly all applications. The Linux LatticeMico32 kernel is nevertheless highly user-configurable if a certain application depends on a certain feature.

However, there is one configuration detail that becomes important when you use a custom bitstream, that is, a custom platform configuration and hardware setup. (See “U-Boot” on page 5). This setting specifies where the kernel will be loaded and executed. See “Setting the Kernel Load Address” on page 29 for a detailed description of this configuration item.

General Kernel Configuration

As with the default Linux distribution, you can configure the kernel by using the “make menuconfig” command.

To activate the configuration dialog box for the Linux LatticeMico32 kernel:

1. Run **`make menuconfig`** in the root directory of the distribution.
2. In the menu, select **Kernel/Library/Defaults Selection** and press **Enter**.
3. Select **Customize Kernel Settings** and check the corresponding box by pressing **Space**.
4. Press the right arrow key to select **Exit** at the bottom of the screen and press **Enter**.
5. Back in the top-level menu, select **Exit**.

6. Save the new configuration by answering the prompt with **Yes**.

   The dialog box closes. After a few seconds, another dialog box appears, showing the Linux LatticeMico32 kernel configuration.

7. Configure the kernel to fit your needs.

Setting the Kernel Load Address

The TEXT_OFFSET configuration item defines where the Linux kernel is loaded and executed. Before starting the kernel, U-Boot decompresses the kernel image and copies the kernel to this specified address. You must manually set the load address of the kernel when you use a custom platform bitstream. The default value, which is adjusted to the configuration of the pre-generated bitstreams, is 0x08000000, the base address of the SRAM component.

When you use a custom bitstream, you must set the value to an address in SRAM that has a subsequent unused portion that is large enough to store the kernel.

To change the kernel load address:

1. Enter the kernel configuration menu, as described in “General Kernel Configuration” on page 28.
2. Select **Processor, Board, and Features** and press **Enter**.
3. Modify the **Set the text offset in memory configuration** item in accordance with the present platform setup.

Since U-Boot unpacks and copies the kernel to the TEXT_OFFSET address, the kernel image must be located at another address when TFTP is used to download the image to RAM. If the location of the image overlaps the destination memory range, the process of decompressing and copying will fail.

If the value of TEXT_OFFSET has been changed, you must adapt the following line in /vendors/Lattice/ECP250/makefile at the make target image:

```
$(MKIMAGE) -A lm32 -T kernel -C gzip -a 0x8000000 -e 0x08000000
```

Replace both occurrences of 0x08000000 with the new value of TEXT_OFFSET in order to build a valid kernel image.

Configuring Userland Applications

You can configure the set of userland applications that are built and included in the generated RAM file system image.

To configure the userland applications:

1. Type **`make menuconfig`** in the root directory of the distribution.
2. In the menu, select **Kernel/Library/Defaults Selection** and press **Enter**.

3. Select **Customize Vendor/User Settings** and check the corresponding box by pressing **Space**.
4. To declare that your selection is the new default setting, check **Update Default Vendor Settings**.
5. Press the right arrow key to select **Exit** at the bottom of the screen and press **Enter**.
6. Back in the top-level menu, select **Exit**.
7. Save the new configuration by answering the prompt with **Yes**.

   The dialog box closes and after a few seconds another dialog box appears, showing the userland configuration.

Some applications are provided by BusyBox, which combines many common UNIX utilities into a small executable. BusyBox must be configured independently of the rest of the userland.

To configure BusyBox:

1. Change to the /path/to/busybox directory.
2. Type **`make menuconfig`** in this directory.
3. Activate or deactivate the utilities that you want to include or exclude.
4. Exit and save your new configuration.

Linux LatticeMico32 Kernel Boot Options

The Linux kernel can use information given in the form of command-line options at boot time. The parameters that must be set depend on the present setup. In the case of the Linux port to LatticeMico232, these parameters are used to supply the kernel with information about the environment in which the board is running. This information is mainly about the present root file system, RAM disk management, and network-related configuration topics.

A boot parameter is in the form <*option_name*>=<*value*>. A set of more than one parameter is a space-separated list of parameters: <*parameter1*> <*parameter2*> <*parameter2*> ... <*parameterN*>.

Bootargs Environment Variable

You can set the boot parameters for the Linux LatticeMico32 kernel by modifying the bootargs environment variable in U-Boot. The boot loader provides the contents of this variable to the kernel at start-up. You can permanently save this environment variable for later use, like any other environment variable, by executing the saveenv command.

There are a variety of different Linux kernel parameters, and none of them are specific to the Linux LatticeMico3 kernel port. All parameters that a general kernel understands can be used with the Linux LatticeMico3 kernel, so the following sections list only the most useful options to use with the Linux LatticeMico3 kernel.

Root File System Options

Following are the Linux kernel root file system options:

- root= – Tells the kernel what device is to be used as a root file system while booting. /dev/nfs is required for NFS root file systems, and /dev/ram0 is required for RAM disks.
- rootfstype= – Provides a comma-separated list of file system types that will be evaluated for a match when you try to mount the root file system. The default value is ext2, which is suitable for RAM disks. Set it to nfs when you use an NFS root file system.
- nfsroot= – Tells the kernel which machine, what directory, and what NFS options to use for an NFS root file system.You can find detailed information about this parameter in the /linux-2.6.x/Documentation/nfsroot.txt file.
- ip= – Tells the kernel how to configure device IP addresses and how to set up the IP routing table. The syntax is as follows (all parameters must be written in the same line):

```
ip=<client-ip>:<server-ip>:<gateway-ip>:<netmask>:\
<hostname>:<device_name>:<autoconf>
```

To use a DHCP server to configure the network interface, use the ip=dhcp kernel parameter. For a more detailed description of the parameters, see /linux-2.6.x/Documentation/nfsroot.txt in the source distribution tarball.

RAM Disk Management Options

There is only one RAM disk management option:

- ramdisk_size= – Sets an upper limit on the size of the RAM disk. The default value, which is 4096, is too small for the built RAM disk within the scope of this port, so you must set this parameter to a much higher value like 12288 (12 MB) when you use a RAM disk.

Miscellaneous Options

Other Linux kernel options are the following:

- console= – Tells the kernel which device is used as the first virtual terminal to capture boot-time messages. Set the parameter to ttyS0,115200 to use the first serial port with a baud rate of 115200.
- init= – Sets the application that is started directly after finishing the boot process. The default value (that is, application) is init. If you want to skip the login process, use init=/bin/msh to start the built-in shell directly.
- quiet= – Suppresses a kernel message about hardware detection at boot time.

Example

The following example configures the kernel to use a RAM disk. The network interface is configured by using a DHCP server. The boot command line is set in U-Boot by modifying the bootargs environment variable:

```
lm32# setenv bootargs 'root=/dev/ram0 ip=dhcp
console=ttyS0,115200
ramdisk_size=16384'
```

The next example tells the kernel to use an NFS root file system. The network interface is configured manually. Additionally, the initialization process is skipped and the msh shell is started directly:

```
lm32# setenv bootargs 'rootfstype=nfs root=/dev/nfs
nfsroot=192.168.0.1:/usr/local/
nfsroot,rsize=1024,wsize=1024,tcp
ip=192.168.0.100:192.168.0.1:192.168.0.1:255.255.255.0:::
console=ttyS0,115200 init=/bin/msh'
```

Building the Kernel and the Initial RAM Disk

Once you have configured the kernel and the desired userland with the “make menuconfig” command, use the “make dep” and “make” commands to build the kernel, build the userland applications, generate the ROM file system tree, create the kernel image, and create the initial RAM disk image:

```
make dep
make
```

The file system tree with all userland applications is built in the romfs/ directory, which represents the root file system. You can find the Linux LatticeMico32 vmlinux.img kernel image and the initial ext2 U-Boot RAM disk in the images/ directory after the build process finishes. The contents of the file system tree and the initial RAM disk are identical.

The kernel image contains the Linux LatticeMico32 kernel, which U-Boot can load and execute on the board. The initial RAM disk, which contains the full file system, is loaded by U-Boot into RAM and mounted by the kernel as a volatile RAM file system. The contents of the romfs/ directory can be used to set up a root NFS environment, where the board mounts an NFS share over the network as its root file system. This setup is preferable, since the NFS share is usually non-volatile, and changes made to the file system are not reversed when the board is reset.

Initial RAM Disk Limitations

Linux kernel 2.6 introduced a new format for initial RAM disks called initramfs. Initrd images, which are currently created, are converted into initramfs at boot time. This task requires free space in RAM the size of the initrd file itself, so the maximum file size of the initrd.img image is limited to the following:

```
(RAM size - U-Boot size - Kernel size) / 2
```

Because of this limitation, a RAM disk image larger than 16 MB is not recommended.

Deploying and Using the Linux LatticeMico32 Kernel

After successfully building the kernel and the initial RAM disk images, you can deploy the kernel and test it on the board. You first see a description of how to load the images onto RAM with a TFTP server. The following section describes how to use an NFS server instead of using the initial RAM disk. Afterwards, it describes how to program the kernel and the initial RAM disk into flash memory and start them from there.

All examples in this chapter are based on the assumption that U-Boot has been successfully built and programmed to the board's flash memory. Each example uses only a certain kernel command-line option configuration. You must adapt these boot options to your network environment.

Assumptions

The examples in the following sections are based on these assumptions:

- A TFTP and a DHCP server are available.
- The TFTP server has an IP address of 192.168.0.1.
- The vmlinux.img and initrd.img files reside in the lm32linux/ directory of the TFTP server.
- The base address of the flash component is 0x04000000.
- The base address of the SRAM is 0x08000000.
- A serial console, such as PuTTY, HyperTerminal, Picocom, or Minicom, is connected to the board.
- U-Boot is programmed into the board's flash memory, and the network interface is configured in U-Boot (see "Testing and Using U-Boot" on page 18).

Loading the Kernel and the Initial RAM Disk Through TFTP

You must load the kernel and the initial RAM disk into RAM by U-Boot. The boot loader starts the kernel and tells the kernel that a RAM disk is present and where it is located in RAM. Then the kernel converts the RAM disk to initramfs and mounts the result as its root file system at boot time.

To load the kernel and the initial RAM disk through TFTP:

1. Use the serial console and the U-Boot tftp command to load the images with TFTP to the board's RAM:

   ```
   lm32# tftp 08200000 /tftp/path/to/vmlinux.imglm32# tftp
   08400000 /tftp/path/to/initrd.img
   ```

   The kernel image is loaded 2 MB behind the location in which U-Boot decompresses the image (0x08000000 in this setup; see "Setting the

Kernel Load Address" on page 29), and the initial RAM disk is downloaded 2 MB behind the kernel start address.

2. Set up the kernel command line with something similar to the following (see "Linux LatticeMico32 Kernel Boot Options" on page 30):

```
lm32# setenv bootargs 'root=/dev/ram0 ip=dhcp
console=ttyS0,115200 ramdisk_size=12288'
```

3. To boot the kernel, use the bootm command, as described in "Useful U-Boot Built-In Tools" on page 21:

```
lm32# bootm 08200000 08400000
```

   The first parameter defines the start address of the application to start (that is, the kernel image location). The second parameter tells the kernel that a RAM disk is used and where in memory it can be found:

   U-Boot then verifies both images and boots Linux. After booting, a login prompt appears.

4. Log in as user "root" with the password "lattice":

```
lm32_eval_ecp250 login: root
Password:
```

5. To automate the process of downloading the images and initializing booting, use the bootcmd environment variable in U-Boot (see "Setting Environment Variables in U-Boot" on page 22):

```
lm32# setenv bootcmd 'tftp 08200000 /tftp/path/to/
vmlinux.img;
      tftp 08400000 /tftp/path/to/initrd.img; bootm 08200000
      08400000'lm32# saveenv
lm32# boot
```

Using NFS Root

Instead of using an initrd image, you can set up an NFS root configuration. The term "NFS root" describes an operating system that mounts its root file system through NFS (Network File System). This setup is very useful when a non-volatile file system is required, the generated initial RAM disk is too large for the present RAM, a more efficient solution than the initial RAM disk is desired, or applications are developed and debugged where only single binaries have to be rebuilt regularly.

To set up an NFS root configuration:

1. Download the kernel through TFTP to the board, using the serial console and the tftp command:

```
lm32# tftp 08200000 /tftp/path/to/vmlinux.img
```

   The kernel image is loaded 2 MB behind the location in which U-Boot decompresses the image (0x08000000 in this setup; see "Setting the Kernel Load Address" on page 29).

   The kernel command-line boot options are used to tell the kernel that an NFS root file system is used (see "Setting Environment Variables in U-Boot" on page 22). The following example assumes that the /usr/local/nfsroot directory is exported by the NFS server at 192.168.0.1:

```
lm32# setenv bootargs 'rootfstype=nfs root=/dev/nfs
      nfsroot=192.168.0.1:/usr/local/
nfsroot,rsize=1024,wsize=1024,tcp
ip=dhcp console=ttyS0,115200'
```

2. Boot the kernel by using bootm (see “Useful U-Boot Built-In Tools” on page 21). This time any second argument is omitted since no RAM disk is used:

   ```
   lm32# bootm 08200000
   ```

3. Optionally, use the bootcmd environment variable (see “Setting Environment Variables in U-Boot” on page 22) to automate the process of booting after a board reset:

   ```
   lm32# setenv bootcmd 'tftp 08200000 /tftp/path/to/
   vmlinux.img;
         bootm 08200000'
   lm32# saveenv
   lm32# boot
   ```

Starting the Kernel from Flash Memory

Instead of downloading the kernel image every time the board is reset, you can program the kernel image into the flash memory and start it from there. This method is useful when the board operates in an environment without a TFTP server and the kernel image is not rebuilt regularly.

To start the kernel from flash memory:

1. Download the kernel image to RAM as usual, using the serial console and the tftp command in U-Boot:

   ```
   lm32# tftp 08000000 /lm32linux/vmlinux.img
   ```

   Since this kernel image is not booted, you can load the image in the start address of the RAM component.

   The U-Boot binary already resides in the first two sectors of the flash memory, so the kernel images must be programmed to a location behind. The 0x04100000 memory address is used, and a region with the size of 0x100000 (four flash sectors) is reserved for the kernel image.

2. Erase this memory location first:

   ```
   lm32# erase 04100000 041fffff
   ```

3. Now use the cp.b command (see “Useful U-Boot Built-In Tools” on page 21) to copy the kernel image to the flash region emptied in the last step (<*from*> <*to*> <*length*>):

   ```
   lm32# cp.b 08000000 04100000 00100000
   ```

   The kernel is started directly from flash, assuming that the initial RAM disk is still used (see “Loading the Kernel and the Initial RAM Disk Through TFTP” on page 33):

   ```
   lm32# tftp 0x08400000 /lm32linux/initrd.img
   lm32# bootm 04100000 08400000
   ```

4. To boot the kernel automatically after a reset of the board, type the following in U-Boot:

```
lm32# setenv bootcmd 'tftp 0x08400000 /lm32linux/initrd.img;
      bootm 04100000 08400000'
lm32# saveenv
lm32# boot
```

Using the Initial RAM Disk from Flash Memory

If neither a TFTP server nor an NFS is available, you can program the RAM disk to the flash memory. Since the kernel must convert the initrd image to initramfs in order to be able to use it, the RAM disk cannot be used directly from the flash memory. You must copy the image from the flash memory to RAM before you boot the kernel by using the U-Boot cp.b command.

To program the RAM disk to the flash memory:

1. First, download the initial RAM disk to any location in RAM, using the serial console and the tftp command:

   ```
   lm32# tftp 08000000 /lm32linux/initrd.img
   ```

2. Erase an empty flash memory region large enough for the RAM disk.

   Assuming that the kernel image has been programmed into the 0x04100000 - 0x041FFFFF region, as described in “Starting the Kernel from Flash Memory” on page 35, the adjacent region is chosen for programming the initial RAM disk. The required size of the region depends on the size of the initrd.img file. Convert the file size to a hexadecimal value, use the flinfo command in U-Boot to see a table of flash sectors, and choose a set of connected flash sectors that is large enough to hold the image.

   Given that one flash sector has a size of 0x40000, if the initial RAM disk has 12 MB + a 64 byte header—that is, 12,582,976 bytes—a memory region of 0xC00040 is required. Given that one flash sector has a size of 0x40000 and 0x04200000 + 0xC00040 = 0x04E00040, the region 0x04200000 to 0x04E80000 must be reserved for the initial RAM disk image (see “Useful U-Boot Built-In Tools” on page 21):

   ```
   lm32# erase 04200000 04e80000
   lm32# cp.b 08000000 04200000 00C80000
   ```

   Note

   Programming the flash memory with this many sectors is a very time-consuming task and will take several minutes to complete.

3. As stated earlier, the initial RAM disk image cannot be used directly from flash, so you must copy it from the flash memory to RAM before you boot the kernel. Use the cp.b command:

   ```
   lm32# cp.b 04200000 08400000 00C80000
   ```

4. Boot the kernel now with bootm, as usual, using the kernel directly from the flash memory with the RAM disk copied to RAM:

   ```
   lm32# bootm 04100000 08400000
   ```

5. Use the bootcmd environment variable to automate the process of copying and booting the kernel upon reset of the board:

```
lm32# setenv bootcmd 'cp.b 04200000 08400000 00C80000
      bootm 04100000 08400000'
lm32# saveenv
lm32# boot
```

Starting the Linux Kernel

This section gives a short description of the process involved in booting the Linux LatticeMico32 kernel and initializing the process control.

Linux Kernel Boot Process

The overall boot process of the Linux LatticeMico32 kernel is the same as the boot process of the general kernel of similiar architectures, except for the hardware configuration passing from U-Boot to the kernel at run time. Refer to “Passing Parameters to the Linux Kernel” on page 48 for information on this topic.

Process Control Initialization

By default, the kernel executes the initialization program after initializing the hardware. This application is the parent of all the processes executed on Linux.

By setting the init= kernel boot argument, you can replace this process with another program, if needed.

The initialization process executes the etc/rc shell script, which handles several initialization tasks, such as mounting procfs and sysfs and changing file permissions where required. The source for this file is located in vendors/Lattice/ECP250/etc/rc of the source tarball. Modify the contents of this file to fit your needs and re-generate the initrd image by executing “make romfs” and “make image.”

After executing the shell script, the initialization process starts the login program, that is, the login prompt application.

4

Hardware Setup Parameters

U-Boot and the Linux kernel must have access to the hardware parameters used to generate the LatticeMico32 platform in order to operate the devices in LatticeMico32. These parameters are written to an .msb file by the Mico System Builder during the generation of the LatticeMico32 platform. This chapter describes the describes these parameters.

The MSBConfigParser tool, which can extract the necessary settings from the .msb file, was created for the LatticeMico32 U-Boot port. The usage and configuration of this tool are documented in detail in “MSB Configuration File Parser (MSBConfigParser)” on page 12. This tool assists in U-Boot configuration for a specific LatticeMico32 platform in the following way: You configure the platform and specify the drivers in the component mapping .cfg file. You run the MSBConfigParser on this file and on the .msb file. The output is written to a C header file in the U-Boot build directory. This header file contains initialization code for structures holding all hardware parameters used for the generation of the specific LatticeMico32 platform where U-Boot is intended to run. This header file is included by a C module that is linked to U-Boot. By including the platform-specific header file, asm-lm32/hwsetup.h, the LatticeMico32 port-specific U-Boot modules can access all hardware parameters that were extracted from the .msb file.

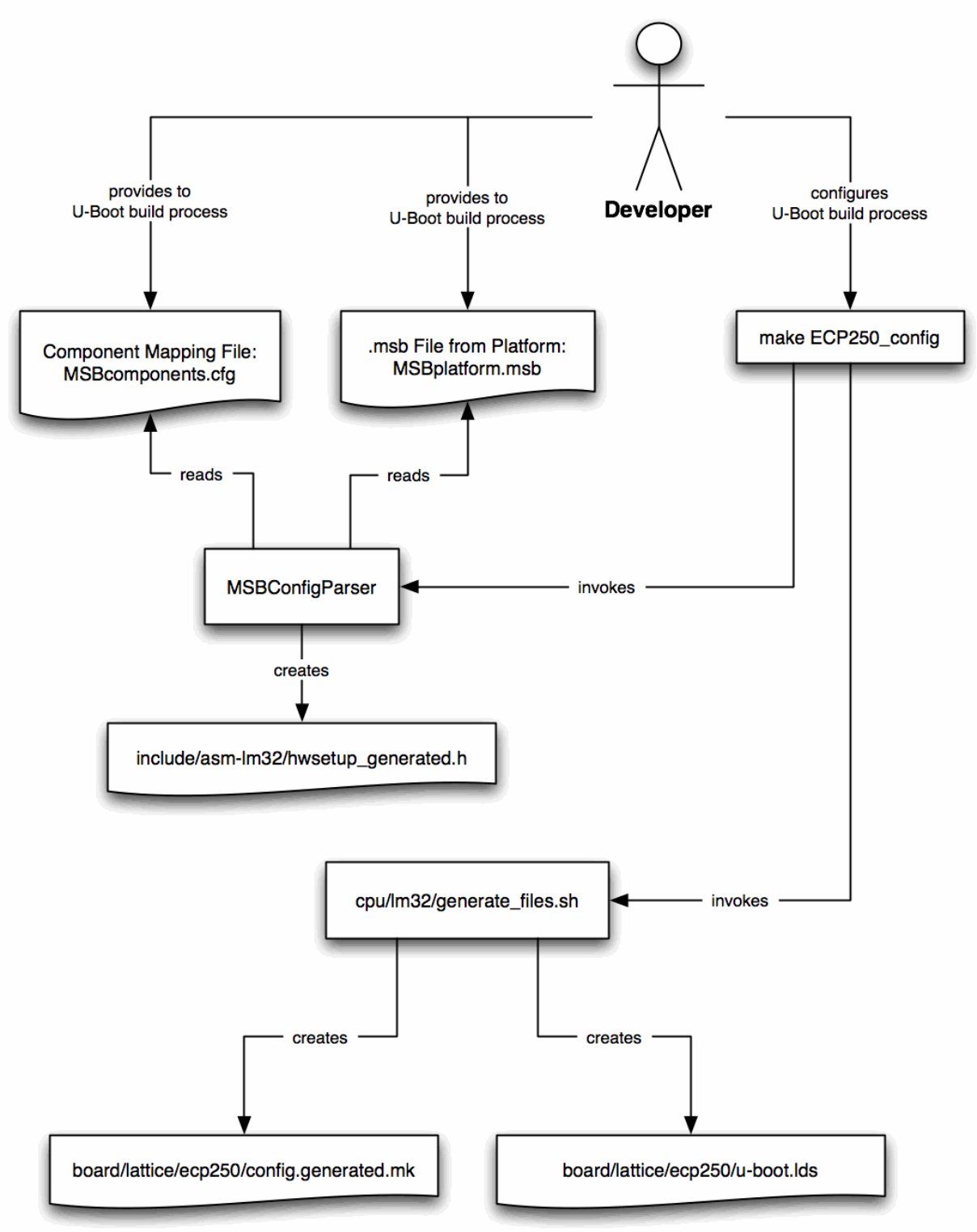
Build Process Integration

The U-Boot build process expects the configuration header created by MSBConfigParser to be located in the include/asm-lm32/hwsetup_generated.h file in the build directory.

Several configuration targets in the root makefile of the U-Boot distribution create this file:

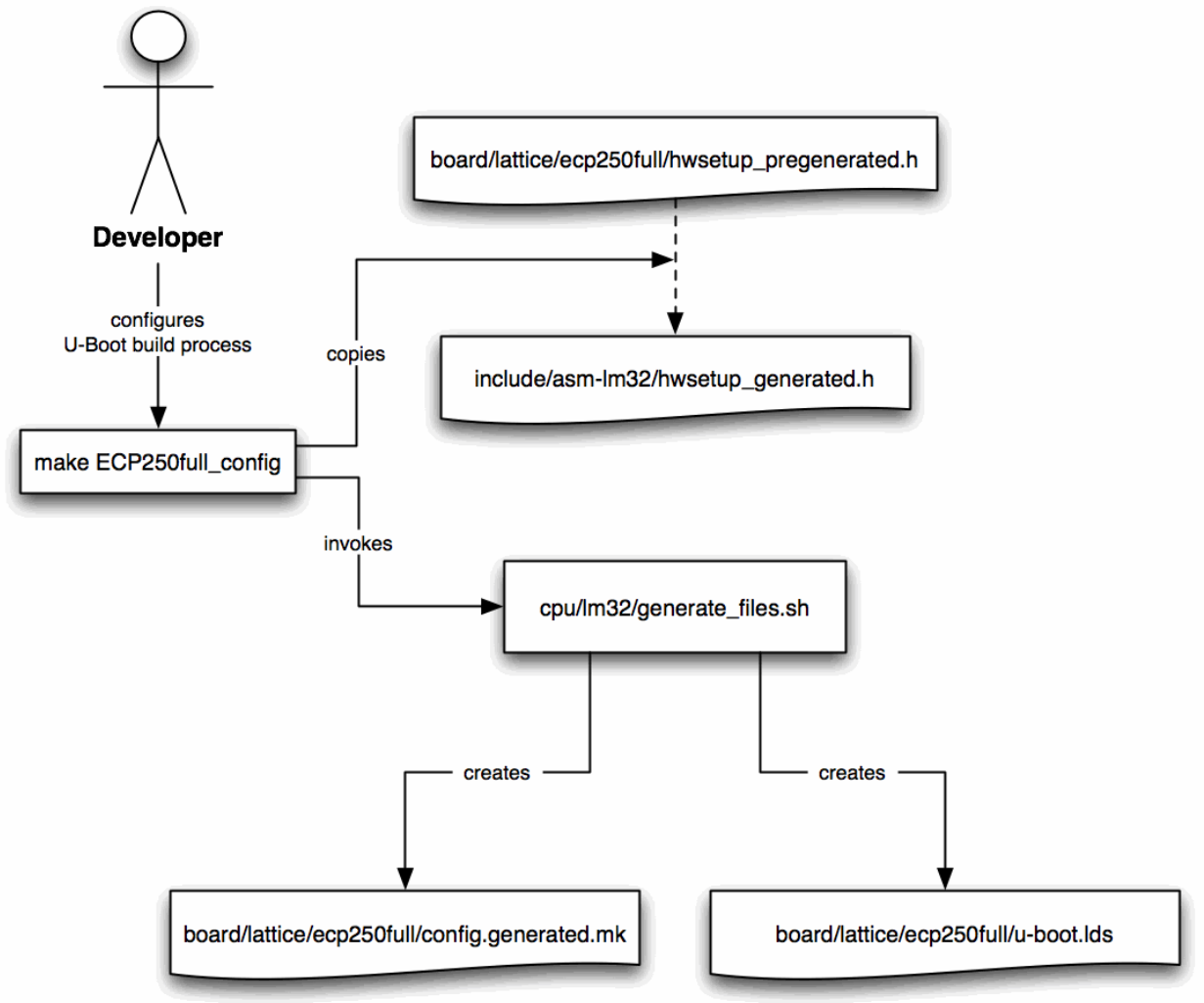
- The ECP250_config make target uses MSBConfigParser to generate hwsetup_generated.h on demand. This process is shown in Figure 3.

Figure 3: U-Boot Configuration from Custom Bitstreams



- The ECP250full_config make target copies the file from hwsetup_pregenerated.h in the /u-boot/board/lattice/ecp250full/ directory. This process is shown in Figure 4.

Figure 4: U-Boot Configuration from Pre-Generated Bitstreams



- The ECP250min_config make target copies the file from hwsetup_pregenerated.h in the u-boot/board/lattice/ecp250min directory.
- The ECP250nolinux_config make target copies the file from hwsetup_pregenerated.h in the u-boot/board/lattice/ecp250nolinux/ directory.

The configuration targets that use pre-generated header files do not require Mico System Builder to be present to configure and build U-Boot. These targets also contain pre-generated bitstreams in their respective board directories. For detailed information on pre-generated bitstreams, see “Pre-Generated Bitstreams” on page 8.

Each bitstream is shipped with the .msb file (Mico System Builder platform file) from which it was generated. The .msb files are located in u-boot/board/lattice/<*board+configuration*>/MSBplatform.msb, where <*board+configuration*> is the abbreviation of the board type, followed by the identification string of the desired pre-generated bitstream.

Data Interface

The interface to the hardware setup information is the C structure data types defined in include/asm-lm32/hwsetup_payload.h. The data types defined are named LM32Tag_<*component_type*>. Some LatticeMico32 components may have several drivers, although all drivers correspond to the same system part in Mico System Builder. One example is GPIO pins that can be used either by a plain GPIO pin driver or by an LED driver or by a seven-segment display driver. In such cases, the hwsetup_payload.h header provides alias types through the typedef mechanism. The maximum length of system part names in the Mico System Builder file is defined in include/asm-lm32/hwsetup_payload.h. Currently it is set to 32 characters.

This header file contains the following items:

- Each alias type (for example, LM32Tag_LEDS_t) is a type without the “struct” prefix, so there is no structure type for alias types (for example, for the LEDS driver, there is no data type called LM32Tag_LEDS).
- Each structure LM32Tag_<*component_type*> type is enclosed in a typedef instruction that defines a corresponding LM32Tag_<*component_type*>_t type. For each driver type, there is always a structure type without a “struct” prefix and with an “a _t” suffix, allowing you to write code without knowing whether the driver is an alias to another driver or whether this driver has defined its own structure type.
- The name of each LatticeMico32 component is stored in a character array of fixed length. The disadvantage of this approach is that the length of the names must never exceed this defined length. The advantage is that the string of the name is always stored completely enclosed in the structure, and there is no pointer reference to another memory area where the string is stored. This advantage is especially important for the interface to the Linux kernel, which also uses these structures.

For an overview of the structure data types, see “The hwsetup_payload.h File” on page 57, which provides a full sample hwsetup_payload.h header file with inline documentation.

U-Boot Hardware Setup Module

The generated header file is used to instantiate the hardware setup data structures in the lib_lm32/hwsetup.c module. This module is linked to U-Boot and can be accessed from any other U-Boot module by using the asm-lm32/hwsetup.h header file, as explained in “Accessing Hardware Setup Parameters in U-Boot” on page 45.

The principle of the hardware parameter data structure initialization works as follows:

- hwsetup.c can use the types of all payload structures by including asm-lm32/hwsetup_payload.h.
- The generated header is also included before anything else is done.
- For each driver type where a component is configured in the component mapping file, the generated header contains two #define statements, one statement defining the number of components for this driver and another statement defining the structure initializer code.
- If there is no #define statement for the number of components, hwsetup.c creates a variable called hsp_num_<*component_type*>, where the component type is set to zero and is replaced by an empty array of configuration structures for the same component name and for the correct component type.
- If there is no #define statement for the number of components, the same variable is defined, but the amount is set to the defined value. An array of payload structures that can hold the number of configured component configurations is also defined. This array is then initialized by using the structure initializer defines.

The following examples, which involve two SDRAM components, show how the generated header is integrated into the C modules.

Here is the SDRAM configuration structure definition in include/asm-lm32/hwsetup_payload.h:

```
/*
 * SDRAM
 */
typedef struct LM32Tag_SDRAM
{
 /* Instance Name */
 char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
 /* Base Address */
 u32 addr;
 /* Memory Size */
 u32 size;
} LM32Tag_SDRAM_t;
```

Next are the SDRAM-relevant contents of the generated header file:

```
/* SDRAM component configuration */
#define LM32TAG_CFG_NUM_SDRAM 2
#define LM32TAG_CFG_STRUCTS_SDRAM \
```

```
 { /*name*/ "ram1_1meg", /*addr*/ 0x02000000, /*size*/
  1048576},
 { /*name*/ "ram2_2meg", /*addr*/ 0x04000000, /*size*/
  2097152},
#define LM32TAG_CFG_SDRAM0_ADDR 0x02000000
#define LM32TAG_CFG_SDRAM0_SIZE 1048576
```

The LM32TAG_CFG_..._ADDR and LM32TAG_CFG_..._SIZE definitions are generated for the first component of each driver and are used by certain special U-Boot components that must have access to these values at compilation time and cannot retrieve this data during run time. For an explanation of the peculiarities of the Linux port to LatticeMico32 with regard to such configuration issues, see “Special Issues Regarding U-Boot” on page 47.

Following are the SDRAM-relevant contents of the lib_lm32/hwsetup.c file:

```
#if defined(LM32TAG_CFG_NUM_SDRAM)
unsigned long lm32tag_num_sdram = LM32TAG_CFG_NUM_SDRAM;
LM32Tag_SDRAM_t lm32tag_sdram[LM32TAG_CFG_NUM_SDRAM] = {
 LM32TAG_CFG_STRUCTS_SDRAM
};
#else
unsigned long lm32tag_num_sdram = 0;
LM32Tag_SDRAM_t* lm32tag_sdram = NULL;
#endif
```

The interface to this data is defined the following way in hwsetup.h:

```
extern unsigned long lm32tag_num_sdram;
extern LM32Tag_SDRAM_t lm32tag_sdram[];
```

The data can be accessed from anywhere in U-Boot by using code similar to the following:

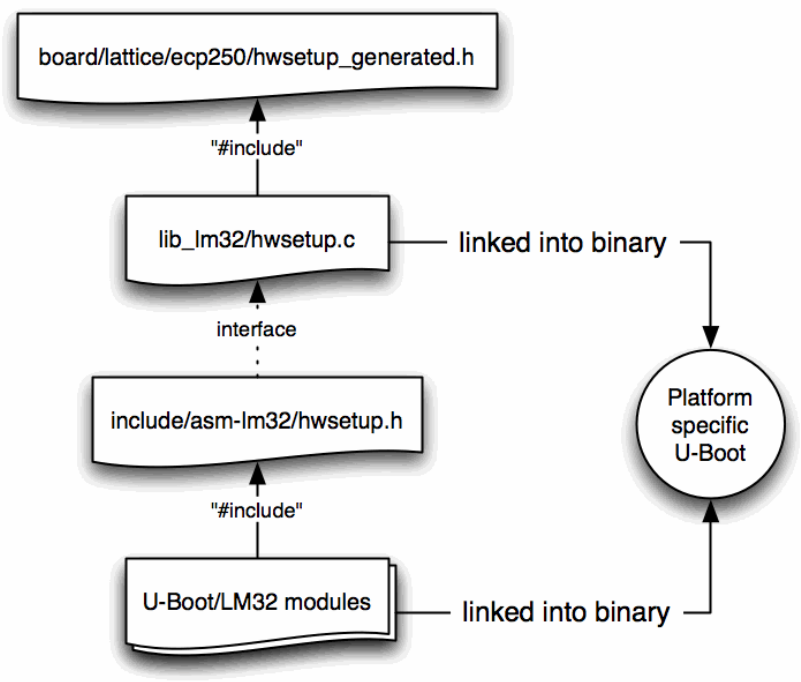
```
/* include the interface to the hardware setup data structures
*/
#include <asm-lm32/hwsetup.h>

/* number of sdrams: */
u32 num_sdrams = lm32tag_num_sdram;
/* Base address of first sdram: */
u32 sdram1_base = lm32tag_sdram[0].addr;

/* Size of second sdram: */
u32 sdram2_size = lm32tag_sdram[1].size;
```

The U-Boot code structure for the hardware configuration is shown in Figure 5.

Figure 5: U-Boot Hardware Setup Code Structure



Accessing Hardware Setup Parameters in U-Boot

You can access the configuration data from any module in U-Boot by including the include/asm-lm32/hwsetup.h header, which provides extern declarations for the symbols in the hwsetup.c module.

- One extern unsigned long hsp_num_...; – Is the declaration for the number of components mapped to the driver identified by ..., for example, extern unsigned long hsp_num_ddr_sdram.
- One extern LM32Tag_..._t lm32tag_...[]; – Is the declaration for the contents of the hardware information structures organized in an array, for example, extern LM32Tag_DDR_SDRAM_t lm32tag_ddr_sdram[];.

The following example shows how to access several hardware setup parameters:

```
/* include the interface to the hardware setup data structures
*/
#include <asm-lm32/hwsetup.h>

/* number of timers: */
u32 num_timers = lm32tag_num_timer;

/* frequency of cpu: */
u32 freq = lm32tag_cpu[0].frequency;
```

```
/* IRQ number of second timer: */
u8 timer2_irq = lm32tag_timer[1].irq;
```

Additional mechanisms are implemented to fully integrate generated platforms into U-Boot. These special issues are described in “Special Issues Regarding U-Boot” on page 47.

Accessing Parameters in U-Boot Stand-Alone Applications

U-Boot enables you to dynamically load and run stand-alone applications that can use U-Boot functions, such as console I/O functions, memory allocation, or interrupt services. These stand-alone applications also have access to the board configuration of the LatticeMico32 architecture, which is stored in the U-Boot global_data structure.

You can access the global_data structure in the same way that U-Boot accesses it. After adding DECLARE_GLOBAL_DATA_PTR;, the gd variable holds a pointer to the global data gd_t structure. The board configuration is stored in a bd_config_t structure in the global data and can be accessed by gd > bd_config. The global data structure is not initialized until calling app_startup().

The bd_config_t structure is defined as follows:

```
typedef struct bd_config {
    unsigned long   lm32tag_num_cpu;
    LM32Tag_CPU_t   *lm32tag_cpu;
    unsigned long   lm32tag_num_asram;
    LM32Tag_ASRAM_t *lm32tag_asram;
    unsigned long   lm32tag_num_flash;
    LM32Tag_Flash_t *lm32tag_flash;
    unsigned long   lm32tag_num_sdram;
    LM32Tag_SDRAM_t *lm32tag_sdram;
    unsigned long   lm32tag_num_ocm;
    LM32Tag_OCM_t   *lm32tag_ocm;
    unsigned long   lm32tag_num_ddr_sdram;
    LM32Tag_DDR_SDRAM_t *lm32tag_ddr_sdram;
    unsigned long   lm32tag_num_ddr2_sdram;
    LM32Tag_DDR2_SDRAM_t    *lm32tag_ddr2_sdram;
    unsigned long   lm32tag_num_timer;
    LM32Tag_Timer_t *lm32tag_timer;
    unsigned long   lm32tag_num_uart;
    LM32Tag_UART_t  *lm32tag_uart;
    unsigned long   lm32tag_num_gpio;
    LM32Tag_GPIO_t  *lm32tag_gpio;
    unsigned long   lm32tag_num_leds;
    LM32Tag_LEDS_t  *lm32tag_leds;
    unsigned long   lm32tag_num_7seg;
    LM32Tag_7SEG_t  *lm32tag_7seg;
    unsigned long   lm32tag_num_trispeedmac;
    LM32Tag_TriSpeedMAC_t   *lm32tag_trispeedmac;
    unsigned long   lm32tag_num_i2cm;
```

```
    LM32Tag_I2CM_t  *lm32tag_i2cm;
    unsigned long   lm32tag_num_spi_s;
    LM32Tag_SPI_S_t *lm32tag_spi_s;
    unsigned long   lm32tag_num_spi_m;
    LM32Tag_SPI_M_t *lm32tag_spi_m;
} bd_config_t;
```

The unsigned long lm32tag_num_DRIVER variables hold a number of components that are mapped to the driver identified by DRIVER. The LM32Tag_..._t *lm32tag_DRIVER pointers point to the address of the first element of the array holding the hardware information of the certain device. The size of this array is the value of lm32tag_num_DRIVER. A description of the hardware information structures is given in “U-Boot Hardware Setup Module” on page 43.

Here is an example of a U-Boot stand-alone application that accesses and prints the frequency of the CPU and some information about the timers:

```
#include <common.h>
#include <exports.h>

int my_main (int argc, char *argv[])
{
 int i;
 DECLARE_GLOBAL_DATA_PTR;
 bd_config_t *bd_config;

 app_startup(); bd_config = gd->bd_config;

 if (bd_config->lm32tag_num_cpu > 0)
 {
 printf("CPU Frequency: % Hz\n", bd_config-
 >lm32tag_cpu[0].frequency);
 }
 for (i = 0; i < bd_config->lm32tag_num_timer; ++i)
 {
  printf("Timer %i - Base Address: 0x%08x, IRQ: %i, i,
   bd_config->lm32tag_timer[i].addr,
   bd_config->lm32tag_timer[i].irq);
 }
 return (0);
}
```

Special Issues Regarding U-Boot

In addition to the hardware configuration structures, arrays, and counters, the following mechanisms are necessary for the integration of freely generated platforms into U-Boot:

- You can configure the LatticeMico32 processor to include or exclude certain features that have associated machine flags required by GCC to create correct binary code. MSBConfigParser supports the extraction of these machine flags from the .msb file. They are stored in a special variable in the hwsetup_generated.h file and can be extracted for

compilation by the grep and sed tools from the generated board/lattice/.../config.generated.mk makefile.

- The base address and the size of the first component instance of a component type are written to the generated header file, in addition to the structure initializers. They provide component instance-name independent base address and size values that can be used to map preprocessor definitions expected by certain driver modules or the build process. Currently the U-boot CFI flash device driver expects such a flash base address and size mapping for working with the CFI flash device, and the boot module expects such a read and write memory mapping.

Passing Parameters to the Linux Kernel

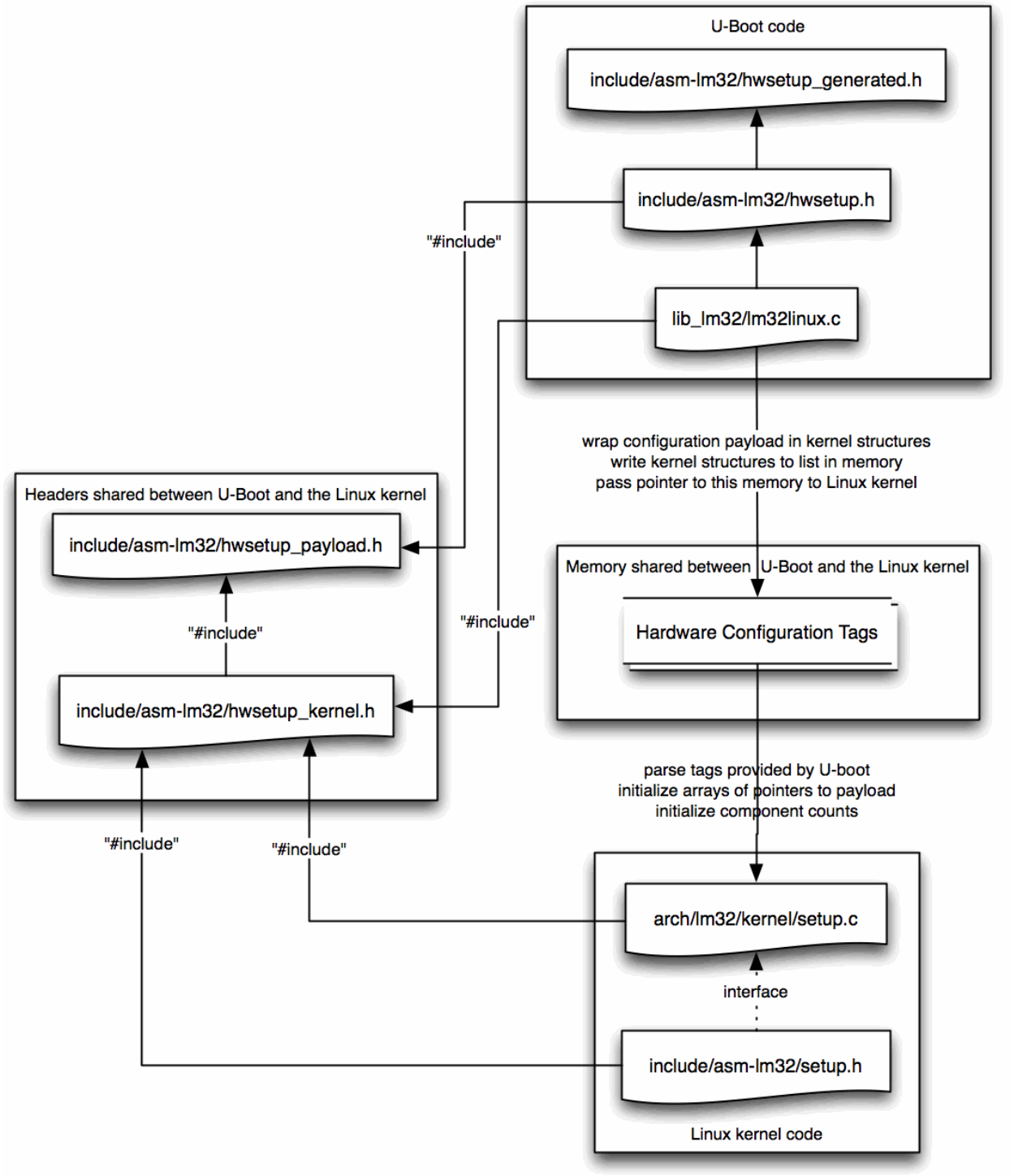
The Linux boot process is as follows: U-Boot wraps the hardware configuration payload structures in other structures called kernel configuration structures. These structures prepend a header structure to the payload. The kernel configuration structures are written to a contiguous region of memory.

The pointer to the memory location of the hardware parameters—that is, the list of structures from hwsetup_kernel.h—is written to register r1, which the Linux kernel reads when it starts and subsequently uses.

Each header contains a size and a tag element. The size is set to the size of the header plus the size of the configuration payload structure. The tag is set to one of the tags defined for driver types. The necessary types and tags, as well as the header, are defined in include/asm-lm32/hwsetup_kernel.h, which includes include/asm-lm32/hwsetup_payload.h. You can find the U-Boot module that creates the configuration list in memory and then boots Linux in lib_lm32/lm32linux.c.

Linux obtains the address of the hardware configuration list and parses it in the setup_arch function in arch/lm32/kernel/setup.c. This file also contains arrays of pointers to payload structures for each driver type and corresponding counters for determining how many pointers refer to valid payload structures and how many devices there are for each driver. U-Boot and the Linux kernel share the header files defining payload and kernel structures for the hardware configuration data. The interface to the hardware configuration data in the Linux kernel is the include/asm-lm32/setup.h header, from which any LatticeMico32-specific kernel code can access the platform configuration similar to the way it accesses the platform configuration from the U-Boot code. Figure 6 shows the code and memory shared between U-Boot and the Linux kernel.

Figure 6: Code and Data Shared Between U-Boot and the Linux Kernel



In addition to the pointer to the hardware parameters, U-Boot passes other pointers to the Linux kernel by using registers. In Table 6 is a list of all

registers used by U-Boot and the Linux kernel to exchange essential configuration data.

Table 6: Passing Kernel Configuration from U-Boot to the Linux Kernel

Register	Pointer to
r1	Hardware parameters; pointer to the list of structures from hwsetup_kernel.h
r2	Kernel command line (\0 terminated)
r3	initrd start
r4	initrd end

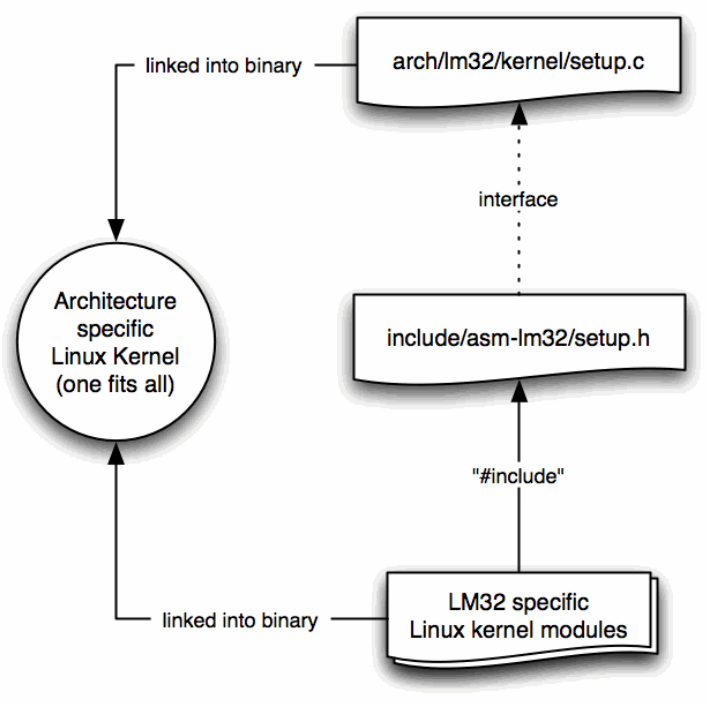
Accessing Parameters in the Linux Kernel

You can access the configuration data from any module in the Linux kernel by including the include/asm-lm32/setup.h header, which provides extern declarations for the symbols in the arch/lm32/kernel/setup.c module:

- One extern unsigned long hs_num_...; – Is the declaration for the number of components mapped to the driver identified by ..., for example, extern unsigned long hs_num_ddr_sdram.
- One extern LM32Tag_..._t* hs_...[]; – Is the declaration for an array of pointers pointing to the contents of the hardware information payload structures, for example, extern LM32Tag_DDR_SDRAM_t* hs_ddr_sdram[];.

Figure 7 shows the structure of the hardware configuration code in the Linux kernel.

Figure 7: Hardware Setup Code Structure in the Linux Kernel



The following code sample shows how to access the hardware configuration from the LatticeMico32-specific Linux code. This example uses the same hardware setup parameters as the example in “Accessing Hardware Setup Parameters in U-Boot” on page 45. There is one notable difference between U-Boot and the Linux kernel access to hardware configuration. The arrays are not arrays of payload structure data types but arrays of pointers to payload structure data types. Therefore, you must use “ ->” instead of “.” to access member elements.

```
/* include the interface to the hardware setup data structures
*/
#include <asm-lm32/setup.h>

/* number of timers: */
u32 num_timers = lm32tag_num_timer

;/* frequency of cpu: */
u32 freq = lm32tag_cpu[0]->frequency;

/* IRQ number of second timer: */
u8 timer2_irq = lm32tag_timer[1]->irq;
```

5

LatticeMico32 LatticeECP2-50-Specific Drivers

This chapter describes the LatticeECP250-specific drivers implemented in the Linux port to LatticeMico32. For each driver, it gives a short implementation description and an example for testing the driver.

LED Driver

The provided LED driver is used to set and unset the dedicated status LEDs on the LatticeECP2-50 evaluation board. Each single LED is addressed separately.

To test the LED driver:

1. First mount sysfs:

   ```
   # cd /# mkdir sys# mount -t sysfs sysfs /sys
   ```

2. Change to the /sys/class/leds/leds-ecp250_<*i*> sysfs directory, where <*i*> must be replaced with the desired LED index.
3. Write to the brightness file to change the LED state. Use 0 to set (turn on) or 1 to unset (turn off) the LED.

Here are some examples:

To set the LED with index 3, use this syntax:

```
# echo -n "0" > /sys/class/leds/leds-ecp250_3/brightness
```

To unset the LED with index 5, use this syntax:

```
# echo -n "1" > /sys/class/leds/leds-ecp250_5/brightness
```

7-Segments Driver

The delivered Linux port to LatticeMico32 includes a driver to control the 7-segment display element of the LatticeECP2-50 evaluation board.

Note

Both segments cannot be set simultaneously. Only the left or the right segment can be activated at a time.

To identify which segment is called the “left segment” and which one is called the “right segment,” it is assumed that the dot next to the 7-segment element indicates the bottom of the segment.

The left segment takes precedence over the right one. If you try to set both segments, only the left one is set.

To test the 7-segments driver:

- Write a four-byte-long string to /dev/ecp250_7seg_0.

  The first two bytes determine the state of the left segment, and the latter two bytes determine the state of the right segment. Each two-byte instruction must conform to the following grammar (Perl regular expression syntax):

  ```
  Instruction  :=  [ 0-9][ \.]
  ```

  The first byte sets the value of the segment, where a space turns the segment off, and the second byte sets or unsets the dot of the corresponding segment.

  You must not write a new line at the end of the sequence to successfully set the state.

Some examples follow:

- The left segment is set to “7.,” and the right segment is turned off:

  ```
  echo -n "7.  " >/dev/ecp250_7seg_0
  ```

  The -n parameter prevents a new line (`'\n'`) at the end of the string.

- The left segment is turned off, and the right segment is set to “4” without a dot:

  ```
  echo -n "  4 " >/dev/ecp250_7seg_0
  ```

Flash Driver

By default, the partitions of the flash device of the LatticeECP2-50-based board are laid out as shown in Table 7.

Table 7: Flash Device Layout

Address Range	Purpose
0x00000000 - 0x0003ffff	U-Boot
0x00040000 - 0x0023ffff	Linux kernel
0x00240000 - 0x01fbffff	Free space
0x01fc0000 - 0x01ffffff	U-Boot environment

You can change the start and end addresses of the partitions in the linux-2.6.x/drivers/mtd/maps/ecp250_flash.c file in the struct mtd_partition ecp250_partitions array:

```
static struct mtd_partition ecp250_partitions[] = {
    {
        .name = "U-Boot",
        .size = 0x00040000, /* 256 kB */
        .offset = 0x0000000
    },
    {
        .name = "Linux Kernel",
        .size = 0x00200000, /* 2 MB */
        .offset = MTDPART_OFS_APPEND,
    },
    {
        .name = "Free",
        .size = (0x02000000 - 0x40000 - 0x40000 - 0x200000),
    /* rest to play with */
        .offset = MTDPART_OFS_APPEND,
    },
    {
        .name = "U-Boot Env",
        .size = 0x40000,
    /* last 256 kB is U-Boot environment */
        .offset = MTDPART_OFS_APPEND,
    }
};
```

The MTDPART_OFS_APPEND preprocessor macro causes the offset—that is, the start address—of the partition to be set to the address (offset + size)—that is, the end address + 0x1—of the previous partition.

GPIO Driver

The GPIO driver implements a character device for reading from and writing to GPIO devices. Each device has one major number and many minor numbers. Each minor number represents one bit per pin of the device.

After you open a device for writing, the commands shown in Table 8 are available.

Table 8: Flash Device Layout

Command	Description
“0”	Writes a logical zero to the output/tristate port.
“1”	Writes a logical one to output/tristate port.
“T”	Sets the tristate port to output.
“t”	Sets the tristate port to input.

Write a command to the character device to trigger the expected behavior.

Reading from the device returns either the last written bit (0 or 1), if the tristate port is set to output (“T”), or the currently active input level (0 or 1), if the tristate port is set to input (“t”).

A

The hwsetup_payload.h File

This appendix gives an example of a hwsetup_payload.h file, which defines the setup structures generated automatically by MSBConfigParser.

```
/*
* (C) Copyright 2008
* Theobroma Systems  <www.theobroma-systems.com>
*
* See file CREDITS for list of people who contributed to this
* project.
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
* as published by the Free Software Foundation; either version
* 2 of the License or (at your option) any later version.
*
* This program is distributed in the hope that it will be
* useful but WITHOUT ANY WARRANTY and without even the implied
* warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
* PURPOSE. See the GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public
* License along with this program; if not, write to the Free
* Software Foundation, Inc., 59 Temple Place, Suite 330,
* Boston, MA 02111-1307 USA
*/
#ifndef _ASM_LM32_HARDWARESETUP_H
#define _ASM_LM32_HARDWARESETUP_H

#include <linux/types.h>

/*
* This file defines the LM32 Hardware Setup Payload structs.
*
* The actual configuration is done in hwsetup.c, which uses the
* generated header file hwsetup_generated.h, which initializes
* the payload structs. This header is generated by the
```

```
* MSBConfigParser tool.
*/

#define LM32TAG_MAX_IDENTIFIER_LENGTH 32

/*
* CPU
*/
typedef struct LM32Tag_CPU
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* CPU Frequency */
  u32 frequency;
} LM32Tag_CPU_t;

/*
 * Asynchronous SRAM
 */
typedef struct LM32Tag_ASRAM
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* Base Address */
  u32 addr;
  /* Memory Size */
  u32 size;
  /* Read Latency */
  u8 read_latency;
  /* Write Latency */
  u8 write_latency;
  /* Address Width */
  u8 address_width;
  /* Data Width */
  u8 data_width;
} LM32Tag_ASRAM_t;

/*
 * Flash Memory
 */
typedef struct LM32Tag_ASRAM LM32Tag_Flash_t;

/*
 * SDRAM
 */typedef struct LM32Tag_SDRAM
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* Base Address */
  u32 addr;
  /* Memory Size */
  u32 size;
} LM32Tag_SDRAM_t;

/*
 * On-Chip Memory
 */
typedef struct LM32Tag_SDRAM LM32Tag_OCM_t;
```

```
/*
 * DDR SDRAM
 */
typedef struct LM32Tag_SDRAM LM32Tag_DDR_SDRAM_t;

/*
 * DDR2 SDRAM
 */
typedef struct LM32Tag_SDRAM LM32Tag_DDR2_SDRAM_t;

/*
 * Timer
 */
typedef struct LM32Tag_Timer
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* Base Address */
  u32 addr;
  /* Writeable Tick Count (Flag) */
  u8 wr_tickcount;
  /* Readable Tick Count (Flag) */
  u8 rd_tickcount;
  /* Start Stop Control (Flag) */
  u8 start_stop_control;
  /* Counter Width (Bits) */
  u8 counter_width;
  /* Default Reload Tick */
  u32 reload_ticks; /* IRQ */
  u8 irq;
  /* reserved and for alignment */
  u8 reserved0;
  /* reserved and for alignment */
  u8 reserved1;
  /* reserved and for alignment */
  u8 reserved2;
} LM32Tag_Timer_t;

/*
 * UART
 */
typedef struct LM32Tag_UART
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* Base Address */
  u32 addr;
  /* Baud Rate */
  u32 baudrate;
  /* Data Bits */
  u8 databits;
  /* Stop Bits */
  u8 stopbits;
  u8 use_interrupt;
  /* Block on Transmit (Flag) */
  u8 block_on_transmit;
  * Block on Receive (Flag) */
  u8 block_on_receive;
  /* Rx Buffer Size */
```

```
  u8 rx_buffer_size;
  /* Tx Buffer Size */
  u8 tx_buffer_size;
  /* IRQ */
  u8 irq;
} LM32Tag_UART_t;

/*
 * GPIO Constants
 */
enum {
 /*
  * GPIO Port Types
  */

 /* Port Type = output: width is stored in data_width */

 LM32TAG_GPIO_PORT_OUTPUT = 0x1,
 /* Port Type = input: width is stored in data_width */

 LM32TAG_GPIO_PORT_INPUT = 0x2,
 /* Port Type = tristate: width is stored in data_width */
 LM32TAG_GPIO_PORT_TRISTATE = 0x3,
 /* Port Type = input and output: the width is
  * stored in input_width and output_width */
 LM32TAG_GPIO_PORT_INPUTOUTPUT = 0x4,

 /*
  * GPIO IRQ Modes
  */

 /* IRQ Mode = level: edge_response field unused */
 LM32TAG_GPIO_IRQMODE_LEVEL = 0x5,
 /* IRQ Mode = edge: edge_response field determines edge mode
 */ LM32TAG_GPIO_IRQMODE_EDGE = 0x6,

 /*
  * GPIO Edge Responses
  */

 /* Edge Response = Positive Edge */
 LM32TAG_GPIO_EDGERESPONSE_POS = 0x7,
 /* Edge Response = Negative Edge */
 LM32TAG_GPIO_EDGERESPONSE_NEG = 0x8,
 /* Edge Response = Either Edge = Both Edges */
 LM32TAG_GPIO_EDGERESPONSE_BOTH = 0x9
};

typedef struct LM32Tag_GPIO
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* Base Address */
  u32 addr;
  /* Port Types: one of LM32TAG_GPIO_PORT_... */
  u32 port_types;
  /* IRQ Mode: one of LM32TAG_GPIO_IRQMODE_... */
  u32 irqmode;
  /* IRQ Mode Edge Response: one of
```

```
  * LM32TAG_GPIO_EDGERESPONSE_... */
  u32 edge_response;
  /* Data Width */
  u8 width_data;
  /* Input Width */
  u8 width_input;
  /* Output Width */
  u8 width_output;
  /* IRQ Mode in use (Flag) */
  u8 irq;
} LM32Tag_GPIO_t;

/*
 * Leds on GPIO
 */
typedef struct LM32Tag_GPIO LM32Tag_LEDS_t;

/*
 * 7-Segment Display on GPIO
 */
typedef struct LM32Tag_GPIO LM32Tag_7SEG_t;

 /*
 * Tri-Speed Ethernet MAC
 */
typedef struct LM32Tag_TriSpeedMAC
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* Base Address */
  u32 addr;
  /* Tx/Rx FIFO Depth */
  u32 fifo_depth;
  /* Use MDIO Interface (Flag) */
  /* Include Statistics registers (Flag) */
  u8 stat_regs_present;
  /*
  irq number */
  u8 irq;
  * reserved and for alignment */
  u8 reserved0;
} LM32Tag_TriSpeedMAC_t;

/*
 * I2CM (I2C Master Controller)
 */
typedef struct LM32Tag_I2CM
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* Base Address */
  u32 addr;
  /* SCL Speed */
  u32 scl_speed;
} LM32Tag_I2CM_t;

/*
 * SPI master or slave
 * Only the typedefs of this struct will be used directly.
```

```
 */
*/struct LM32Tag_SPI
{
  /* Instance Name */
  char name[LM32TAG_MAX_IDENTIFIER_LENGTH];
  /* Base Address */
  u32 addr;
  /* data length */
  u8 data_length;
  /* shift direction (0 = msb first, 1 = lsb first) */
  u8 shift_direction;
  /* phase (0 = leading edge, 1 = trailing edge) */
  u8 phase;
  /* polarity (0 = idle low, 1 = idle high) */
  u8 polarity;
  /* master (0 = slave, 1 = master) */
  u8 master;
  /* number of slaves (only valid for master) */
  u8 num_slaves;
  /* sclk rate (only valid for master) */
  u8 sclk_rate;
  /* tx start delay (only valid for master) */
  u8 tx_start_delay;
  /* clock counter width (only valid for master) */
  u8 clock_counter_width;
  /* tx interframe pause (only valid for master) */
  u8 tx_interframe_pause;
  /* reserved and for alignment */
  u8 reserved0;
  /* reserved and for alignment */
  u8 reserved1;
};

/*
 * SPI Slave
 */
typedef struct LM32Tag_SPI LM32Tag_SPI_S_t;

/*
 * SPI Master
 */
typedef struct LM32Tag_SPI LM32Tag_SPI_M_t;

/*
 * Board configuration
 */
typedef struct bd_config {
 unsigned long lm32tag_num_cpu;
 LM32Tag_CPU_t *lm32tag_cpu;
 unsigned long lm32tag_num_asram;
 LM32Tag_ASRAM_t *lm32tag_asram;
 unsigned long lm32tag_num_flash;
 LM32Tag_Flash_t *lm32tag_flash;
 unsigned long lm32tag_num_sdram;
 LM32Tag_SDRAM_t *lm32tag_sdram;
 unsigned long lm32tag_num_ocm;
LM32Tag_OCM_t *lm32tag_ocm;
 unsigned long lm32tag_num_ddr_sdram;
 LM32Tag_DDR_SDRAM_t *lm32tag_ddr_sdram;
```

```
 unsigned long lm32tag_num_ddr2_sdram;
 LM32Tag_DDR2_SDRAM_t *lm32tag_ddr2_sdram;
 unsigned long lm32tag_num_timer;
 LM32Tag_Timer_t *lm32tag_timer;
 unsigned long lm32tag_num_uart;
 LM32Tag_UART_t *lm32tag_uart;
 unsigned long lm32tag_num_gpio;
 LM32Tag_GPIO_t *lm32tag_gpio;
 unsigned long lm32tag_num_leds;
 LM32Tag_LEDS_t *lm32tag_leds;
 unsigned long lm32tag_num_7seg;
 LM32Tag_7SEG_t *lm32tag_7seg;
 unsigned long lm32tag_num_trispeedmac;
 LM32Tag_TriSpeedMAC_t *lm32tag_trispeedmac;
 unsigned long lm32tag_num_i2cm;
 LM32Tag_I2CM_t *lm32tag_i2cm;
 unsigned long lm32tag_num_spi_s;
 LM32Tag_SPI_S_t *lm32tag_spi_s;
 unsigned long lm32tag_num_spi_m;
 LM32Tag_SPI_M_t *lm32tag_spi_m;
} bd_config_t;

#endif
```

B

MSBConfigParser Document Type Definitions

This appendix describes the document type definitions, or syntax and rules, that MSBConfigParser uses.

Driver Description File (driver_structs.dtd)

Following is an example of the driver description file, driver_structs.dtd, which contains the document type definitions for driver structures.

```
<?xml version='1.0' encoding='utf-8'?>

<!-- DTD for the driver description file (driver_structs.xml)
-->

<!ELEMENT drivers (driver+) >
<!ATTLIST drivers
    nameMaxLength    CDATA    "32"
>

<!ELEMENT driver (struct?) >
<!ATTLIST driver
    name                 CDATA              #REQUIRED
    type                 (CPU|Memory|IO)    "IO"
    abstract             (true|false)       "false"
    extends              CDATA              #IMPLIED
    baseAddressSource    CDATA              "BASE_ADDRESS"
    sizeSource           CDATA              "SIZE"
>

<!ELEMENT struct (member+) >

<!ELEMENT member (condition*, or*) >
<!ATTLIST member
    type               (u8 |
                        u32 |
```

```
                        char |
                        string |
                        enum |
                         bool |
                         name |
                         irq |
                         cpufreq)    #REQUIRED
     name             CDATA            "unknown"
     source           CDATA            #IMPLIED
     default          CDATA            #IMPLIED
     value            CDATA            #IMPLIED
     condition        CDATA            #IMPLIED
     notCondition     CDATA            #IMPLIED
>

<!ELEMENT condition EMPTY >
<!ATTLIST condition
    isTrue           CDATA        #IMPLIED
    isFalse          CDATA        #IMPLIED
>

<!ELEMENT or (condition*) >
<!ATTLIST or
     value            CDATA        #IMPLIED
     condition        CDATA        #IMPLIED
     notCondition     CDATA        #IMPLIED
>
```

Compiler Flags Description File (compiler_flags.dtd)

Following is an example of the compiler flags description file, compiler_flags.dtd, which contains the document type definitions for the compiler flags.

```
<?xml version='1.0' encoding='utf-8'?>

<!-- DTD for the compiler flag description file
(compiler_flags.xml) -->

<!ELEMENT flags (flag+) >

<!ELEMENT flag EMPTY >
<!ATTLIST flag
    flag             CDATA               #REQUIRED
    driver           (CPU|Memory|IO)     "CPU"
    source           CDATA               #IMPLIED
    include_if       CDATA               #IMPLIED
    exclude_if       CDATA               #IMPLIED
>
```

Index