LPC32x0 startup code

Table of contents

List of tables

1 Introduction

This document provides important information on developing and using the NXP provided startup code and boot loaders for the LPC32x0 processors. The code provided with this package is intended to provide a starting point for LPC32x0 system developers to get their products up and running fast.

This document also explains the topic of boot loaders and the LPC32x0 boot process. Included with the startup code are several different variants of boot loaders that allow booting from different boot device types. Reference software to burn the boot loader image into FLASH or download the boot loader directly into memory is also provided with this package.

Overall, this package provides the following features:

- Configurable and scalable startup code
  - Can optionally setup a basic MMU page table
  - Can optionally setup the board functions (clocking, SDRAM, GPIOs)
  - Full configurable reference code for standard and mobile single and double data rate SDRAM initialization
  - Can optionally set up runtime stacks
- Reference kickstart loader
  - Works with the startup code
  - Provides initial boot capability for the LPC32x0 (boots stage 1 application)
  - Works with small and large block NAND and SPI FLASH and EEPROMs
- Reference burner software
  - Works with the startup code
  - Provides an easy and tool-free method for burning the kickstart and stage 1 applications
  - Works with the serial loader tool provided in this package (requires UART5 to be available on system reset)
  - Can burn images to small and large block NAND FLASH, SPI FLASH and EEPROMs, and NOR FLASH
  - Includes reference drivers for small and large block NAND SLC and MLC controllers, NOR FLASH, and SPI FLASH
- Stage 1 Loader (S1L)
  - Works with the startup code
  - A generic version of S1L is provided with this package can be loaded and used by the kickstart loader or booted directly from NOR FLASH
  - Works with the serial loader tool provided in this package (requires UART5 to be available on system reset) – an image can be up and running on a new board in less than an hour

  - Provides secondary image loading capability
  - Provides some test capability for SDRAM or other interfaces
- Support for various toolchains
  - All code builds with ARM Realview 3.x, Keil MDK 4.0, or CodeSourcery GNU
  - Make file based environment allows same build environment for all toolchains

The code is free to use for NXP processors and can be integrated into customer designs.

1.1 Overview of this document

This document covers various aspects of the LPC32x0 boot process, startup code, and boot loaders. It is intended to help with rapid beingup of new boards based on the LPC32x0 processors.

The document is broken up into the following sections:

LPC32x0 boot process
Read this section to understand how the LPC32x0 boot ROM loads and runs an image and the restrictions for image loading from the various non-volatile device types. This section should be read before jumping into the section on boot loaders.

Startup and board code
This section covers the startup code and board drivers included in this package. This section covers code configuration, SDRAM setup, and other aspects of bringing up a new system. The startup code is used as the basis for the other applications in this package – including the kickstart, Stage 1 Loader, and the burner software. The board drivers are used by the burner software, kickstart loader, and Stage 1 Loader. For new boards, these drivers may need to be edited or tweaked.

Applications – boot loaders, burner software, and stage 1 applications
This section covers applications based on the startup and board the can be booted with the LPC32x0 from NAND FLASH, NOR FLASH, SPI FLASH, or via the UART. Read this section to learn the concept of the kickstart and stage 1 loader applications.

Burner software applications are applications that help during initial board bringup. Once your board is running the startup code and application, you can use the burner applications to burn your boot image(s) into the boot device on the board. The burner software works with the Serial Loader Tool and doesn't require any special hardware (such as a JTAG interface).

Serial Loader tool

The Serial Loader tool is used with the LPC32x0 on reset to provide a UART based boot capability. An image can be built that loads into Internal RAM (IRAM) on system reset via the UART. This helps save time during early board debug trying to get a full featured boot loader up and running with non-volatile boot support.

Stage 1 Loader (S1L)
Read this section for an understanding of the Stage 1 Loader. The Stage 1 Loader is a stage 1 application that provides various features for new board testing or launching user applications. S1L is included as the default boot loader on Phytec and Embedded Artists 3250 boards.

Build process and examples
The section explains how to build and deploy the software. Included examples show how to build he kickstart loader and S1L with SDRAM support using the free CodeSourcery GNU toolchain. The image is then deployed using the Serial Loader tool and SDRAM is tested. For new designs, this is an ideal way to initially get SDRAM up and running.

Sizing information for various configurations is also provided.

1.2 Stage 1 loader

This package includes a generic version of the Stage 1 Loader (S1L) that can be easy ported to a new LPC32x0 based board or used for early board debug. S1L is a small program that usually runs in IRAM and provides a monitor program with a collection of functions to help with debug and application development. The main features of the stage 1 loader are shown below:

- Register and memory change and dump
  - Poke, peek, dump, fill
- Image load via a serial port , SDMMC card, or FLASH
  - Supports raw binary and S-record files
  - Images can be executed after loading
  - Images can be saved in NAND FLASH
- NAND FLASH support
  - Erase of NAND blocks
  - Direct read and write of FLASH blocks and pages
  - Bad block management
- MMU functions
  - Data and instruction cache control
  - Virtual address translation enable/disable
  - Virtual address remapping
  - Page table dump
- System support functions
  - Baud rate control, clock control, system information
- Automatic load and run support

  - Automatic load and execution of images from NAND FLASH, SDMMC, or via the terminal
- Testing functions
  - SDRAM memory tests, bandwidth tests
  - SDRAM calibration and configuration data

The source code for the kickstart loader is provided free of charge by NXP for use with NXP processors. It is available on NXP's website in the LPC32x0 CDL.

2 LPC32x0 boot process

This section explains the LPC32x0 boot process for the various non-volatile memory types or the UART.

Depending on the boot device, there may be limitations to the boot image size. This limitation may prevent a desired image from initially loading and executing – instead requiring a smaller boot loader to be loaded first and then using it to load and run a more capable boot loader or application.

2.1 LPC32x0 boot options

The LPC32x0 supports booting from small and large block NAND FLASH, SPI EEPROMS or FLASH, NOR FLASH, and via the UART. In production systems, the UART boot method isn't used. However, the UART boot method serves a useful purpose in early board debug,

The boot order for the LPC32x0 is as follows: UART, SPI, NOR, NAND. UART boot can be disabled to reduce boot time. Each boot device will be tried prior to the next device. A device is bootable if the boot ROM locates a special boot pattern that identifies the device as bootable. It's possible that a system can have SPI FLASH, NOR FLASH, and NAND FLASH and only boot from NAND FLASH although the SPI and NOR methods are attempted first. (For example, the Phytec 3250 board has all 3 of these boot devices, but normally boots from NAND FLASH).

When the LCP32x0 chip is reset, the internal ROM is executed. It queries each device looking for a special boot pattern used to identify the device as bootable (or that it exists). The first device that returns a correct pattern will be used by the boot ROM as the bootable device. The boot ROM will then either copy data from the bootable device (NAND and SPI FLASH) or transfer control to the bootable device (NOR FLASH).

Each boot method is explained in the following sections. See the LPC32x0 User’s Guide for more information on the LPC32x0 boot process.

2.1.1 UART boot

The UART boot option allows a binary to be downloaded into the board and then executed prior to any code running in NAND, SPI, or NOR FLASH. This is useful for burning new images into FLASH, testing simple applications, or recovering the board from some bad boot code.

UART boot is used with the burner tools in this package to program the bootloader(s) into the boot device. UART5 is supported with the UART boot option, so new designs should make UART5 available for debugging. For a system that entirely runs from IRAM, this allows a system to download an image via the UART that provides interactive access once the image is executed. The downloaded interactive program can

be used for bandwidth or memory tests, register dumps, further image loading, or other functions.

Initial versions of S1L can be used with this boot method to test different SDRAM configurations using the included SDRAM initialization code. Using this approach, startup and debug of the SDRAM code (one of the tougher issues of bringing up a new board) can be drastically simplified.

2.1.2 NOR FLASH boot

On a system that boots from NOR FLASH, the ARM core directly executes code from the NOR FLASH device starting at address 0xE0000004. The first 4 bytes at address 0xE0000000 are used for the special boot pattern value required by the boot ROM to identify the device as bootable and to provide the boot width (8, 16, or 32 bits).

2.1.3 SPI FLASH/EEPROM boot

On a system that boots from SPI, a small image is copied from the SPI EEPROM or FLASH device at offset 0x8 into IRAM at address 0x00000000. The first 4 bytes are used for the special boot pattern value required by the boot ROM. The next 4 bytes are used to designate the size of the image to load at offset 0x8 in the SPI device. After the image is loaded into IRAM, control is transferred to address 0x00000000.

The maximum image boot size that can be loaded with this method is 54K.

2.1.4 NAND FLASH boot

On a system that boots from NAND FLASH, a small image is copied from block 0 or block 1 of the NAND device into IRAM at address 0x00000000. The maximum bootable image size for NAND FLASH boot is 54K, or 1 block minus 1 page, whichever is smaller. The first page of the boot block is used to store boot information, such as the boot image size and values needed by the boot ROM used for the boot sequence.

For small block NAND (32 pages per block and 512 bytes per page) with the first page of the block dedicated to the boot ROM boot information, the maximum boot size is 31 pages of 512 bytes per page, or 15.5K.For large block NAND (64 pages of 2Kbytes per page), the maximum boot size is 54Kbytes.

The boot ROM always uses the MLC NAND controller for NAND boot. This requires the boot image booted from NAND by the boot ROM to be programmed into NAND with the MLC NAND controller or with the ECC algorithms required by the MLC NAND controller.

3 Startup and board code

3.1 Startup code

The configurable startup code provided with this package provides basic board initialization for the LPC32x0 after control is passed from the boot ROM to the image execution address.

The startup code is designed to be small and flexible and can be used to initially place the board into a known state and then launch another application. The kickstart boot loader, S1L, and burner software all use this same startup code with slightly different options.

The startup code provides the following features:

Configurable and scalable to keep image size small
A small image size is important when the boot image size requirement is small (for example, 15.5K boot image size maximum from small block NAND FLASH)

Optionally sets up a basic MMU page table
If caches and MMU support is needed, a basic section table is created in IRAM. To use data cache and get optimal performance from the LPC32x0, the MMU needs to be setup. For OSes like Linux and WinCE, this doesn't have to be setup by the startup code and can be removed to save space.

Can optionally setup the board functions (clocking, SDRAM, GPIOs)
Board and chip specific functions such as GPIO states and muxing, system clock rates, memory timings, and SDRAM setup usually vary by board and should be setup early. SDRAM in particular must be setup before the memory range offered by SDRAM can be used. The board functions can be optionally disabled to save space in the startup code. If they are disabled, they are usually required elsewhere.

Full configurable reference code for standard and mobile single and double data rate SDRAM initialization
Although this is part of the board setup code, full reference code based on the JEDEC SDRAM initialization sequence is provided in this package. In most cases, only the SDRAM initialization definitions need to be changed to get SDRAM working on your system. The SDRAM initialization code is included as part of the board function code.

Runtime stack setup
All the ARM core stacks can be setup in the startup code if they are needed in the application kicked off by the startup code. This can be removed to save some space. If removed, only the SVC mode stack is initialized.

Data relocation
On systems that use NOR FLASH as the boot device, the code executes directly from read-only NOR FLASH, but the data needs to be relocated to a writable area. Optional data segment relocation code is included in the startup code for use with NOR FLASH systems that relocates the data segment to IRAM.

Call to a user's application
The user application is called from the startup code via a call to e_entry(). Although stacks are setup for the called function, a full C/C++ runtime environment is not setup. If a full runtime environment is needed, this needs to be setup by the user application.

3.1.1 Startup code configuration

Selection of the startup code's capabilities depends on the configuration of the startup code as it's compiled. Compilation options are selected by changing definition values in the startup header files or altering the definitions used with the compiler parameters as the startup code is compiled.

3.1.1.1 Startup code definitions in header files

Definition values in header files are usually set once per board design and not changed again. For the startup code, the definitions used to define startup operation are located in the setup.h, setup.inc, and setup_gnu.inc files in the ./startup directory. Before building any code, the values in these files may need to be changed.

3.1.1.1.1 setup.h definitions

The following definitions are defined in the setup.h header file. Detailed comments for each definition are in the header file. These need to be changed as necessary for new designs.

Commenting or un-commenting the following line may affect the final image size. Un-commenting the definition will allow extra code used for debug to be included in the system images.

```
#define ENABLE_DEBUG
```

For all devices except the LPC3220, this should not be changed from the 0x40000 value. For LPC3220 devices, this value should be changed to 0x20000.

```
#define IRAM_SIZE <size of IRAM in bytes>
```

Default CPU speed in Hz. The startup code will setup the system PLL to this rate.

```
#define CPU_CLOCK_RATE <Value in Hz>
```

Default bus clock divider. The startup code will setup the bus divider to this value to generate the bus clock rate.

```
#define HCLK_DIVIDER <value>
```

Default peripheral clock divider. The startup code will setup the peripheral divider to this value to generate the peripheral clock rate.

```
#define PCLK_DIVIDER <value>
```

SDRAM type definition: Depending on this value, one of the 4 variants of SDRAM initialization will be used in the startup code. See Section 3.1.1.2 for more information on SDRAM setup.

```
#define SDRAM_OPTION <SDRAM system definition>
```

Un-commenting out the following line will add some extra code to initialize SDRAM devices on the second SDRAM chip select (DYSC1). If SDRAM devices are only on chip select DYCS0, then this define should be commented out.

```
#define USE_DUAL_SDRAM_DEVICES
```

3.1.1.1.2 setup.inc and setup_gnu.inc definitions

The setup.inc and setup_gnu.inc files are used for configuration in the assembly files. The _gnu version of the file is used only for the CodeSourcery GNU toolchain.

The following definitions are defined in the setup.inc and setup_gnu.inc header files. Detailed comments for each definition are in the header file. These need to be changed as necessary for new designs.

The value of the IRAM_SIZE definition should match the value in the setup.h file.

```
IRAM_SIZE EQU 0x40000 (setup.inc)
.EQU IRAM_SIZE,              0x40000 (setup_gnu.inc)
```

These definitions are used to define the stack sizes setup by the startup code. The stacks are setup in IRAM for the startup code, so if large stacks are needed, the startup code must be changed to allow placing the stacks in SDRAM, or the application needs to setup it's own stacks.

```
FIQ_STACK_SIZE     EQU    64 (setup.inc)
IRQ_STACK_SIZE     EQU    512 (setup.inc)
ABORT_STACK_SIZE   EQU    64 (setup.inc)
UNDEF_STACK_SIZE   EQU    64 (setup.inc)
SYSTEM_STACK_SIZE  EQU    64 (setup.inc)
SVC_STACK_SIZE     EQU    2048 (setup.inc)
.EQU FIQ_STACK_SIZE,          64 (setup_gnu.inc)
.EQU IRQ_STACK_SIZE,          512 (setup_gnu.inc)
.EQU ABORT_STACK_SIZE,        64 (setup_gnu.inc)
.EQU UNDEF_STACK_SIZE,        64 (setup_gnu.inc)
.EQU SYSTEM_STACK_SIZE,       64 (setup_gnu.inc)
.EQU SVC_STACK_SIZE,          2048 (setup_gnu.inc)
```

3.1.1.2 SDRAM initialization code overview and configuration

Setting up SDRAM is one of the more complex tasks of bringing up a new design. To help reduce the complexity of this task, the SDRAM initialization code is broken into 3 device specific sequences that support all 4 variants of SDRAM that the LPC32x0 can support. This includes low power (mobile) and standard Single Data Rate (SDR) SDRAM and low power and standard Double Data Rate (DDR) SDRAM. The standard initialization sequences are based on the JEDEC SDRAM programming sequence and should work for different devices without changing the code.

To setup the SDRAM code for a specific board, only the definitions for the SDRAM device are needed to be setup in the dram_configs.h header file (in addition to the SDRAM type select in setup.h). The dram_config.h has 4 sections; 1 section for each variant of supported SDRAM type. Only the section that applies to your SDRAM type needs to be edited.

Note that the SDRAM initialization code and SDRAM definitions are designed to shield developers from the more tedious and detail oriented work on setting up SDRAM such as mode word offset computation, clock to timing value computation, general register definitions, or clock sequencing.

These parameters are explained in detailed comments in the dram_configs.h header file. A sample section is shown below.

```
/* DDR refresh interval in inverse time (1 / tREFI). This applies to
the
   EMC DynamicRefresh register.
   Example: For a 16uS refresh cycle, this value will be
   1/16uS = 62500 */
#define SDRAM_RFSH_INTERVAL 128205

/* Number of rows, columns, and bank bits for the SDRAM device */
#define SDRAM_COLS 10
#define SDRAM_ROWS 13
#define SDRAM_BANK_BITS 2 /* 2 bits = 4 banks, 1 bit = 2 banks */

/* If using DDR or SDR with only a 16-bit configuration, set this
   value to 0. For 32-bit SDR SDRAM, set this to 1 */
#define SDRAM_32BIT_BUS 0

/* SDRAM can be configured for low power (Bank-Row-Col (BRC)) mode or
   performance (Row-Bank-Col (RBC)) mode. Performance mode uses mores
   power, but is faster. Set the following define to 0 for low power
   mode, or 1 for performance mode */
#define SDRAM_USE_PERFORMANCE_MODE 1

/* RAS and CAS clock latencies, CAS latencies are in 1/2 clocks,
   while RAS latencies are in full clocks. So a RAS latency of
   2 and a CAS latency of 2-1/2 would be RAS=2 and CAS=5 */
#define SDRAM_RAS_LATENCY 2
```

```
#define SDRAM_CAS_LATENCY 6

/* (tRP) Precharge command period - this can be defined in terms of
   clocks or inverse time (1 / t) */
#define SDRAM_TRP_DELAY 66666666

/* (tRAS) Dynamic Memory Active to Precharge Command period - this
   can be defined in terms of clocks or inverse time (1 / t) */
#define SDRAM_TRAS_DELAY 66666666

/* (tSREX) Dynamic Memory Self-refresh Exit Time - this can be defined
   in terms of clocks or inverse time (1 / t) */
#define SDRAM_TSREX_TIME 33250000

/* (tWR) Dynamic Memory Write Recovery Time - this can be defined in
   terms of clocks or inverse time (1 / t) */
#define SDRAM_TWR_TIME 66666666

/* (tRC) Dynamic Memory Active To Active Command Period - this can be
   defined in terms of clocks or inverse time (1 / t) */
#define SDRAM_TRC_TIME 18181818

/* (tRFC) Dynamic Memory Auto-refresh Period - this can be defined in
   terms of clocks or inverse time (1 / t) */
#define SDRAM_TRFC_TIME 14285714

/* (tXSNR or tXSR) Dynamic Memory Exit Self-refresh - this can be
   defined in terms of clocks or inverse time (1 / t) */
#define SDRAM_TXSNR_TIME 8333333

/* (tRRD) Dynamic Memory Active Bank A to Active Bank B Time - this
   can be defined in terms of clocks or inverse time (1 / t) */
#define SDRAM_TRRD_TIME 100000000

/* (tMRD) Dynamic Memory Load Mode Register To Active Command
   Time - this can be defined in terms of clocks or inverse time
   (1 / t) */
#define SDRAM_TMRD_TIME 2

/* (tCDLR) Dynamic Memory Last Data In to Read Command Time - this can
   be defined in terms of clocks or inverse time (1 / t) */
#define SDRAM_TCDLR_TIME 2

/* DDR mode word. This value defines how the SDRAM device is
configured.
   See the SDRAM data sheet for info on this value. The software will
   place the mode word on the right pins.
   CAS 2, burst of 2, sequential */
#define SDRAM_MODE_WORD 0x31

/* DDR extended mode word. This value defines how the SDRAM device is
   configured. See the SDRAM data sheet for info on this value. The
   software will place the mode word on the right pins. When defining
   this value, do not define it with the BA0,1 signal states.
```

```
   Normal driver strength, DLL enabled */
#define SDRAM_EXT_MODE_WORD 0x00

/* Extended mode word write mask for banks. The extended mode word is
   usually written with the BA1 bank bit low and BA0 high. The
   following define specifies that mapping. */
#define SDRAM_EXT_MODE_BB sdram_get_bankmask(0, 1)

/* Set slew rates to fast by commenting out the following line, or
   set it to slow by uncommenting it. */
/*
#define DDR_USE_SLOW_SLEW
*/
```

3.1.2 Startup code build time configuration

There are several definitions used inside the startup code that conditionally include or exclude specific areas of functionality such as MMU and cache setup or board initialization. These definitions are intended to be passed to the compiler through compiler arguments. By compiling the code with these specific definitions enabled or disabled in the compiler argument list, the functionality and size of the code can be altered. This is important when scaling the size of the boot image when the image size must not exceed the LPC32x0 boot size limit.

The following definitions are defined in the startup code. To alter the inclusion of the code associated with these definitions, the makefile used to build the startup code must be altered. This will explained in more detail in the section on building the code.

`USE_MMU`
If the *USE_MMU* definition is defined, code to setup the MMU page table and enable the MMU will be included in the image. If this is not defined, the instruction and data caches will not be enabled. The MMU page table is located in the last 16K of IRAM. Even if this definition is not enabled, the space for the MMU page table is always reserved.

`RW_RELOC`
If the *RW_RELOC* definition is defined, code will be included in the image to relocate the data segment to a writable area. This is only needed for systems that boot from NOR FLASH where the code is running in read-only NOR FLASH, but data needs to be placed in read-write IRAM. Adding the code for other systems will only increase the image size.

`USE_BOARD_INIT`
If the *USE_BOARD_INIT* definition is defined, the code to setup GPIO and muxing, CPU and bus clocking, memory and SDRAM is included in the image. If this isn't defined, the image will be setup to run the CPU and buses at 13MHz and SDRAM will not be initialized.

`USE_ALL_STACKS`

The *USE_ALL_STACKS* definition enables extra code in the image for setting up all the stacks before calling c_entry(). The stacks are setup in IRAM before the MMU page table and are based on the sizes in the setup.inc or setup_gnu.inc files. If this definition is not set, then only the ARM SVC mode stack will be setup.

3.1.2.1 Default startup code configurations

Depending on which examples your building, the startup code's makefiles may include different options for the startup code to keep the image size smaller. As a general rule, the following assumptions are used foe building startup code for a specific boot device type. These can be changed by altering the makefiles for the example being built.

Large block NAND FLASH or SPI FLASH
The kickstart loader build options include MMU setup and board setup, but not relocation or all stack support. This generally creates an image size much smaller than 54K, allowing the boot ROM to load and execute these image types in IRAM.

Small block NAND FLASH
The kickstart loader build options do not include MMU setup, board setup, relocation, or all stack support. If all of these options are included, the image size would exceed 15.5K and wouldn't be bootable from small block NAND FLASH. Because these options are not enabled in the kickstart loader, the image loaded by kickstart cannot be loaded into SDRAM and the stage 1 application needs to initialize board specific code and MMU setup.

NOR FLASH boot
This type of boot method doesn't a kickstart loader and the system can boot directly into the stage 1 application from the boot ROM. MMU setup, board setup, and all stacks are supported. Because the code is running from read-only memory, relocation is also enabled allowing the data segment to be relocated to IRAM at boot time. This generally creates a larger image, but image size isn't a limitation with this boot method.

3.2 Board code

The board code consists of various drivers that support NOR, NAND, and SPI FLASH. In most cases, these drivers may need to be tweaked to support new hardware, although the tweaks are usually very minor.

The board drivers are not used directly by the startup code. However, applications such as the kictsart loader or the Stage 1 Loader use may use the drivers.

The board code provides the following features:

MLC NAND controller drivers
Small and large block NAND drivers are provided using the NAND MLC controller. This driver is used for burning a bootable image into block 0 for boot with the boot

ROM. Applications that are not booted from the boot ROM do not use this driver – they use the NAND SLC driver instead. Note the MLC NAND controller is not used in the normal LPC32x0 software such as u-boot, e-boot, or Linux; the SLC NAND controller is used instead.

SLC NAND controller drivers
Small and large block NAND drivers are provided using the NAND SLC controller. This driver is used for burning an image into NAND FLASH using the SLC NAND controller. This driver is not intended for burning bootable images into NAND that boot with the LPC32x0 boot ROM.

SPI FLASH drivers
A sample SPI FLASH driver is provided for burning images into SPI FLASH.

NOR FLASH drivers
A sample NOR FLASH driver is provided for burning images into NOR FLASH.

3.2.1 Information on bad block support

The MLC and SLC NAND controllers handle the spare area of NAND FLASH differently. The MLC controller has strict requirements for location of spare and ECC data and may overwrite factory bad block markers. The SLC controller does not alter the location of factory bad block markers. It is recommended that the NAND MLC controller only be used for the boot image in block 0, while the NAND SLC controller is used for normal NAND operation. The examples in this package are setup to use the NAND MLC controller only for block 0 and the NAND SLC controller for all other operations.

For small block devices, the factory provided bad block markers are not altered in block 0 with the NAND MLC driver. For large block devices, the factory bad block markers are altered with the NAND MLC controller. Because of this, the NAND SLC driver may report blocks on large block devices written with the NAND MLC controller as bad blocks due to the factory marker bring altered. As long as the NAND MLC controller isn't used to write to blocks used by the NAND SLC controller, this shouldn't be an issue.

3.2.2 Board code tweaks and configuration

The board drivers are located in the ./source directory. If you need to use one of these drivers for your board, you may need to make some tweaks to the driver for your hardware.

3.2.2.1 Board code configuration

Basic configuration for the board drivers can be made in the ./include/board_config.h file. This file contains configurations for NAND controller timings, NOR FLAHS timings, and SPI clock rate. These should be changed as necessary for new hardware. The timings are shown below:

```
/* For systems that use SPI, define the SPI clock rate here */
#define SPICLKRATE 5000000

/* Timing setup for the NAND MLC controller timing registers. These
   values can be adjusted to optimize the program time using the
   burner software with NAND. If your not worried about how long it
   takes to program your kickstart loader (or if your not using a
   kickstart loader), don't worry about changing these values. If you
   need to burn the kickstart, this can be speeded up by tweaking
   these values. See the 32x0 user's guide for info on these
   values. These should be programmed to work with the selected bus
   (HCLK) speed - because the burn image is usually small (under 54K),
   there really is not reason to change these values. */
#define MLC_TCEA_TIME   0x3
#define MLC_TWBTRB_TIME 0xF
#define MLC_TRHZ_TIME   0x3
#define MLC_TREH_TIME   0x7
#define MLC_TRP_TIME    0x7
#define MLC_TWH_TIME    0x7
#define MLC_TWP_TIME    0x7

/* Timing setup for the NAND SLC controller timing registers. On
   systems using NAND, these values effect how fast the kickstart
   loader loads the stage 1 application or how fast the S1L
   application handles NAND operations. See the 32x0 user's guide for
   info on these values. These should be programmed to work with the
   selected bus (HCLK) speed. */
#define SLC_NAND_W_RDY    0xF
#define SLC_NAND_W_WIDTH  0xF
#define SLC_NAND_W_HOLD   0xF
#define SLC_NAND_W_SETUP  0xF
#define SLC_NAND_R_RDY    0xF
#define SLC_NAND_R_WIDTH  0xF
#define SLC_NAND_R_HOLD   0xF
#define SLC_NAND_R_SETUP  0xF

/* External static memory timings used for chip select 0 (see the users
   guide for what these values do). Optimizing these values will help
   with NOR program and boot speed. These should be programmed to work
   with the selected bus (HCLK) speed. */
#define EMCSTATICWAITWEN_CLKS  0xF
#define EMCSTATICWAITOEN_CLKS  0xF
#define EMCSTATICWAITRD_CLKS   0x1F
#define EMCSTATICWAITPAGE_CLKS 0x1F
#define EMCSTATICWAITWR_CLKS   0x1F
#define EMCSTATICWAITTURN_CLKS 0xF
```

3.2.2.2 Board code tweaks

The various drivers may be setup for a specific device type and need some software tweaks prior to working on a new board. Each driver's require changes are detailed in the following sections.

Once these drivers are correctly setup for the new hardware, the applications that use them should work without changes.

3.2.2.2.1 NAND large and small block bad block offsets

The offset into a block for a bad block marker may need to be changed in the ./include/nand_support_common.h file.

3.2.2.2.2 NAND MLC large and small block drivers

In addition to the MLC NAND timings in the ./include/board_config.h file, some code in the board_mlc_nand_lb_driver.c and board_mlc_nand_sb_driver.c files in the ./source directory may need to be changed. Review the file to determine the required changes.

The following items may need to be changed:

- block and page addressing order may need to be adjusted for the specific NAND device
- In the nand_xx_mlc_init() function, the geometry (blocks, pages, data size) for the specific NAND device ID may need to be adjusted
- The addressing scheme used for block erase may need to be changed

The tweaks are expected to be very minor.

3.2.2.2.3 NAND SLC large and small block drivers

In addition to the SLC NAND timings in the ./include/board_config.h file, some code in the board_slc_nand_lb_driver.c and board_slc_nand_sb_driver.c files in the ./source directory may need to be changed. Review the file to determine the required changes.

The following items may need to be changed:

- block and page addressing order may need to be adjusted for the specific NAND device
- In the nand_xx_slc_init() function, the geometry (blocks, pages, data size) for the specific NAND device ID may need to be adjusted
- The addressing scheme used for block erase may need to be changed

The tweaks are expected to be very minor.

3.2.2.2.4 SPI FLASH driver

In addition to the SPI clock rate in the ./include/board_config.h file, some code in the board_spi_flash_driver.c file in the ./source directory may need to be changed. Review the file to determine the required changes. The default driver assumes a 64K addressable SPI FLASH device is supported.

The tweaks are expected to be very minor.

Note the SPI driver uses the SSP0 peripheral. The chip select is controlled manually via a GPIO signal instead of the automated SPI chip select. The current SPI driver will not program over 64Kbytes, as it only supports 16-bit addressing.

3.2.2.2.5 NOR FLASH driver

In addition to the EMC static device 0 timings (used for NOR FLASH) in the ./include/board_config.h file, some code in the board_nor_flash_driver.c in the ./source directory may need to be changed.

This driver is currently setup for NOR FLASH that supports the JEDEC programming sequence with 2 16-bit devices (to get a 32-bit boot configuration). This driver may require considerable tweaks to work with CFI FLASH, or other NOR FLASH types.

4 Applications – boot loaders, burner software, and stage 1 applications

This section covers the applications that are used with the startup code. When used with the startup code, an application can be anything that is entered via the c_entry() function as called from the startup code. After the startup code completes its setup, the c_entry() function of application is called. *The same startup code is used with all the applications in this package, but with different options enabled when the startup code is compiled.*

4.1 Boot loaders – kickstart and stage 1 applications

As detailed in the section on the LPC32x0 boot process, the maximum boot image size depends on the boot device used to store the boot image. This maximum boot image size may prevent the booted image from having all the desired features we want. In cases that the boot image size is smaller than the boot image, the boot image size needs to be trimmed so the boot ROM will correctly load and execute it.

For the purposes of this document and the accompanying software, the terms kickstart loader and stage 1 application are used to describe the bootable images for the LPC32x0.

A stage 1 application is the application we would ideally like to start when the LPC32x0 is reset or powered up. Stage 1 applications include programs such as u-boot, e-boot, or another full-featured application. Unfortunately, most stage 1 applications are too big to be directly booted by the LPC32x0 boot ROM. For systems where the stage 1 application can't be directly booted, a smaller 'kickstart' boot loader is required to boot form the LPC32x0 boot ROM and then continue loading and start execution of the stage 1 application.

For systems that boot from large block NAND FLASH or SPI FLASH, the maximum boot image size is 54K. If the application to be loaded exceeds this size or if the application needs to be loaded anywhere besides IRAM, the kickstart loader is required. The kickstart loader will in turn load the stage 1 application an transfer control to it after it has been loaded. If the application to be loaded is smaller than 54K and resides in IRAM, the stage 1 application can be directly loaded from the boot ROM instead of being routed through the kickstart loader.

For systems that boot from small block NAND FLASH, the maximum boot image size is 15.5K.

For systems that boot from NOR FLASH, no kickstart loader is needed (in most cases). The stage 1 application can be executed directly from NOR FLASH.

4.1.1 Kickstart loader

This package includes multiple variations of the kickstart loader that can be altered to work with new board designs. The variations provide support for boot from small and large block NAND FLASH, NOR FLASH, and SPI FLASH.

For most systems, a robust kickstart loader can easily be developed within 54K. This loader can include startup code that sets up the board's initial GPIO states and muxes, setup the initial clocking, setup a basic MMU table if needed, setup all the memory interface, and then load an image and transfer control to that image.

For small block NAND systems with the 15.5K boot image limit, this is a little tougher to manage. Tradeoffs may need to be made on what functions can be handled by the kickstart loader. Enabling all the functions may make a kickstart image that exceeds 15.5K. The image size would have to be reduced in this case by removing features from the kickstart loader and moving them to the stage 1 application. If your stage 1 application needs to execute in SDRAM, but the kickstart is too large with all the SDRAM init code, then code needs to be rearranged or aggressively optimized to meet the boot image size limitation.

4.1.1.1 Kickstart loader bootup procedure

The following sequence shows how the LPC32x0 handles kickstart load and transfer from the boot ROM using NAND or SPI FLASH.

- LPC32x0 processor is reset
- LPC32x0 boot ROM interrogates boot devices
- For SPI FLASH, the image loaded at SPI FLASH offset 8 is loaded into IRAM at address 0x0
- For NAND FLASH, the image loaded in page 1 of block 0 or 1 is loaded into IRAM at address 0x0
- After the image is loaded, the boot ROM transfers control to the image's startup code loaded at address 0x0

Depending on how the startup code is configured, the loaded image may have support for board setup (SDRAM, clocks, GPIO), MMU support, or stacks. After the startup code is executed, c_entry() is called to start execution of the actual loader application. The loader loads the stage 1 application from the desired device and then transfers control of the ARM processor to the loaded application.

If the board initialization code is part of the image and SDRAM is initialized, then the kickstart loader can load the stage 1 application into SDRAM. If SDRAM support isn't part of the loaded image, then only IRAM or external SRAM support is available. If the stage 1 application is loaded into IRAM, it must not be loaded into a location used by the kickstart loader. The kickstart loaders code and data start at address 0x0, while the MMU page table and stacks start at the end of IRAM. This allows some space for stage 1 applications to load into IRAM between the kickstart loader and the kickstart loader

stacks. Once the stage 1 application has started, it can reclaim the areas used by the kickstart loader.

4.1.1.1.1 Kickstart loader NAND FLASH operation

When the LPC3250 is powered and/or reset and NAND FLASH is used for system boot, the internal boot ROM checks the image in FLASH block 0 and loads it if it is valid. *See the LPC3250 User's Guide for more information on the boot process.* The image is loaded and execution is started at address 0x0.

The kickstart loader then starts loading the stage 1 application in page 0 of block 1. The kickstart loader will skip any blocks during the load process if the block is marked as bad, so if block 1 is bad, it will skip to block 2. The kickstart loader image load address and size depend on the values of several definitions when the kickstart loader was compiled. If the image size is greater than 1 block, the next block will be used to continue loading the image if the block isn't marked as bad. Once all the data is loaded, execution is passed to the image at its load address.

For a small block NAND FLASH device with 32 pages per block and 512 bytes per page, the kickstart loader and stage 1 application are organized as shown in Table 1.

Table 1 Kickstart and stage 1 application small block NAND FLASH data organization

Block	Page(s)	Data in NAND FLASH
0	0	Page dedicated to NAND boot from boot ROM
0	1-31	Kickstart loader (maximum 15.5K)
1[1]	0-31	Start of stage 1 application (32K)
2-?[1]	0-31	Stage 1 application code (continued)

[1]Bad blocks will be skipped. Block 0 is assumed to always be good.

For a large block NAND FLASH device with 64 pages per block and 2048 bytes per page, the kickstart loader and stage 1 application are organized as shown in Table 2.

Table 2 Kickstart and stage 1 application large block NAND FLASH data organization

Block	Page(s)	Data in NAND FLASH
0	0	Page dedicated to NAND boot from boot ROM
0	1-63	Kickstart loader (maximum 54K)
1[1]	0-63	Start of stage 1 application (128K)
2-?[1]	0-63	Stage 1 application code (continued)

[1]Bad blocks will be skipped. Block 0 is assumed to always be good.

The LPC3250 IROM and kickstart loader sequence is shown in Table 3.

Table 3 Kickstart loader operation

Operation	Function

LPC3250 is reset	IROM loads image from NAND FLASH based on data in block 0, page 0
Kickstart loader started	IROM passes control to kickstart loader at address 0x0
Board and MMU init	Kickstart performs optional board and MMU initialization
Read in stage 1 application	Kickstart loader reads stage 1 application data from NAND FLASH starting at block 1, page 0 (loaded into a different address than kickstart loader is using)
Transfer to stage 1 application	Transfer of control is passed to loaded application at the load address

The kickstart loader must be programmed into NAND FLASH using the NAND MLC controller or using ECC algorithms that work with the NAND MLC controller. However, the kickstart loader loads the stage 1 application from NAND using the SLC NAND controller. This is because the boot ROM uses the MLC NAND controller for NAND boot, while applications (such as Linux and WinCE) use the more flexible SLC NAND controller for normal NAND operation.

4.1.1.1.1.1 Kickstart loader NAND FLASH configuration definitions

The kickstart example code for small and large block NAND FLASH is located in the kickstart_lb_nand and kickstart_sb_nand directorys in the ./startup/examples/kickstart folder. The makefile included with these examples provides the default options used when building the startup code used with the kickstart loader.

The STAGE1_LOAD_ADDR and STAGE1_LOAD_SIZE definitions need to be setup prior to building the kickstart loader. If building the stage 1 application to run in IRAM, the address must not overlap the code and data areas used by the kickstart loader. If building the stage 1 application to run form SDRAM, the kickstart loader must initialize SDRAM prior to loading the stage 1 application into IRAM.

The STAGE1_LOAD_ADDR and STAGE1_LOAD_SIZE definitions can be configured in the misc_config.h header file.

4.1.1.1.2 Kickstart loader SPI FLASH operation

When the LPC3250 is powered and/or reset and SPI FLASH is used for system boot, the internal boot ROM checks the 32-bit word in SPI FLASH offset 0. If the 32-bit word matches the pattern used to identify a bootable image, the 32-bit word at offset 4 is loaded and used for the boot image size. *See the LPC3250 User's Guide for more information on the boot process.* The image is loaded and execution is started at address 0x0.

The kickstart loader then starts loading the stage 1 application in SPI FLASH at an offset past the kickstart loader. The kickstart loader image load address and size depend on the

values of several definitions when the kickstart loader was compiled. Once all the data is loaded, execution is passed to the image at its load address.

Table 4 shows a possible approach to organizing SPI FLASH with the kickstart loader and stage 1 application.

Table 4 Possible kickstart and stage 1 application SPI FLASH data organization

Offset	Data in NAND FLASH
0	SPI FLASH boot ID (4 bytes)
4 (load_size)	Image load size (4 bytes)
8	Kickstart application (load_size bytes)
8 + (load_size)	Stage 1 application code

The LPC3250 IROM and kickstart loader sequence is shown in Table 5.

Table 5 Kickstart loader operation

Operation	Function
LPC3250 is reset	IROM loads image from SPI FLASH at offset 8 with size designated at offset 4
Kickstart loader started	IROM passes control to kickstart loader at address 0x0
Board and MMU init	Kickstart performs optional board and MMU initialization
Read in stage 1 application	Kickstart loader reads stage 1 application data from SPI FLASH
Transfer to stage 1 application	Transfer of control is passed to loaded application at the load address

4.1.1.1.2.1 Kickstart loader SPI FLASH configuration definitions

The kickstart example code for SPI FLASH is located in the kickstart_spi_flash directory in the ./startup/examples/kickstart folder. The makefile included with this example provides the default options used when building the startup code used with the kickstart loader.

The STAGE1_LOAD_ADDR, STAGE1_LOAD_SIZE , and SPI_S1APP_OFFSET definitions need to be setup prior to building the kickstart loader. The SPI_S1APP_OFFSET definition is used to define where the stage 1 application offset is located in the SPI FLASH.

If building the stage 1 application to run in IRAM, the address must not overlap the code and data areas used by the kickstart loader. If building the stage 1 application to run from SDRAM, the kickstart loader must initialize SDRAM prior to loading the stage 1 application into IRAM.

The STAGE1_LOAD_ADDR, STAGE1_LOAD_SIZE , and SPI_S1APP_OFFSET definitions can be configured in the misc_config.h header file.

4.1.1.1.3 Kickstart loader and NOR FLASH

The kickstart loader can be used for NOR FLASH if an execute-in-place model for the NOR bootable stage 1 application isn't possible. For example, if u-boot needs to run from SDRAM, it may be complicated to develop a version of u-boot the initializes SDRAM when running from NOR FLASH and then relocates code and data to SDRAM. In this case, the kickstart loader can be used initially to boot from NOR FLASH and initialize the board and memory. After SDRAM is operational, the u-boot image can be copied from NOR FLASH (for example, at offset 32K) into SDRAM. *This method is used for u-boot that executes in SDRAM and boots from NOR FLASH.*

4.2 Stage 1 applications

Stage 1 applications are applications that meet one of the following criteria:

- Loads and executes directly from the UART into IRAM
- Loads and executes from the kickstart loader in IRAM or SDRAM
- Loads and executes in place from NOR FLASH

If the kickstart loader is used to load and start a stage 1 application, then the STAGE1_LOAD_ADDR, STAGE1_LOAD_SIZE , and SPI_S1APP_OFFSET (SPI FLASH bootable systems only) definitions in the misc_config.h header file should be setup to identify where the stage 1 application is loaded in memory from the kickstart loader and the image size loaded.

Note: There is no requirement to boot a stage 1 application using the same device that is used with the kickstart loader. For examples, the kickstart loader for loading a stage 1 application from large block NAND can be booted from SPI FLASH. For this specific case, the kickstart loader would br programmed into SPI FLASH and the kickstart loader would load and start the stage 1 application from large block NAND FLASH.

4.3 Kickstart and stage 1 application scenarios

The following section describes some real world scenarios using the kickstart loader and stage 1 applications used on current boards.

4.3.1 Examples with the Phytec 3250 board

4.3.1.1 S1L and the Phytec 3250 board using small block NAND FLASH

The maximum boot image size using this boot configuration is 15.5K. For a typical S1L image size of about 56K, the kickstart loader is required to load and start S1L. Also, because of the small boot image size, the kickstart loader has been trimmed to only load the stage 1 application image from NAND FLASH – the board and SDRAM initialization

and MMU setup code are not in the kickstart. Because of this, S1L runs from IRAM and its startup code handles board, SDRAM, and MMU initialization.

With this boot configuration, memory is very tight. Loading a stage 1 application larger than about 180K into IRAM may be challenging.

The default version of the kickstart loader and S1L included on the Phytec board used small block NAND FLASH. Because of this, the board bringup code isn't in the kickstart loader. The kickstart loader loads the stage 1 application into IRAM and the stage 1 application actually sets up the board. This approach works, but does have some drawbacks. Because the kickstart loader doesn't setup the board, the stage 1 application can only boot into IRAM and must setup the board. This makes larger applications such as u-boot more challenging to handle with small block boot. For this scenario, S1L helps bridge the gap between the kickstart and u-boot providing a capability to load u-boot from a boot device into SDRAM after S1L has initialized SDRAM.

4.3.1.2 S1L and the Phytec 3250 board using NOR FLASH

There is no limit on boot size using this configuration so S1L can execute directly from NOR FLASH. The start code in S1L initializes the board, SDRAM, and MMU tables. Prior to using any read-write data, the data segment is relocated to IRAM.

This version is convenient because the kickstart loader isn't required. However, the use of data relocation can be tricky to handle in software, especially if the data is located in SDRAM and SDRAM hasn't yet been setup yet by the code running in NOR FLASH. The data may need to be re-locatable to written in such as way as to avoid any SDRAM accesses until SDRAM has been initialized.

4.3.1.3 S1L and the Phytec 3250 board using SPI FLASH

The maximum boot image size using this boot configuration is 54K. For a typical S1L image size of about 56K, the kickstart loader is required to load and start S1L. However, the kickstart loader has enough room to include the board, SDRAM initialization and MMU setup code.

There is no requirement to boot the stage 1 application from SPI FLASH. The kickstart loader can be loaded and started from SPI FLASH. SPI FLASH can then initialize SDRAM. Then the loader application of the kickstart loader can load a larger application, such a u-boot (the Linux boot loader) into SDRAM.

Optionally, it might be possible to trim some functions from S1L to get the image size under 54K and boot directly into S1L from the boot ROM.

Since the kickstrt loader in SPI FLASH initializes SDRAM, the stage 1 application doesn't have to initialize SDRAM. The kickstart loader can load the stage 1 application directly into SDRAM. For u-boot (which doesn't provide board, clock, and SDRAM

initialization), the capability to boot u-boot directly from the kickstart loader makes u-boot porting a bit easier for new board designs. If needed, u-boot can directly update its image in SPI FLASH and the kickstart loader will boot it on next reset without the need to update the kickstart loader (as long as the stage 1 load size in greater than or equal to the u-boot image size programmed into FLASH).

4.3.2 S1L and the Embedded Artists 3250 board using large block NAND FLASH

The maximum boot image size using this boot configuration is 54K. The same reasoning described in section 4.3.1.3 applies to large block NAND boot.

4.4 Burner software

The term 'burner software' refers to the example code included in this package for programming the kickstart loader or a stage 1 application in the NAND, SPI, or NOR FLASH. Examples programs are provided for burning the kickstart load and stage 1 application for NAND small and large block devices using the MLC and SLC NAND controllers, SPI FLASH, and NOR FLASH.

The burner software executes in IRAM at address 0x0 and works with the Serial Loader tool. The image to be loaded can be loaded into IRAM or SDRAM (assuming the startup code of the burner software initializes SDRAM). The size of the image to program is based on the size of the image transferred to the burner software.

Table 6 shows the possible burner methods for different boot methods for the kickstart loaders and stage 1 applications.

Table 6 Burner sources for various boot methods

Burn location	When to use this burner application	Location of example code under the ./startup/examples/burner directory
Large block NAND FLASH block 0	For burning images that load from large block NAND FLASH into IRAM such as the kickstart loader using the MLC NAND controller	./nand_lb/kickstart
Large block NAND FLASH blocks 1 and on	For burning images that load from the kickstart loader only using the SLC NAND controller	./nand_lb/s1app
Small block NAND FLASH block 0	For burning images that load from large block NAND FLASH into IRAM such as the kickstart loader using the MLC NAND controller	./nand_sb/kickstart
Small block NAND FLASH blocks 1 and	For burning images that load from the kickstart loader only using the	./nand_sb/s1app

on	SLC NAND controller	
SPI FLASH (kickstart)	For burning images that load from SPI FLASH into IRAM such as the kickstart loader (burns at offset 8)	./spi/kickstart
SPI FLASH (stage 1 application)	For burning images that load from the kickstart loader only	./spi/s1app
NOR FLASH	For burning images that boot directly from NOR FLASH	./nor

4.4.1 How the burner software works

The burner software works with the serial loader tool to burn the desired boot image into the desired location in a non-volatile device. The burner software can program bootable images for the LPC32x0 boot ROM or for the kickstart loader.

The basic operation of the burner software used with the Serial Loader tool is shown in Table 7

.

Table 7 Serial Loader and burner sequence

Serial loader tool (running on PC)	LPC32x0 based board
Burner image and boot image are selected in the serial loader tool file boxes	
Serial loader waits for ‘5’	
	LPC32x0 is reset or powered on
	LPC32x0 sends ‘5’
Serial loader sends ‘A’	
Serial loader waits for ‘5’	
	LPC32x0 sends ‘5’
Serial loader sends ‘U’ and ‘3’	
Serial loader waits for ‘R’	
	LPC32x0 sends ‘R’
32-bit load address is sent (4 bytes)	
32-bit load size is sent (4 bytes)	
Burner image is sent to the board	
Serial loader waits for ‘X’	
	LPC32x0 starts the burner image
	Burner image startup code is executed
	Burner image startup code sends ‘X’
	Burner image startup code waits for ‘p’
Serial loader sends ‘p’	
32-bit load address is sent (4 bytes)	
32-bit load size is sent (4 bytes)	

Serial loader waits for ‘o’	
	LPC32x0 sends ‘o’
Kickstart loader or S1L image is sent to the board	
Serial loader waits for ‘t’	
	Burner image startup code sends ‘t’
	Burner image programs kickstart or S1L image in non-volatile memory
	Program status is output to serial port and burner image halts

4.4.2 Burner software configuration

The burner software requires one definition value to be setup to work correctly. The BURNER_LOAD_ADDR definition in the misc_config.h header file needs to be configured for the load address the burner application will use to temporarily store the image to burn. If the startup code used for the burner application sets up SDRAM memory, this location can in SDRAM. Otherwise, this location must be in IRAM or external SRAM. The maximum image load size is limited by this value (for example, IRAM only has 256K max, but some is used for the burner software itself, so less than 256K can be used.).

5 Serial Loader tool

The Serial Loader tool is used with the LPC32x0 on reset to provide a UART based boot capability. An image can be built that loads into Internal RAM (IRAM) on system reset via the UART. This helps save time during early board debug trying to get a full featured boot loader up and running with non-volatile boot support.

To use the Serial Loader tool with a board, the board must have UART5 available at boot time and the nService jumper state setup to support UART boot.

5.1 Serial Loader tool use

The Serial Loader tool is included with the LPC32x0 CDL in the LPC32x0 software area. To start the tool, simple double click the “LPC3250_Loader.exe” icon. After it starts, you should see an image similar to Figure 1. A binary image can be loaded and executed in IRAM at address 0x0 by selecting the image name in the “Primary Boot (IRAM)” box, pressing the “Load bin’s/start primary” button, and then resetting the board. If the board is configured correctly (connected to UART5 with nService mode enabled), the image will automatically download to the board and start executing.

The IRAM version examples of S1L included in the BSP can be downloaded and executed in the board using this method without any special tools (ie, JTAG) and no requirement to burn the image into a non-volatile device to system boot. This is a great way to perform early testing and debug of the system.

If the “Secondary Executable (SDRAM)” box is checked with an associated binary file, the Serial Loader tool will attempt to send a 2nd image after the first image has sent. This is intended to be used with the burner software as described in the previous sections.

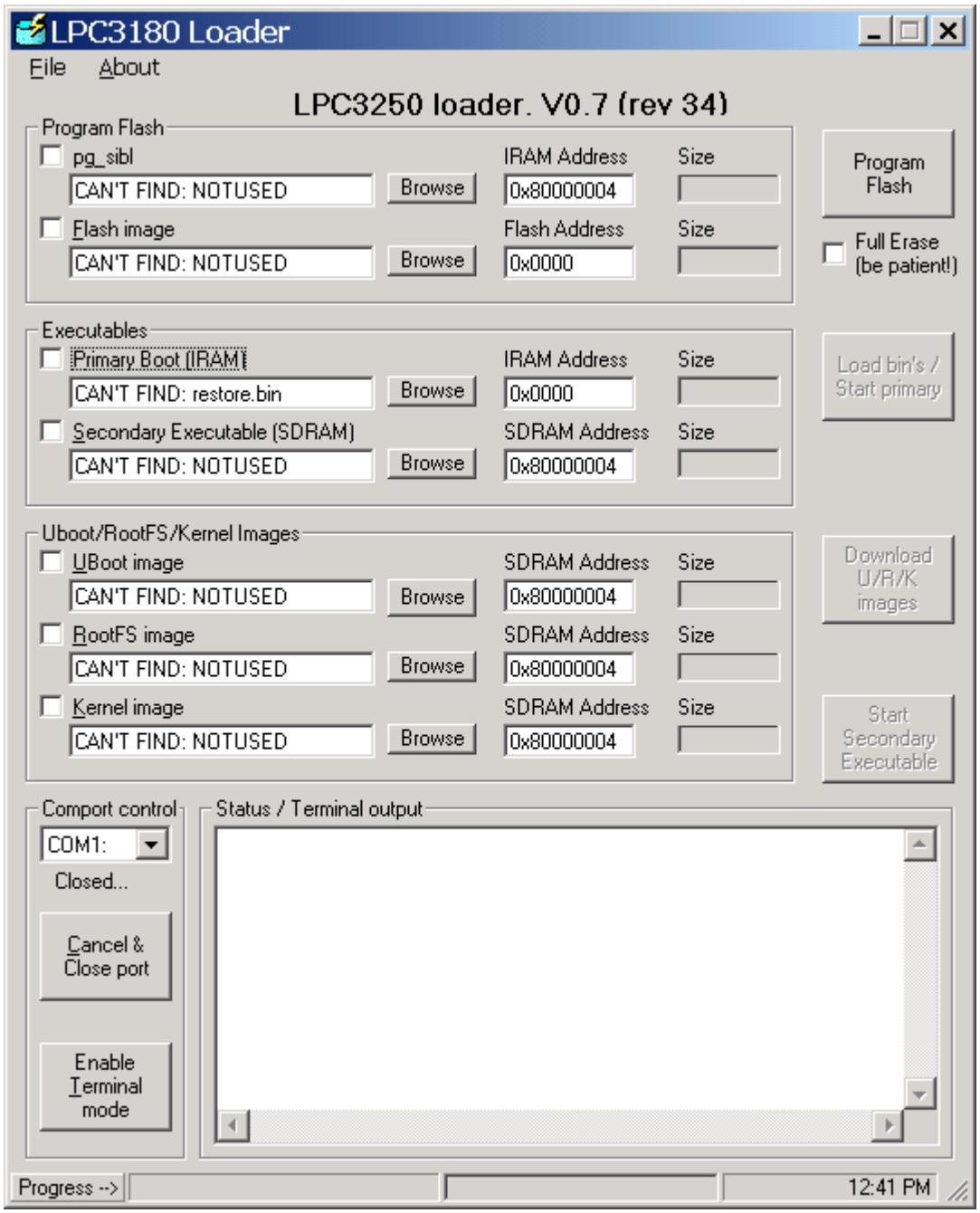

Figure 1 Serial Loader tool

5.1.1 Serial Loader tool example using S1L for IRAM

In the following example, a version of S1L was built for IRAM and the binary setup as the primary boot file. After the board was reset, the file was transferred over the UART and started. The complete output from boot to the S1L interactive prompt is shown below.

Waiting for BootID .. 5 .. found!
Sending 'A' .. done!
Expect 2-nd BootId .. 5 .. found!
Sending 'U','3'.. done!
Expect 'R' .. R .. found!
Sending startaddress .. done!
Sending size .. done!
Sending code .. done!

--- Enabling terminal mode ---
Using default system configuration

S1L for the LPC32x0
Build date: May 18 2010 11:51:37

32x0>

5.1.2 Serial Loader tool example using burner software

In the following example, a version of the kickstart loader was built along with the kickstart burner application. The kickstart burner was setup as the primary boot file, while the kickstart loader was setup as the secondary. When the board was reset, the burner was first transferred over to the board via the UART and then execution of the burner was started. The kickstart loader was then transferred to the kickstart burner software and then programmed into the board. After the 2nd file was transferred, the "Enable terminal mode" button was pressed to see the program status. The complete output from boot to the program status is shown below.

--- Switching to programmed mode ---
Waiting for BootID .. 5 .. found!
Sending 'A' .. done!
Expect 2-nd BootId .. 5 .. found!
Sending 'U','3'.. done!
Expect 'R' .. R .. found!
Sending startaddress .. done!
Sending size .. done!
Sending code .. done!

--- Loading Secondary executable ---
Wait for 'X' .. X .. found!
-- Sending command .. p ..done!
Sending startaddress .. done!
Sending size .. done!

Wait for acceptance from primary boot .. o ..OK!
Sending code .. done!
Expect final 't' .. t .. found,
---- Secondary Executable loaded. ---

--- Enabling terminal mode ---
Formatting blocks...
Format complete
Writting kickstart into flash...
Verifing data.....Successfully
NAND flash is programmed Successfully

6 Stage 1 Loader

This package includes a generic version of the Stage 1 Loader (S1L) that can be easy ported to a new LPC32x0 based board or used for early board debug. S1L is a small program that usually runs in IRAM and provides a monitor program with a collection of functions to help with debug and application development. The main features of the stage 1 loader are shown below:

- Register and memory change and dump
  - Poke, peek, dump, fill
- Image load via a serial port , SDMMC card, or FLASH
  - Supports raw binary and S-record files
  - Images can be executed after loading
  - Images can be saved in NAND FLASH
- NAND FLASH support
  - Erase of NAND blocks
  - Direct read and write of FLASH blocks and pages
  - Bad block management
- MMU functions
  - Data and instruction cache control
  - Virtual address translation enable/disable
  - Virtual address remapping
  - Page table dump
- System support functions
  - Baud rate control, clock control, system information
- Automatic load and run support
  - Automatic load and execution of images from NAND FLASH, SDMMC, or via the terminal
- Testing functions
  - SDRAM memory tests, bandwidth tests
  - SDRAM calibration and configuration data

The source code for the kickstart loader is provided free of charge by NXP for use with NXP processors.

6.1 Stage 1 loader startup

The stage 1 loader is started at address 0x8000 once it is loaded from the kickstart loader. (S1L loaded directly into IRAM uses address 0x0.) Depending on how the startup code is configured, the board, clocks, and SDRAM may or may not be initialized as part of the stage 1 application.

The default build options do not add board init code for stage 1 loader operation from NAND large block FLASH, SPI FLASH. However, NOR FLASH or small block NAND FLASH S1L images do have board init code.

Regardless of how startup for S1L is configured, once startup is completed, the S1L program is started. Depending on how S1L is configured, S1L may attempt to load and execute and image, or go to the interactive prompt.

6.2 Stage 1 loader resource usage

The stage 1 loader uses a UART and several standard timers. Depending on which board specific options are enabled, S1L may also use the NAND FLASH controllers, the SD card controller, the SPI interface, or other interfaces.

Besides the storage space used for S1L itself, S1L may also use a small portion of non-volatile storage for saved S1L data. The location of this saved data varies per board. (For example, the Phytec 3250 board uses SPI FLASH, while the EA3250 board uses I2C FLASH). Depending on the speed of the non-volatile interface, operations to save and load this data may be instant or very slow. When S1L is first started, a load of this device is performed; a slow interface may delay S1L startup.

When applications are loaded and executed with the stage 1 loader, the loader will disable all its used peripherals prior to executing the loaded application. Some peripheral and loader settings are passed unchanged to the application; these include the settings configured by the startup code. When the application is called, the system is in the state as setup by the startup code and changed by S1L:

MMU page table located at address 0x0803C000 (startup code)
MMU may be enabled or disabled (startup code or S1L)
Data and instruction caches may be enabled or disabled (startup code or S1L)
SDRAM initialized at address 0x80000000 (startup code)
GPIO directions, states, and muxing unchanged from startup code (startup code)
All peripherals disabled (S1L)
All stacks in IRAM (startup code)

Applications called from the stage 1 loader may safely use the resources setup by the stage 1 loader. If the application wants to return to the stage 1 loader, the application must return the system back to its original state before exiting. Applications that require bigger stacks should setup their own stacks as part of the application. If an application overwrites the memory used by the stage 1 loader, returning from the application may cause the system to crash.

6.2.1 IRAM organization

IRAM usage of S1L on a LPC32x0 device with 256K of IRAM as booted from the kickstart loader is organized as shown in Table 8.

Table 8 IRAM organiztion for stage 1 loader

IRAM base	Size (bytes)	IRAM use
0x0003C000	16K	MMU page table
0x0003C000[1]	512	FIQ mode stack
0x0003BE00[1]	1024	IRQ mode stack
0x0003B800[1]	512	Abort exception stack
0x0003B600[1]	512	Undefined instruction exception stack
0x0003B400[1]	512	System mode stack
0x0003B200[1]	4096	Service mode stack
0x00008000	As required	Stage 1 loader code and data
0x00000000	32K	Kickstart loader area, safe to use

[1]Stacks grown down.

6.2.2 Stage 1 loader persistent data storage

The stage 1 loader may store configuration data in non-volatile memory. This data contains information such as the prompt string, baud rate, autoboot parameters, and image stored in FLASH. These values are used by the stage 1 loader between power cycles to remember the last stage 1 loader configuration. Some commands automatically updated this data (such as the prompt command) whenever a configurable options is changed. The first time the board is powered up, the data is populated with default values.

The following configurable items can be changed with the corresponding data stored in the serial EEPROM:

- Prompt string
- Autoboot timeout
- Autoboot source, file type, load address, filename, execution address
- FLASH initial format status
- Configured baud rate
- Saved FLASH image status
- Startup script

6.2.2.1 Resetting default values

If for some reason, the board is configured in such as way as to become unbootable after the stage 1 loader starts, holding down a special configuration reset button as the board is booting will restore the default values for the configuration data. The restored values will only apply for that power cycle.

6.3 Stage 1 loader monitor program operations

This section explains the monitor program operations that can be executed from the command prompt of the stage 1 loader.

6.3.1 Accessing the monitor program

The monitor program can be accessed by connecting a serial cable between a PC running a terminal program and UART5 on the LPC32x0 based board. The terminal program (such as Teraterm) should be configured for 115.2Kbits/sec, 8 data bits, no parity, and 1 stop bit. A board message and prompt such as *3250>* should appear on the terminal. If the default configuration has been changed, the prompt may be different or another program may have been configured to be autobooted.

Commands are entered at the monitor program prompt. All commands are case insensitive. Pressing DEL on the current command line will erase the current command entered. After each command has been typed, pressing enter will execute the command.

6.3.1.1 Command syntax

Commands consist of a base command and a required or optional list of arguments. A command has the following syntax:
command [required argument] <optional argument>

Arguments for commands with options may not always be required. For some commands, arguments are required and the command will fail if the arguments are not specified.

Argument types vary per command, but most commands share the type of arguments that can be used. Examples of common argument types are listed below and the required format for that argument.

<hex>
This value must be a hexadecimal value preceded with a “0x” such as 0x1234, 0x00800100, or 0x9.

<bytes>, <width 1, 2, 4>, [starting sector], etc.
These values are decimal such as 128, 1, or 500.

6.3.2 Monitor program commands

6.3.2.1 Support and configuration commands

6.3.2.1.1 help

The *help* command is used to display the menu or the syntax of a specific command. Type *help menu* to see the current supported commands, or type *help <command>* to display the syntax and instructions for a specific command.

Command syntax:
help menu
help all

help <group>
help <command>

6.3.2.1.2 baud

The *baud* command is used to reset the baud rate. The usable baud rates in bits per second are 115200, 57600, 38400, 19200, and 9600. Once a baud rate has been set, it becomes immediately effective. Configured baud rate is persistent and will remain set to this new value across power and reset cycles.

Command syntax:
baud [new baud rate (115200, 57600, 38400, 19200, 9600)]

Persistent configuration:
Saved across power cycles and resets.

6.3.2.1.3 bwtest

Measures the amount of Microseconds required to copy data from one region of memory to another. This function can be used to measure typical bandwidth between cached and uncached SDRAM and SRAM regions. Burst operations sized at 4 32-bit words are used to transfer data between the source and destination. Both the source and destination addresses must be aligned on a 32-bit boundary. The number of bytes to test must be divisble by 16. The actual number of bytes transferred and the total time in MicroSeconds required to transfer the data will be displayed.

Command syntax:
bwtest [hex addr 1] [hex addr 2] [bytes] [loops]

Examples:
Bwtest 0x80000000 0x82000000 1048576 50 (Move 1048576 bytes of data between source address 0x80000000 and destination 0x8200000)

6.3.2.1.4 comp

Performs a bytewise data comparison between 2 data regions.

Command syntax:
comp [hex addr 1] [hex addr 2] [bytes]

Examples:
Comp 0x80000000 0x82000000 1024 (Compares 1024 bytes)

6.3.2.1.5 copy

Performs a bytewise data copy between 2 data regions.

Command syntax:

copy [hex addr 1] [hex addr 2] [bytes]

Examples:
*copy 0x80000000 0x82000000 1024 (c*opies *1024 bytes from 0x80000000 to 0x82000000)*

6.3.2.1.6 dump

Dumps a range of data sized in 8-bit, 16-bit, or 32-bit chunks. The command can be used without arguments to continue a dump where it previously ended with the same characteristics of the previous dump command.

Command syntax:
dump <hex addr> <bytes> <width 1, 2, 4>

Examples:
dump 0x80000000 256 4 (Dumps 64 32-bit words starting at address 0x80000000)
dump (Dumps data based on the previous dump command at the last address)

Notes:
Dumping invalid memory ranges when the MMU is enabled will cause the stage 1 loader to generate an exception.

6.3.2.1.7 fill

Fills a data range with a value.

Command syntax:
fill [hex addr] [hex bytes] [hex value] [width 1, 2, 4]

Examples:
fill 0x80008000 0x80 0xAA 1 (Fills 128 bytes starting at address 0x80008000 with 0xAA)
fill 0x80010000 0x40 0x00 2 (Fills 32 16-bit words starting at address 0x80010000 with 0x0000)

Notes:
Filling invalid memory ranges when the MMU is enabled will cause the stage 1 loader to generate an exception.

6.3.2.1.8 info

Displays current system information such the MMU enabled status, clock speeds, FLASH geometry, loaded image, image in FLASH, and autoboot configuration.

Command syntax:
info

6.3.2.1.9 peek

Dumps the 8-bit, 16-bit, or 32-bit value at a specific address location. This function is good for examining individual locations. Consecutive calls to peek without arguments will always dump the same address and size.

Command syntax:
peek <hex addr> <width 1, 2, 4>

Examples:
peek 0x80008000 1 (Dumps the hex byte at address 0x80008000)
peek (Dumps the same hex information as the previous use of this command)

Notes:
Peeking invalid memory when the MMU is enabled will cause the stage 1 loader to generate an exception.

6.3.2.1.10 poke

Places a 8-bit, 16-bit, or 32-bit hex value into an address.

Command syntax:
poke [hex addr] [hex value] [width 1, 2, 4]

Examples:
poke 0x80000000 0x99 1 (Places the value 0x99 into the byte address 0x80000000)
poke 0x80000000 0x99 2 (Places the value 0x0099 into the 16-bit address 0x80000000)

Notes:
Poking invalid memory when the MMU is enabled will cause the stage 1 loader to generate an exception.

6.3.2.1.11 prompt

Changes the default prompt and autoboot delay. The prompt can be changed to any string with a maximum length of 16 characters. The timeout delay is used to prevent an image from automatically loading and starting when the stage 1 loader is started. A timeout delay of 0 will always cause the command prompt to be displayed.

Command syntax:
prompt [name] [timeout(secs), 0 = no timeout]

Examples:
prompt lpc> 0 (Sets the prompt to lpc> and disabled the autoboot delay)
prompt : 10 (Sets the prompt to : and sets the autoboto delay to 10 seconds)

Persistent configuration:

Saved across power cycles and resets.

6.3.2.1.12 rstcfg

Restores the default system configuration. Resets the prompt to lpc3250>, the baud rate to 115200, and the autboot timeout to 0. Also restores default system clock values.

Command syntax:
Rstcfg

6.3.2.1.13 ls

Display the files in the root directory of the SDMMC card. Files are display in 8.3 file format.

Command syntax:
ls

Notes:
SD and MMC cards are supported. High capacity cards are not supported.

6.3.2.1.14 script

This command sets up initial S1L commands that the system will execute when the system is powered up or reset. Scripts cane be used to automatically load and start one or more files, or fill in memory with specific values prior to the S1L prompt appearing. When the setup subcommand of the script command is used, a prompt will appear that allows you to enter each script line. You cannot use backspaces or direction keys on this line. Press enter on an empty line to accept your scripts.

Command syntax:
script [on, off, setup]

Examples:
script on
script setup

Persistent configuration:
Scripts ands script states are saved across power cycles and resets

Notes:
There is a limit of 256 characters per script line with a maximum of 16 script lines and a total script size (all lines) that cannot exceed 1024 bytes.

6.3.2.2 MMU control commands

6.3.2.2.1 dcache

Enables, disables, or flushes the data cache. The data cache is only operational when the MMU is enabled. When disabling the data cache, the data cache is flushed prior to the data cache being disabled. When enabling the data cache, the data cache is invalidated prior to being enabled.

Command syntax:
mmu dcache [0, 1, 2]

Examples:
dcache 0 (Flush and disable data cache)
dcache 1 (Invalidate and enable data cache)
dcache 2 (Flush data cache)

Persistent configuration: No, always enabled when the stage 1 loader is started.

6.3.2.2.2 inval (Cache flush and invalidate)

Flushes and invalidates both the instruction and data caches.

Command syntax:
inval

6.3.2.2.3 icache

Enables or disables the instruction cache. When enabling the instruction cache, the instruction cache is invalidated prior to being enabled.

Command syntax:
mmu icache [0, 1]

Examples:
icache 0 (Disable instruction cache)
icache 1 (Invalidation and enable instruction cache)

Persistent configuration: No, always enabled when the stage 1 loader is started.

6.3.2.2.4 map

Maps or unmaps a range of physical addresses to virtual addresses. Depending on the argument type, a virtual address range can be mapped as cached or uncached.

Command syntax:
map [virt hex addr] [phy hex addr] [sections] [0, 1, 2]

Examples:
map 0x40000000 0x80000000 32 0 (Maps 32Mbytes of data at address 0x80000000 to virtual address 0x40000000 as uncached)

map 0x40000000 0x80000000 8 1 (Maps 8Mbytes of data at address 0x80000000 to virtual address 0x40000000 as cached)
map 0x40000000 0x80000000 32 2 (Unmaps 32Mbytes of data at virtual address 0x40000000)

Notes:
Unmapped regions will generate an exception if accessed.

Persistent configuration: No, default MMU page table from startup code is used when the stage 1 loader is started.

6.3.2.2.5 mmuenab

The *mmuenab* command enables and disables the MMU. When disabling the MMU, the data cache is flushed prior to the MMU being disabled. When enabling the MMU, the data and instruction caches are invalidated prior to being enabled.

Command syntax:
Mmuenab <0, 1>

Notes:
Use 0 to disable the MMU or 1 to enable it. Because the stage 1 loader's virtual and physical addresses are identical, this can be safely performed in the stage 1 loader.

Persistent configuration: No, always enabled when the stage 1 loader is started.

6.3.2.2.6 mmuinfo

Dumps the current MMU, data cache, and instruction cache enabled status. Also dumps the current virtual to physical address mapping, region sizes, and cached region status.

Command syntax:
mmuinfo

6.3.2.3 NAND support commands

6.3.2.3.1 erase

Erase a range of FLASH blocks. Bad blocks are not erased unless specifically requested to be erased.

Command syntax:
erase [starting block] [number of blocks] [erase bad blocks]

Examples:
erase 100 1000 0 (Erases blocks 100 to 1099, ignoring bad blocks)
erase 300 800 1 (Erases blocks 300 to 1099, including bad blocks)

Notes:
Care must be taken to not erase any applications loaded in FLASH. If the kickstart loader or stage 1 loader FLASH data areas are attempted to be erased, verification will be requested for the operation. Erasing an application stored with *nsave* will prevent the *nload* function from working although the *info* command will still indicate that the image is stored in FLASH.

6.3.2.3.2 nandbb

Generates a list of bad blocks on the FLASH device.

Command syntax:
nandbb

6.3.2.3.3 read

Reads a range of FLASH sectors into memory. Bad blocks are normally skipped, but can be optionally loaded during the operation.

Command syntax:
read [starting sector] [sectors] [hex address] [erase bad blocks]

Examples:
read 100 32 0x80000000 (Reads 32 sectors starting at sector 100 into address 0x80000000, bad blocks are skipped)
read 480 64 0x80000000 1 (Reads 64 sectors starting at sector 480 into address 0x80000000, bad blocks are also read)

Notes:
Sectors do not need to be aligned to the first page in a block.

6.3.2.3.4 write

Writes data from memory into a range of FLASH sectors. Bad blocks are normally skipped, but can be optionally used during the operation.

Command syntax:
nwrite [starting sector] [sectors] [hex address] [erase bad blocks]

Examples:
write 100 10 0x81000000 0 (Writes 10 sectors of data from address 0x81000000 to FLASH starting at sector 100, skips bad blocks)
write 500 25 0x81000000 1 (Writes 25 sectors of data from address 0x81000000 to FLASH starting at sector 500, bad blocks are not skipped)

Notes:

Sectors do not need to be aligned to the first page in a block.

6.3.2.3.5 nburn

Places an image loaded into memory at a contiguous location in FLASH. This function is typically used to place large amounts of data in FLASH.

Command syntax:
Nburn [starting block][overwrite bad blocks]

Example:
nburn 100 0 (Stores the image in FLASH starting at block 100, skips bad blocks)

6.3.2.3.6 dumpex

Dumps the data in the spare data area of a NAND FLASH range of sectors.

Command syntax:
dumpex [starting sector][number of sectors]

Example:
dumpex 3200 10 *(Dumps 10 sector's spare data area starting at sector 3200)*

6.3.2.4 Application support commands

These commands support loading, saving, executing, and booting user applications.

6.3.2.4.1 nload

Loads an image stored in FLASH into memory. The image was previously saved with the *nsave* command as a raw binary image. The information for the currently loaded image can be examined with the *info* command. The *info* command also shows the information for the currently saved image in FLASH from the *nsave* command.

Command syntax:
nload

Notes: See Section 6.3.4 for more information on this command.

6.3.2.4.2 nsave

Saves an image stored in memory into FLASH that can be used with the *nload* command. The information for the currently loaded image can be examined with the *info* command. After this command, the *info* command shows the updated information for the saved image in FLASH.

Command syntax:
nsave

Notes: Only raw binary images that use a contiguous data area can be saved in FLASH. See Section 6.3.4 for more information on this command.

Persistent configuration: Updates data about the saved image in FLASH.

6.3.2.4.3 load

This command loads an image from the terminal, SD or MMC card, or from NAND into memory. Supported image types include raw binary, and S-record files. Files loaded from an SD or MMC card also require a filename in the 8.3 file format. Only files in the root directory of the SD or MMC card can be loaded.

Command syntax:
load [src <file>] [ty] <hex addr 1> <hex addr 2>

Examples:
load blk im1.bin raw 0x80000000 (Loads the im1.bin file from the root directory of the SD/MMC card into address 0x80000000 with execution address 0x80000000)
load term raw 0x80008000 0x8000A900 (Loads a raw binary image from the terminal at address 0x80008000 with execution address 0x8000A900)
load flash srec (Loads the image saved with nsave using an S-record format)

Notes: See Section 6.3.3 for more information on this command.

6.3.2.4.4 aboot

Sets up the board to automatically load and boot an image from the terminal, SD or MMC card, or from FLASH.

Command syntax:
aboot [src <file>] [ty] <hex addr 1> <hex addr 2>

Examples:
aboot blk ldr.hex srec (Loads the S-record file ldr.hex from the root directory of the SD or MMC card and executes it)
aboot flash srec (Loads the SREC file stored in FLASH using the nsave command. Note that the nsave image was loaded and stored as a raw binary image).

Notes: See Section 6.3.5 for more information on this command.

Persistent configuration: Updates autoboot information in FLASH.

6.3.2.4.5 boot

Boots the current autoboot image. The current setup configured with the *aboot* command will be loaded and started.

Command syntax:
boot

6.3.2.5 Other commands

6.3.2.5.1 memtst

Runs one or more memory tests on the specified range of memory. A specified range can be cached or uncached memory with a *width* of 1 (bytes), 2 (half-words), or 4 (words). Using a *test* value of *all* will run all the tests. A test will fail and return the failed address, actual value, and expected value on the first failed data occurance.

Test values for the *test* argument are shown below:
0 – waling '1' test
1 – walking '0' test
2 – inverse address test
3 – Pattern (0xAA, 0x55, 0xA5, 0x5A) test
4 – Pattern (0x00, 0xFF) test

Command syntax:
Memtst [hex add] [bytes] [width] [test]

Examples:
Memtst 0x80000000 102400 4 0 (runs test 0 on words at address with size)
Memtst 0x94000000 67108864 1 all (runs all tests on bytes at address with size 64M)

6.3.3 Loading and executing files

6.3.3.1 Image types

The stage 1 loader supports 2 types of file: raw binary and S-record files. The *load* and *aboot* commands handle one of these file types based on the arguments used with those commands.

Raw binary files are absolute raw data that is loaded at an address. There is no extra data or structures in the file that identifies where the file is loaded or executed. Most build tools can generate a file of this type. Because these files don't have information on where they are loaded or executed, the load address and execution address need to be manually provided when the image is loaded. A binary file always loads in a contiguous data region using memory starting at its load address. Loading a binary image from FLASH and an SD/MMC card is error free, but loading an image of this type via the terminal may have errors.

S-Record files are text files formatted in the Motorola S-record format. These files consist of records of text that define an address and data. Another record type defines the execution address. S-records also provide checksums for each record, so any errors that may occur during a terminal load can be detected. S-records may or may not be loaded in a contiguous data range and are well suited for images with multiple load regions.

Regardless of the file type that is loaded, it is tagged as a raw image once in internal memory. Table 9 shows the supported file load options.

Table 9 Support image load options

File type load options		
	S-record	Raw binary
FLASH	Yes[1]	No[3]
Terminal	Yes[2]	Yes[2]
SD/MMC card	Yes	Yes

[1]Data stored in FLASH can only be a raw image. Data loaded from FLASH can be any file type.
[2]Files transferred to the board via the terminal are not guaranteed error free. Use an SD/MMC card for error free operation.
[3]Raw binary downloads of images saved in FLASH are not supported. If a raw binary image is loaded from FLASH using the *load* command, the load and execution in the *load* command arguments are not used. The saved values are used instead.

6.3.3.2 Loading an image

The *load* command can be used to load an image from FLASH, an SD or MMC card, or the terminal. The context and function of the command changes slightly depending on the load source and file type used. The command syntax is shown below:
load [src <file>] [ty] <hex addr 1> <hex addr 2>

6.3.3.2.1 Image load sources

The source (src) can be the *term* (terminal), *blk* (clock device - SD or MMC card), or *flash*.

If the source is *blk*, a filename must be included after the *blk* source and before the file type *ty*. The filename is in an 8.3 format and must be located in the root directory of the SD/MMC card. Long file names are not supported. The *ls* command can be used to list the files on the card.
Example: *load* ***blk*** *lcd.bin 0x80000000*

If the source is *flash*, then an image saved with *nsave* will attempt to be loaded.
Example: *load* ***flash*** srec

If the source is *term*, the command prompt will disappear and a message will appear requesting download via the terminal. The image should be sent over as a binary. Free

terminal programs such as teraterm work well. The transfer can be cancelled by sending a BREAK from the terminal program to the stage 1 loader.
Example: *load* ***term*** bin

6.3.3.2.2 Image types and load command arguments

The image type (ty) can be *raw* (raw binary) or *srec*(S-record) file.

If the image type is *raw*, one or two hex addresses must be provided after the image type. The first address is required and specifies where to load the image in memory. Memory will be loaded as a contiguous data region from the specified load address in the first address field. The second address is optional and specifies the execution address. If the second address isn't provided, the load address will be used as the execution address.
Example: *load term* ***raw 0x80000000 0x80001880***

If the image type is *srec*, the load and execution addresses are not used as they are already specified in the S-record file. S-records without an execution address record in the file will not load successfully. S-record files may be loaded in multiple, non-contiguous data regions.
Example: *load flash* ***srec***

Loaded image information can be examined using the info command.

6.3.3.3 Executing a loaded file

Once an image is loaded, it can be executed using the *exec* command. The address for the start of execution can be examined with the *info* command. Execution at a specific address can be forced be providing an optional hex address after the *exec* command.

6.3.3.3.1 Completion of an executed image

The executed image can exit and the command prompt will be restarted. For this to work correctly, the loaded image must return the system to its pre-run state.

6.3.4 Saving files in FLASH

6.3.4.1 Files types that can be saved in FLASH

Only raw binary images can be stored in FLASH. Regardless of the file type loaded with the *load* command, the file is a raw image once in memory and can be saved in FLASH with the *nsave* command if the data for the loaded image is contiguous.

Along with the saved image in FLASH, information is stored in the serial EEPROM that identifies the saved FLASH image such as load and execution addresses and image size. This information is needed to retrieve the image later with the *nload* command.

6.3.4.1.1 Contiguous vs. non-contiguous files

Files with multiple load regions are not contiguous in memory and can't be programmed into FLASH. S-record files may be non-contiguous. If a non-contiguous file needs to be stored in FLASH, it should be loaded as a raw binary using the load command and then saved in FLASH with *nsave*. The load command can be used with the FLASH source to load the file as its native file type.

Example of storing an S-record file in FLASH:
load term raw 0x80000000
Send the S-record file to the stage 1 loader as a raw binary
nsave
load flash srec
The file is loaded from FLASH as an S-record. Notice the nload command is not used
exec
The loaded file is started

6.3.5 Setting up autoboot

Autoboot allows an image on FLASH, an SD or MMC card, or loaded through the terminal to automatically load and execute. This is setup with the *aboot* command.

6.3.5.1 Autoboot timeout and load failures

The autoboot capability will not start until the autoboot timeout has expired. This timeout is in seconds and is set with the *prompt* command. A minimum of 1 second is required. The timeout allows a key to be pressed on the terminal to override the autoboot sequence and get the monitor program prompt.

After autoboot timeout and if no key on the terminal has been pressed, an image load attempt will be made. If the load attempt fails, the monitor program prompt is displayed instead.

6.3.5.2 Auto-booting raw images from FLASH

Raw images saved with the *nsave* command can be autobooted. Note that a raw image is an image loaded with the *load <src> raw* command and may be a image file type of raw binary or S-record. If the file type is raw, the load and execution addresses used with the aboot command are not used – the saved FLASH image values are used instead.

6.3.6 Load, save, and boot examples

6.3.6.1 Terminal->S-record->FLASH

An S-record file is loaded as a raw image from the terminal, saved in FLASH, and then autobooted from FLASH.

Step 1: Image is loaded as a raw image into SDRAM
load term raw 0x80000000

Step 2: Image is saved into FLASH
nsave

Step 3: Autoboot is setup to load an S-record from FLASH
aboot flash srec

Step 4: Autoboot timeout is set to 1 second.
prompt lpc3250> 1

On the next power cycle or reset, the S-record will be loaded from FLASH and execution will start automatically after 1 second.

6.3.6.2 SD/MMC card->raw binary

A raw binary image stored in the root directory of an SD/MMC card is loaded and autobooted.

Step 1: Place image in the root directory of the SD/MMC card (example name: img.bin)
Step 2: Autoboot is setup to load a raw binary image from the SD/MMC card.
aboot blk img.bin raw 0x80000000

Step 3: Autoboot timeout is set to 5 seconds.
prompt lpc3250> 5

On the next power cycle or reset, the raw binary image will be loaded from the SD/MMC card and execution will start automatically after 5 seconds.

6.3.6.3 Terminal->raw binary->FLASH

A raw binary image is loaded from the terminal, saved in FLASH, and then autobooted from FLASH.

Step 1: Image is loaded as a raw image into SDRAM
load term raw 0x80000000 0x80001000

Step 2: Image is saved into FLASH
nsave

Step 3: Autoboot is setup to load a raw image from FLASH. The first address is required, but not used. The address used with the load command will be used instead when the autoboot occurs.
aboot flash raw 0x12345678

Step 4: Autoboot timeout is set to 1 second.
prompt lpc3250> 1

On the next power cycle or reset, the raw image will be loaded from FLASH and execution will start automatically after 1 second. Image is loaded at address 0x80000000 and execution starts at address 0x80001000.

7 Build process and examples

This section explains how to build an example image, how to change configuration for an image type, and how to deploy it to an LPC32x0 based board. Several examples are explained in detail for specific tasks.

All the examples in this section assume the GNU compiler is being used. You can also use the Realview or Keil compilers by changing the tool option in step 7.1.1.

7.1 Make based build environment

The build environment for this BSP is based on command line driven makes files. The build system is intended to be hosted on a Windows based machine and has been tested on Windows XP, Vista, and Windows 7 based systems.

CodeSourcery GNU, ARM Realview, and Keil MDK toolchains have all been tested.

7.1.1 Setting up the build environment

To setup the build environment, first open a cmd shell (cmd.exe) in windows and set the current working directory to where the CDL software is located. It is recommended that the CDL be installed in the c:/nxpmcu/software directory, but may be installed anywhere.

Once in the correct directory, run the setenv.bat file with one of the following tool options:

CodeSourcery GNU compiler
setenv generic_32x0 gnu 1

ARM Realview 3.1 compiler
setenv generic_32x0 rvw 1

Keil MDK compiler (3.x, 4.x)
setenv generic_32x0 keil 1

If the tools have been installed correctly for the selected toolchain, nothing else is needed. If an error message appears, you may need to verify your tool installations or alter paths in the setenv.bat file. The results of the operations above are shown below:

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

*C:\temp>cd *

C:\>cd nxpmcu

C:\nxpmcu>cd software

C:\nxpmcu\software>setenv generic_32x0 gnu 1
Base LPC install dir : C:\nxpmcu\software
Base LPC unix dir : C:/nxpmcu/software
Extra Tool install dir : C:\nxpmcu\software\tools
Selected CSP : lpc32xx
Selected BSP : generic_32x0
Selected toolchain : gnu

C:\nxpmcu\software>

7.1.2 Test build an example

In the command window, switch to the generic_32x0 directory and go to the large block NAND kickstart burner example. Test your tool chain by building the example. Type 'make' and the example will start building. At the completion of the build, your should have a number of new files; object files, binary, S-record, elf/axf. If any error messages, appear something is not configured probably with the tools.

All examples can be built using the same approach. A shortened version of the build output is shown below:

…
…
arm-none-eabi-ar: creating C:\nxpmcu_new\software\lpc\lib\liblpcarm926ej-sgnu.a
make[1]: Leaving directory `C:/nxpmcu_new/software/lpc/source'
arm-none-eabi-gcc kickstart_nand_lb_burn.o uart.o write_protect_disable.o clo
*ck_setup.o gpio_setup.o mem_setup.o mmu_setup.o board_init.o sdram_common.o ddr_*
lp_sdram.o ddr_st_sdram.o sdr_sdram.o cache_support.o startup_entry.o -static -
Wl,--start-group C:\nxpmcu_new\software\csps\lpc32xx\lib\liblpc32xxgnu.a C:\nxpm
cu_new\software\csps\lpc32xx\bsps\generic_32x0\lib\libgeneric_32x0gnu.a C:\nxpmc
u_new\software\lpc\lib\liblpcarm926ej-sgnu.a -lgcc -lc -lg -lm -lstdc++ -lsupc++
-Wl,--end-group -Xlinker -Map -Xlinker burner_kickstart_nand_large_block_gnu.
*map *
-Xlinker -T C:\nxpmcu_new\software\csps\lpc32xx\bsps\generic_32x0\star
tup\examples\buildfiles\ldscript_iram_gnu.ld -o burner_kickstart_nand_large_bl
ock_gnu.elf
*arm-none-eabi-objcopy -I elf32-littlearm -O binary --strip-all --verbose burner_*
kickstart_nand_large_block_gnu.elf burner_kickstart_nand_large_block_gnu.bin
copy from `burner_kickstart_nand_large_block_gnu.elf' [elf32-littlearm] to `burn
er_kickstart_nand_large_block_gnu.bin' [binary]
*arm-none-eabi-objcopy -O srec --strip-all --verbose burner_kickstart_nand_large_*
block_gnu.elf burner_kickstart_nand_large_block_gnu.srec
copy from `burner_kickstart_nand_large_block_gnu.elf' [elf32-littlearm] to `burn

er_kickstart_nand_large_block_gnu.srec' [srec]

```
C:\nxpmcu_new\software\csps\lpc32xx\bsps\generic_32x0\startup\examples\Burners\n
and_lb\kickstart>dir
Volume in drive C is AWS_System
 Volume Serial Number is E849-4998

 Directory of C:\nxpmcu_new\software\csps\lpc32xx\bsps\generic_32x0\startup\exam
ples\Burners\nand_lb\kickstart

05/18/2010  01:35 PM    <DIR>          .
05/18/2010  01:35 PM    <DIR>          ..
05/18/2010  11:49 AM               644 .depend
05/18/2010  01:35 PM               952 board_init.o
05/18/2010  01:35 PM            22,184 burner_kickstart_nand_large_block_gnu.bin
05/18/2010  01:35 PM            71,435 burner_kickstart_nand_large_block_gnu.elf
05/18/2010  01:35 PM            41,429 burner_kickstart_nand_large_block_gnu.map
05/18/2010  01:35 PM            61,176 burner_kickstart_nand_large_block_gnu.srec
05/18/2010  01:35 PM               752 cache_support.o
05/18/2010  01:35 PM             1,620 clock_setup.o
05/18/2010  01:35 PM             1,748 ddr_lp_sdram.o
05/18/2010  01:35 PM             1,832 ddr_st_sdram.o
05/18/2010  01:35 PM             1,056 gpio_setup.o
05/13/2010  05:11 PM             8,630 kickstart_nand_lb_burn.c
05/18/2010  01:35 PM             3,332 kickstart_nand_lb_burn.o
05/18/2010  11:53 AM             1,184 makefile
05/18/2010  01:35 PM             1,040 mem_setup.o
05/18/2010  01:35 PM             1,440 mmu_setup.o
05/18/2010  01:35 PM             3,632 sdram_common.o
05/18/2010  01:35 PM             1,468 sdr_sdram.o
05/18/2010  01:35 PM             1,728 startup_entry.o
05/18/2010  01:35 PM             1,308 uart.o
05/18/2010  01:35 PM               952 write_protect_disable.o
              21 File(s)        229,542 bytes
               2 Dir(s)  11,828,703,232 bytes free
```

7.1.3 Cleaning up code

The ‘make clean’ command can be used to clean the object files and other build output files. The entire distribution can be cleaned by running the distclean batch file in the c:/nxpmcu/software area.

7.1.4 Altering build options

Build options can be altered by altering the compiler and linker options in the c:/nxpmcu/software/makerules/lpc32xx area. For the specific toolchain being used (gnu, keil, Realview), the associated make file with that toolchain needs to e changed.

Compiler optimizations can be changed and tested here to try to get smaller (or faster) images. Be sure to do a distclean before rebuilding any code after the make options are changed.

7.2 Startup code compilation flags

The inclusion of board MMU setup, data relocation, or stack code is controlled by define flags passed to the compiler during the build process. Because of this, any changes to these compiler flag option requires the code to be cleaned (make clean) before rebuilding. Although predefined options for the inclusion of code related to these flags has already been made in the makefiles, you may want to change these options for a custom build.

In the makefiles for specific examples, 4 specific flags are defined to control these options.

USE_MMU
If this definition is passed to the compiler, MMU setup code will be included in the startup code. This will setup the MMU page table and caches, but will add some size to the image.

USE_BOARD_INIT
If this definition is passed to the compiler, board setup code will be included in the startup code. This will setup the GPIO states and muxing, clocking, and SDRAM, but will add some size to the image.

USE_ALL_STACKS
If this definition is passed to the compiler, all the ARM mode stacks will be setup in the startup code. This is useful for systems that require interrupt support, but it increases the image size (by just a little). It also requires more IRAM. Stacks sizes are defined in the setup.inc and setup_gnu.inc files.

RW_RELOC
If this definition is passed to the compiler, code to relocate the RW data segmwnt is added to the startup code. This is required for systems that boot FROM NOR FLASH.

For any project that uses the startup code; this includes the kickstart loaders, burner software, and S1L; the makefiles associated with that project contain the defines to include or exclude code based on these flags. To include the code for a specific flag, just uncomment the line with the flag in the makefile. To disable the code, comment the line in the makefile.

The following segment below shows the portion of a makefile that controls startup code configuration for the Realview compiler. For this specific case, MMU setup and data relocation support is disabled. MMU setup and board setup code are enabled.

```
ifeq ($(TOOL), rvw)

# S1L runs in IRAM and with SDRAM, board init, and all stacks
#AFLAGS += --predefine "USE_MMU SETL {TRUE}"
AFLAGS += --predefine "USE_BOARD_INIT SETL {TRUE}"
AFLAGS += --predefine "USE_ALL_STACKS SETL {TRUE}"
#AFLAGS += --predefine "RW_RELOC SETL {TRUE}"

# S1L runs from IRAM
LDSCRIPT =$(BASE_BSP)\startup\examples\buildfiles\ldscript_iram_rvw.ld

endif
```

7.3 Build and deployment examples

The startup code and boot loaders included in this BSP provide a number of different configuration options that can be used as a basis for various LPC32x0 board designs. The following 3 examples describe 3 unique boards; build options and the process associated with those boards, and reasoning for selecting those build options is explained. Each example also shows how the images were deployed to the board.

7.3.1 Board with DDR mobile SDRAM and large block Micron NAND FLASH with plans to run Linux

This board type supports large block Micron NAND FLASH with 16-bit DDR SDRAM memory. This board provides 1 possible boot option for the chip; NAND FLASH.

As we plan on running Linux on this board, the u-boot boot loader should be the stage 1 application we intend on running. The normal size of u-boot exceeds 54Kbytes (by a factor of 2 or 3), so we'll need a smaller initial boot loader to bring up the board and then load and start u-boot. The u-boot boot loader had no board specific code in it for initialization of SDRAM, and u-boot runs from SDRAM, so the initial boot loader must also setup SDRAM. Requirements and limitations for the design are shown below:

- Boots from large block NAND
- Desired application to boot is u-boot
  - u-boot had no SDRAM initialization code
  - u-boot (usually) runs in SDRAM
- Initial boot loader must:
  - Boot from large block NAND FLASH
    - Must not exceed 54K in size
    - Must load and run in IRAM

- Setup SDRAM (and other board/CPU functions)
- Load u-boot from large block NAND FLASH
  - Loads and runs in SDRAM
- Start u-boot

The desired NAND partitioning scheme for this setup would be:

- Block 0 (128K. 54K max usable) : kickstart loader
- Blocks 1 – 4 (512K total): u-boot with some spare area for bad blocks
- Blocks 5 and on : Dedicated to Linux kernel and root file system

Based on these requirements and limitations, the best starting point would be to use the large block NAND kickstart loader with the board initialization code enabled. The board init code will setup GPIO and mux states, setup system clock rates used by Linux and u-boot, and initialize SDRAM and memory interfaces. This kickstart loader could initially be programmed into NAND block 0 using the large block NAND kickstart burner application. u-boot (built outside this package using the Linux tools) could be stored in blocks 1 – 4 using the large block NAND stage 1 burner application. (After u-boot has been initially programmed into the device, it can be used to update itself).

The following applications are needed for this configuration:

kickstart loader – large block NAND FLASH
kickstart burner – large block NAND FLASH
stage 1 burner – large block NAND FLASH
u-boot.bin (built from Linux tools)

7.3.1.1 Changes to build files

Some files may need to be customized to support this configuration. The following sections explain possible changes that may be needed.

7.3.1.1.1 Code and driver files

The large block NAND MLC and SLC drivers are located in the ./generic_32x0/source area and are required for the burner and kickstart loader. As the drivers are based on Micron NAND FLASH already, the required changes are probably very small. At a minimum, the NAND IDs that need to be checked may need to be added to the nand_init() function. Addressing may also need to be changed.

The write_protect_disable.c file in the ./generic_32x0/startup/examples/common area may also need a change to the nand_flash_wp_disable() disable function based on how the NAND write protect is implemented on the board.

For DDR mobile support, the defines specific to enabling DDR support and configuring the DDR device timing need to be changed in the setup.h and dram_configs.h files. The

default system clock speeds are also setup in setup.h. In any specific board muxing is required; the gpio_setup.c file should also be changed. To get the fastest NAND FLASH timings (which effects boot time and load times), the timing parameters for NAND FLASH in the board_config.h file should be optimized.

7.3.1.1.2 Changes to build files

Since the kickstart loader should initialize the board interfaces and SDRAM, the makefile associated with the kickstart loader should have the following options enabled and disabled.

```
#AFLAGS += --predefine "USE_MMU SETL {TRUE}"
AFLAGS += --predefine "USE_BOARD_INIT SETL {TRUE}"
#AFLAGS += --predefine "USE_ALL_STACKS SETL {TRUE}"
#AFLAGS += --predefine "RW_RELOC SETL {TRUE}"
```

Note that u-boot setups up its own stacks and disables the MMU on startup, so enabling these options will only make the kickstart image larger than needed. The relocation option doesn't apply here, as NOR FLASH isn't being used.

The default build configuration of the burner software is with all options disabled. This will require the burner software to load the image to burn into IRAM. Because the u-boot image can get fairly large, it would be easier to use SDRAM for storage of the images to burn into FLASH because there are no size restrictions. Change the options as follows for the kickstart loader and stage 1 burner applications.

```
#AFLAGS += --predefine "USE_MMU SETL {TRUE}"
AFLAGS += --predefine "USE_BOARD_INIT SETL {TRUE}"
#AFLAGS += --predefine "USE_ALL_STACKS SETL {TRUE}"
#AFLAGS += --predefine "RW_RELOC SETL {TRUE}"
```

7.3.1.1.3 Changes to code configuration files

Since u-boot boots into SDRAM, the load address for the stage 1 application needs to be changed to SDRAM. The size is also changed to 256K, to allow kickstart to load an image size of 256K into SDRAM. These options are changed in the misc_config.h file in the ./generic_32x0/startup/examples/common area.

```
#define STAGE1_LOAD_ADDR 0x83fc0000 /* Last 256K of SDRAM */
#define STAGE1_LOAD_SIZE 0x40000 /* Load 256k */
```

Kickstart will now load the stage 1 application with size 256Kbytes starting in NAND FLASH block 1 into SDRAM at address 0x83fc0000 when it runs.

The burner software board init option was enabled in the previous section so SDRAM is being initialized as part of the burner software, but the location where the burner software

loads the image to program into NAND FLASH still needs to be specified. This option is changed in the misc_config.h file in the ./generic_32x0/startup/examples/common area.

```
#define BURNER_LOAD_ADDR 0x80000000 /* Start of SDRAM */
```

This same define will be used with the kickstart loader burner and stage 1 burner applications and allows very large images to be loaded with the Serial Loader tool.

7.3.1.2 Build and deployment

To build the 2 burner applications and the kickstart loader application, the build environment needs to be setup for the specific toolchain as per Section 7.1.1. Then run make in the directory for the burner and kickstart loader applications to get the binaries.

To burn the kickstart loader using the Serial loader tool, setup the primary boot file with the kickstart burner binary. Setup the Secondary boot file with the kickstart loader binary. Transfer the files to the board (See Section 5.1.2) and verify they were programmed.

To burn u-boot using the Serial loader tool, setup the primary boot file with the stage 1 burner binary. Setup the Secondary boot file with the u-boot binary. Transfer the files to the board (See Section 5.1.2) and verify they were programmed.

If everything worked correctly, the board can then be reset and u-boot will start.

7.3.2 Board with SDR standard SDRAM, small block Samsung NAND FLASH, and SPI FLASH

This board type supports small block Samsung NAND FLASH and SPI FLASH with 32-bit SDR SDRAM memory. This board provides 2 possible boot options for the chip; NAND FLASH or SPI FLASH.

Although no specific application has been specified to run on this board, we can assume the application will run in SDRAM and will be larger than 54K. A kickstart loader will be required regardless of whether NAND or SPI FLASH boot is used. However, the choice of the boot method (SPI or NAND) will likely affect how the kickstart loader is configured. Regardless of whether the kickstart loader boots from SPI or NAND FLASH, the stage 1 application will (probably) have to boot from NAND FLASH, unless a very large SPI FLASH device is used. Requirements and limitations for the design are shown below:

- Boots from either small block NAND or SPI FLASH
- Desired application to load and start in SDRAM
- Initial boot loader must:
    - If booting from small block NAND FLASH
        - Must not exceed 15.5K in size

        - Kickstart loader will need heavy optimizations if using board initialization code
    - If booting from SPI FLASH
        - Must not exceed 54K in size
    - Setup SDRAM (and other board/CPU functions)
    - Load stage 1 application from small block NAND FLASH
        - Loads and runs in SDRAM
    - Start application

The desired small block NAND partitioning scheme for this setup would be:

- Block 0 (15.5K max) : kickstart loader with board initialization code
- Blocks 1 and on : Stage 1 application without board initialization code

The desired SPI FLASH partitioning scheme for this setup would be:

- Start of SPI FLASH (15.5K) : kickstart loader with board initialization code
- NAND FLASH blocks 1 and on : Stage 1 application without board initialization code

Using the standard startup code and kickstart variants, the small block kickstart loader with the board initialization and MMU setup code will probably be over the 15.5K boot image size limit for small block NAND FLASH boot. So if small block NAND FLASH boot is used, the stage 1 application needs to handle board initialization. This could be a problem if the stage 1 application needs to execute from SDRAM and will require some work to heavily optimize the kickstart loader to build and fit within 15.5K.

If the SPI FLASH is used as the boot device, the kickstart loader with board initialization code and MMU setup code should be under 54K. This would be the easier approach in this case, but would use some of the SPI FLASH storage for boot code.

The following applications are needed for this configuration:

Stage 1 burner – small block NAND FLASH
Stage 1 application (binary)
kickstart loader – small block NAND FLASH

The following applications are needed for this configuration if using small block NAND FLASH for the kickstart loader:

kickstart burner – small block NAND FLASH

The following applications are needed for this configuration if using SPI FLASH for the kickstart loader:

kickstart burner – SPI FLASH

7.3.2.1 Changes to build files

Some files may need to be customized to support this configuration. The following sections explain possible changes that may be needed.

7.3.2.1.1 Code and driver files

The small block NAND SLC driver is located in the ./generic_32x0/source area and are required for the stage 1 application small block NAND burner and the kickstart loader. If the kickstart loader will be booted from NAND FLASH, then the small block NAND MLC driver may also need changes. At a minimum, the NAND IDs that need to be checked may need to be added to the nand_init() function. Addressing may also need to be changed.

The write_protect_disable.c file in the ./generic_32x0/startup/examples/common area may also need a change to the nand_flash_wp_disable() disable function based on how the NAND write protect is implemented on the board.

For SDR standard SDRAM support, the defines specific to enabling standard SDR support and configuring the SDR device timing need to be changed in the setup.h and dram_configs.h files. The default system clock speeds are also setup in setup.h. In any specific board muxing is required; the gpio_setup.c file should also be changed. To get the fastest NAND FLASH timings (which effects boot time and load times), the timing parameters for NAND FLASH in the board_config.h file should be optimized.

7.3.2.1.2 Changes to build files

Since the kickstart loader should initialize the board interfaces and SDRAM, the makefile associated with the kickstart loader should have the following options enabled and disabled.

```
AFLAGS += --predefine "USE_MMU SETL {TRUE}"
AFLAGS += --predefine "USE_BOARD_INIT SETL {TRUE}"
#AFLAGS += --predefine "USE_ALL_STACKS SETL {TRUE}"
#AFLAGS += --predefine "RW_RELOC SETL {TRUE}"
```

The USE_MMU define can be optionally disabled if cache and MMU support isn't needed. The kickstart loader does need all the stacks, so it can remain disabled. The relocation option doesn't apply here, as NOR FLASH isn't being used.

The default build configuration of the burner software is with all options disabled. This will require the burner software to load the image to burn into IRAM. Because the stage 1 application is probably fairly large, it would be easier to use SDRAM for storage of the images to burn into FLASH because there are no size restrictions. Change the options as follows for the kickstart loader and stage 1 burner applications.

```
#AFLAGS += --predefine "USE_MMU SETL {TRUE}"
```

```
AFLAGS += --predefine "USE_BOARD_INIT SETL {TRUE}"
#AFLAGS += --predefine "USE_ALL_STACKS SETL {TRUE}"
#AFLAGS += --predefine "RW_RELOC SETL {TRUE}"
```

7.3.2.1.3 Changes to code configuration files

Since the stage 1 application boots into SDRAM, the load address for the stage 1 application needs to be changed to SDRAM. The size is also changed to the actual size of the image (larger is ok).. These options are changed in the misc_config.h file in the ./generic_32x0/startup/examples/common area. The example below shows how to setup the kickstart loader for a 4MByte image loaded into the state of SDRAM (address 0x80000000).

```
#define STAGE1_LOAD_ADDR 0x80000000 /* Start of SDRAM */
#define STAGE1_LOAD_SIZE (4 * 1024 * 1024) /* Load 4M */
```

Kickstart will now load the stage 1 application with size 4Mbytes starting in NAND FLASH block 1 into SDRAM at address 0x80000000 when it runs.

The burner software board init option was enabled in the previous section so SDRAM is being initialized as part of the burner software, but the location where the burner software loads the image to program into NAND FLASH still needs to be specified. This option is changed in the misc_config.h file in the ./generic_32x0/startup/examples/common area.

```
#define BURNER_LOAD_ADDR 0x80000000 /* Start of SDRAM */
```

This same define will be used with the kickstart loader burner and stage 1 burner applications and allows very large images to be loaded with the Serial Loader tool. *As the size of the image loaded via the Serial Loader tool is very large (4MB), it may make sense to use another option for burning the images into FLASH.*

7.3.2.2 Build and deployment

To build the 2 burner applications (stage 1 burner and either the small block NAND or SPI FLASH burner) and the kickstart loader application, the build environment needs to be setup for the specific toolchain as per Section 7.1.1. Then run make in the directory for the burner and kickstart loader applications to get the binaries.

To burn the kickstart loader using the Serial loader tool, setup the primary boot file with the kickstart burner binary. Setup the Secondary boot file with the kickstart loader binary. Transfer the files to the board (See Section 5.1.2) and verify they were programmed.

To burn the stage 1 application using the Serial loader tool, setup the primary boot file with the stage 1 burner binary. Setup the Secondary boot file with the stage 1 application

binary. Transfer the files to the board (See Section 5.1.2) and verify they were programmed.

If everything worked correctly, the board can then be reset and the stage 1 application will start.

7.3.3 Using u-boot on the Phytec 3250 board without S1L

The Phyetc 3250 board, as configured from Phytec, boots the kickstart loader from small block NAND FLASH block 0. The kickstart loader loads and starts S1L in IRAM from NAND FLASH block 1. Because the kickstart loader must be under 15.5K, the board initialization code is not included in the kickstart loader. Instead, it is included in S1L. This makes routing applications such as u-boot via S1L a requirement as u-boot runs in SDRAM.

However, u-boot can be made to boot directly from the kickstart loader with a few changes as listed below:

Add board initialization code to kickstart loader
For the kickstart loader, change the makefile to include the board initialization code (SDRAM, clock, stc.) by un-commenting out the line below. The clean and rebuild the project.

```
AFLAGS += --predefine "USE_BOARD_INIT SETL {TRUE}"
```

Change the load address and size for the kickstart loader
In the misc_config.h file, changes the load and address size to address 0x83FC0000 in SDRAM and a size of 256KBytes. u-boot may be smaller than 256K, but it's ok to let the kickstart loader load the full 256K.

```
#define STAGE1_LOAD_ADDR 0x83FC0000 /* u-boot run address */
#define STAGE1_LOAD_SIZE 0x40000 /* Load 256K */
```

Optimize the small block NAND FLASH version of the kickstart loader
Try different compiler optimization levels and edit the board init code to get the size of the kickstart loader as executing from NAND FLASH under 15.5K. This can be done without too many changes on the Realview and Keil toolchains, but may require a little work with GNU.

Once the changes have been made, rebuild the kickstart loader to get a new binary with the new load address and size and included board initialization code.

Build the SPI FLASH burner and burn the kickstart loader into SPI FLASH

The SPI FLASH burner is used to burn the kickstart loader into SPI FLASH. Build the SPI FLASH burner and use the Serial loader tool with the SPI FLASH burner and the kickstart loader to burn the kickstart loader into SPI FLASH.

Build the stage 1 application burner and burn u-boot into small block NAND FLASH
The stage 1 application burner is used to burn an application into small block NAND FLASH at block 1 and on. Build the stage 1 application burner and use the Serial loader tool with the stage 1 application burner and the u-boot binary loader to burn u-boot into block 1 and on of small block NAND FLASH.

After the images are programmed, the board should boot u-boot when the board is reset. The boot sequence is shown in the following steps:

- LPC32x0 chip is reset and code in LPC320 internal boot ROM starts executing
- SPI FLASH is found as a valid boot method by boot ROM
- Kickstart loader at start of SPI FLASH is loaded into IRAM at address 0x0
- Boot ROM transfers control to kickstart loader at address 0x0
- Kickstart loader initializes board; this includes SDRAM, clocking, GPIO, and memory
- Optionally, the kickstart loader sets up the MMU page table and enables the MMU
- Kickstart loader loads u-boot starting from block 1 into SDRAM at address 0x83FC0000. It loads 256Kbytes.
- Kickstart loader transfer control to u-boot loader at address 0x83FC0000

The following space is used in SPI FLASH and small block NAND FLASH:

SPI FLASH:

- Start of SPI FLASH is used for kickstart loader

Small block NAND FLASH:

- Block 0 is available
- Blocks 1 to ? are used for u-boot storage
- All other blocks are available

8 Notes & information

8.1 Special notes about u-boot, eboot, and S1L

The Linux boot loader, u-boot, and the WinCE boot loader, eboot, do not contain any code to initialize the board SDRAM interfaces. Because u-boot and eboot run in SDRAM, they require at least the kickstart loader to initialize the board to a known state before loading and executing.

Phytec boards are shipped with the kickstart loader and S1L pre-installed in small block NAND FLASH. The kickstart loader basically loads S1L into IRAM and then starts it without initializing SDRAM. S1L then initializes the board, MMU, and SDRAM. S1L in this configuration uses no resources from SDRAM; all of its code and data and the generated MMU page table are in IRAM. Because of this approach and the fact the u-boot and eboot run in SDRAM, S1L must run prior to u-boot or eboot when booting from small block NAND FLASH. S1L does have facilities for loading u-boot and eboot.

Here is the boot scenario in the case of small block NAND FLASH:

- LPC32x0 boot ROM loads kickstart loader image (smaller than 15.5K) from block 0 into IRAM at address 0x0
- Kickstart loader loads S1L image from block 1 and on into IRAM at address 0x8000. *Older versions of kickstart relocated to the end of IRAM and the S1L image was loaded at address 0x0 – this is no longer the case.*
- S1L initializes board, clocks, SDRAM
- S1L provides an optional monitor interface or can be setup for autoboot of another image
- S1L loads u-boot or eboot into SDRAM and then transfers control to the loaded image

The use of S1L is fine, but can be bypassed in systems that boot from large block FLASH, NOR FLASH, or SPI FLASH. For these system types, the kickstart loader can be up to 54K in size, allowing the board, SDRAM, clock, and MMU setup code to part of the kickstart loader. S1L is completely optional for these systems, as the kickstart loader can now boot any image size to any location in memory.

Phytec 3250 board users can use alternate versions of the kickstart loader that run from SPI FLASH or NOR FLASH that allow direct load and start of u-boot or eboot without being routed through S1L. Note that the kickstart loaders load address and size may need to be changed to support this. A NOR flash version of S1L can also be used that doesn't require a kickstart loader.

8.2 Various image sizes using different toolchains

This section shows estimated images sizes created with the GNU, Keil, and Realview toolchains for different startup and board configurations. Although these sizes are from compiled code, these should be used as an estimate only to get an idea of the size of your build images.

8.2.1 CodeSourcery GNU compilers

The applications below were built using the CodeSourcery GNU compiler version 2009q3-68. The following compiler optimization levels are shown in the full compiler argument command line below:

```
arm-none-eabi-gcc -c -mcpu=arm926ej-s -Wall -Os -mno-sched-
prolog -fno-hosted -mno-thumb-interwork
```

Note: Size can be reduced slightly by removing the –g option. Note that for debug code is enabled for all builds – this can be disabled macking the image smaller by commenting out the `ENABLE_DEBUG` macro in the setup.h file.

Table 10 Image sizes generated with the GNU compiler

Application	Code+data size	
S1l built for IRAM	63084	MMU code, board code, all stacks, small block NAND FLASH driver, standard SDRAM
S1l built to boot from kickstart	60536	All stacks, small block NAND FLASH driver, standard SDRAM
S1L built for boot from kickstart, but with board and MMU code	63084[1]	MMU code, board code, all stacks, small block NAND FLASH driver, standard SDRAM
S1L built to run from NOR FLASH		MMU code, board code, all stacks, data relocation code, small block NAND FLASH driver, standard SDRAM
Kickstart loader for loading image from large block NAND		MMU code, board code, large block NAND FLASH driver, standard SDRAM
Kickstart loader for loading image from small block NAND		Small block NAND FLASH driver
Kickstart loader for loading image from small block NAND		MMU code, board code, small block NAND FLASH driver, standard SDRAM
Kickstart loader for loading image from SPI FLASH		MMU code, board code, SPI FLASH driver, standard SDRAM

[1]This is the same as the IRAM version.

8.2.2 Realview compilers

The applications below were built using the ARM Realview Development Suite version 3.0. The following compiler optimization levels are shown in the full compiler argument command line below:

```
armcc -c -O2 -OTime --cpu=ARM926EJ-S -g
```

Note: Size can be reduced slightly by removing the –g option. Note that for debug code is enabled for all builds – this can be disabled macking the image smaller by commenting out the `ENABLE_DEBUG` macro in the setup.h file.

Table 11 Image sizes generated with the Realview compiler

Application	Binary image size (bytes)	Build options
S1l built for IRAM	62552	MMU code, board code, all stacks, small block NAND FLASH driver, standard SDRAM
S1l built to boot from kickstart	60072	All stacks, small block NAND FLASH driver, standard SDRAM
S1L built for boot from kickstart, but with board and MMU code	62552[1]	MMU code, board code, all stacks, small block NAND FLASH driver, standard SDRAM
S1L built to run from NOR FLASH	62596	MMU code, board code, all stacks, data relocation code, small block NAND FLASH driver, standard SDRAM
Kickstart loader for loading image from large block NAND	17168	MMU code, board code, large block NAND FLASH driver, standard SDRAM
Kickstart loader for loading image from small block NAND	10008[2]	Small block NAND FLASH driver
Kickstart loader for loading image from small block NAND	17180[3]	MMU code, board code, small block NAND FLASH driver, standard SDRAM
Kickstart loader for loading image from SPI FLASH	16436	MMU code, board code, SPI FLASH driver, standard SDRAM

[1]This is the same as the IRAM version.
[2]The image size is well under the 15.5K small block NAND boot size limit. This version doesn't initialize the board and MMU, so the stage 1 application the kickstart loader loads must do these tasks.
[3]The image size is over the 15.5K small block NAND boot size limit and must be trimmed for small block FLASH boot. Removing the MMU setup code and disabling debug might help, but it will still be very close to the 15.5K limit. One area that can be optimized is the clock setup function in the ./startup/board/clock_setup.c file.

8.2.3 Keil compilers

This information was not yet available at the time of this document's publication. As the Keil MDK is based on Realview, the Realview sizes should be very similar with similar optimizations.

8.3 Board and driver timing values

If the program time or NOR boot time is taking too long, then the timing values for the NAND controllers or the NOR interface may need to be optimized. Select values in the ./include/board_config.h file to get the fastest possible timing for the currently configured bus speed.

8.4 WinCE and reserved block marking

If NAND FLASH support is enabled in the WinCE build, WinCE may attempt to partition and format NAND FLASH for its use. This will cause the kickstart and stage 1 loader to be erased, as well as any stored data in NAND FLASH.

To prevent this from happening, any NAND FLASH blocks that shouldn't be touched by WinCE should be marked as reserved. This is done by tagging (setting low) a specific location in the spare area of the first page in a NAND block.

See the FLASH driver included with the LPC32x0 WinCE port for information on which bytes and bit location to tag. Blocks marked with this tag will not be erased by WinCE. Note that erase commands will reset the reserved block markers.

8.5 Special S1L support for SDRAM testing

If the debug code is enabled, special support code will be added to S1L to help with SDRAM testing. This special code provides examination of the SDRAM configuration values (and is very helpful to NXP support staff) and altering several SDRAM functions. The SDRAM configuration information is shown when the S1L info command is used.

Several added commands also provide some additional SDRAM testing.

The *rosci* command is used to test ring oscillator stability. When the *rosci* command is executed on a DDR based system, the ring oscillator calibration is forced for a large number of cycles and the value monitored. Large variations of this value may indicate a possible problem with DDR hardware.

The *rbswap* command is provided to cycle between SDRAM low power and performance modes. This allows dynamic switching of the SDRAM address scheme to allow either high performance (more access) or lower power (less performance) access. You can try them both to see which one better suits your better.

8.6 Known issues with S1L

8.6.1 MMUENAB command sometimes hangs

Avoid using the *mmuenab* command, as it is unreliable when disabling the MMU. If it needs to be used, first flush the data caches, disable the data cache, and then disable the MMU in that order using the appropriate commands.

8.6.2 INFO shows image loaded in memory on ABOOT failure

If the autoboot setup fails to boot for some reason (for example, when autoboot is setup to boot from an SD card and an SD card is not inserted), the *info* command will show that an image has been loaded into memory. This issue only occurs when an autoboot occurs from an SD card and the SD card is not inserted or the file is not found on the SD card.

8.7 Building S1L to boot from NOR FLASH

S1L can also be built to boot from NOR FLASH. When S1L is built to boot from this FLASH type, the S1L executable is modified in the following way:

- Code links for execution on NOR address space (0xE0xxxxxx) on CS0
- Code entry address is 0xE0000004
- Code is execute-in-place (XIP) directly from NOR
- Read-write data (zero and pre-initialized) is still in IRAM starting at address 0x08001000
- An extra data move is required to move pre-initialized RW data to IRAM

An example link map is shown below of the generated image. Note that the address of code entry is 4 bytes past the start of NOR FLASH. The LPC32XX uses the first 4 bytes for a key value used by the boot ROM for NOR FLASH boot.

```
Image Entry point : 0xe0000004

  Load Region FLASH (Base: 0xe0000004, Size: 0x0000f948, Max:
0xffffffff, ABSOLUTE)
    Execution Region ER_RO (Base: 0xe0000004, Size: 0x0000e1bc, Max:
0xffffffff, ABSOLUTE)
    Execution Region ER_RW (Base: 0x08001000, Size: 0x0000178c, Max:
0xffffffff, ABSOLUTE)
    Execution Region ER_ZI (Base: 0x0800278c, Size: 0x0000358c, Max:
0xffffffff, ABSOLUTE)
```

Once an image is built, the NOR FLASH loader tool (included with the LPC32XX CSP) can be used to burn the image into BOR FLASH for boot. The NOR FLASH loader will append the correct key value to address 0xE0000000 and burn the image into FLASH.

The kickstart loader used with NAND FLASH isn't needed when booting from NOR FLASH.