PRELOADER USER GUIDE

Rev. 0.9.3

26 Dec. 2012

Realtek Semiconductor Corp.

No. 2, Innovation Road II, Hsinchu Science Park, Hsinchu 300, Taiwan

Tel.: +886-3-578-0211 Fax: +886-3-577-6047

www.realtek.com

USING THIS DOCUMENT

This document is intended for use by the system engineer when integrating with Realtek switch products. Though every effort has been made to assure that this document is current and accurate, more information may have become available subsequent to the production of this guide. In that event, please contact your Realtek representative for additional information that may help in the development process.

Revision	Release Date	Author	Summary
0.9	2012/12/07	Jia-Jhe Li	● First draft.
0.9.1	2012/12/10	Jia-Jhe Li	● Removed platform-specific information.
0.9.2	2012/12/16	Jia-Jhe Li	● Introduced a section for "parameters" structure. ● Introduced a chapter for Composer. ● Reorganized chapter sequence.
0.9.3	2012/12/26	Jia-Jhe Li	● Refined Composer.

Table of Contents

1 Introduction....5
2 Building Preloader....6
3 Configuring Preloader....7
3.1 Inside "soc.tcl"....7
3.2 Inside "parts_for_nor_spi.tcl"....8
3.3 Inside "parts_for_dram_gen2.tcl"....10
4 Software Patch Mechanism....14
5 Directory Arrangement of Preloader....15
6 Control Flow of Preloader....16
6.1 Data Structure....17
6.2 CPU and Cache....17
6.3 UART Controller....18
6.4 Memory Controller....18
6.5 NOR SPI Flash Controller....18
7 Composer....20

1 Introduction

Preloader is a concise bootloader developed by Realtek. Its purpose is to initialize SoC related components and starts other applications, such as OS and U-Boot. These SoC related components include CPU, cache, UART, memory controller, PLL controller, NOR SPI flash, and NAND flash. Preloader also implements a set of APIs for controlling these components, and exposes these APIs to other applications.

To ease the overhead of modifying these initializations and drivers of SoC components from time to time, Preloader characterizes these components with a text-based configuration file called soc.tcl. With such a centralized configuration file, bootloader and kernel can optionally leverage it for a better portability. In build time, a tool which comes with Preloader called *otto_composer* encodes soc.tcl to a binary form and integrates it with all images to conduct the final image, as Figure 1 indicates. In other words, by modifying soc.tcl, user can tune the behavior of Preloader, bootloader, and kernel without modifying and building them from source again.

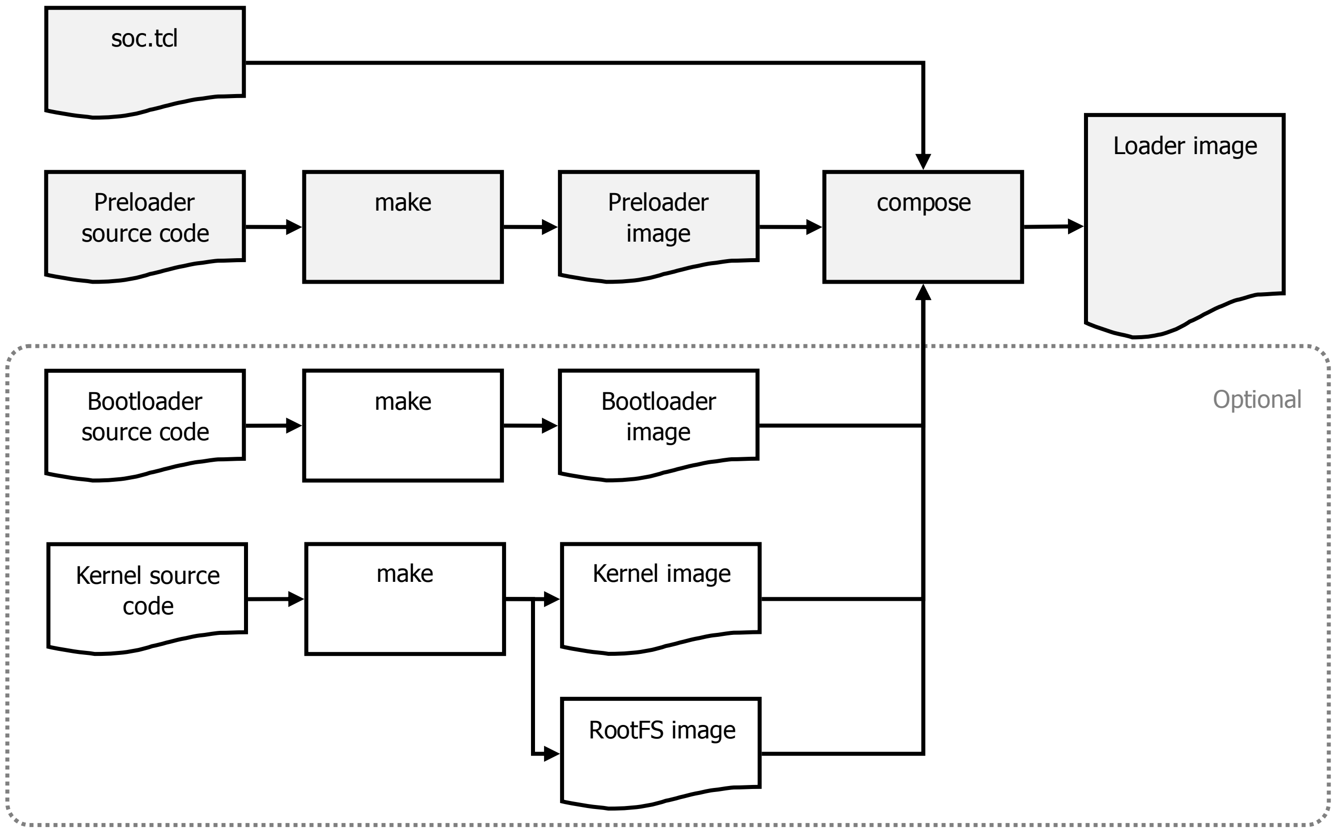


Figure 1: Preloader build flow

2 Building Preloader

Assume that \$(preloader) points to the path of Preloader source, one can conduct image of Preloader with following steps:

1. Refer \$(preloader)/toolkit_path.in.sample to create a \$(preloader)/toolkit_path.in, so that

   `CROSS_COMPILE` is the prefix to your toolkits (i.e., \$(CROSS_COMPILE)gcc refers to your C compiler),

   `BOOTLOADER_IMG` points to your bootloader image to put to loader image,

   `KERNEL_IMG` points to your kernel image to put to loader image, and

   `ROOTFS_IMG` points to your root file system image to put to loader image.

   Note that only `CROSS_COMPILE` is necessary, others are optional.

2. Select a target platform:

   ```
   $ make preconfig_<PlatformName>
   ```

   In this point, \$(preloader)/platform/current is created and symbolically links to \$(preloader)/platform/<PlatformName>[1].

3. To build Preloader, one can apply following command under \$(preloader):

   ```
   $ make all
   ```

   It conducts \$(preloader)/release/plr.img.

4. Build \$(BOOTLOADER_IMG), \$(KERNEL_IMG), and \$(ROOTFS_IMG) if you have assigned any.

5. Tweak \$(preloader)/release/soc.tcl with your favorite text editor if it's necessary. Please refer Chapter 3 for the detail of soc.tcl configuration.

6. To conduct the final loader image, apply following command under \$(preloader):

   ```
   $ make compose
   ```

   This step combines Preloader image with (if any) images of Bootloader, Kernel, and rootfs.

7. Finally, you can find \$(preloader)/release/loader.img. It is the image you need to boot a system.

8. To examine the configuration integrated in \$(preloader)/release/loader.img, apply following command under \$(preloader):

   ```
   $ make dumpcfg
   ```

[1] Please consult your contact window for the exact platform names available.

3 Configuring Preloader

3.1 Inside “soc.tcl”

```
source $PARTS_DB_DIR/parts_for_nor_spi.tcl
add_model nor_spi WINBOND_W25Q32BV
add_model nor_spi MXIC_MX25L12845E

source $PARTS_DB_DIR/parts_for_dram_gen2.tcl
add_model dram_gen2 SAMSUNG_K4B1G1646G-BCH9

namespace eval rtk_pll_gen2 {
    variable set_by         pin
    variable cpu_clock_mhz  50
    variable dram_clock_mhz 25
    variable lx_clock_mhz   32
}
add_model pll_gen2 rtk_pll_gen2
namespace eval rtk_peri {
    variable uart_baudrate  115200
}
add_model peri rtk_peri
namespace eval default_flash_layout {
    variable layout [subst {
        { follow    bootloader1 "$bootloader_image" }
        --runtime
        { 256K      env         -reserve 16K }
        { 1M        kernel1     -reserve 3M }
        { follow    rootfs1     -reserve 4M+512K }
    }]
}
add_model flash_layout default_flash_layout
variable flash_type        spi_nor
variable padding_unit      4K
```

Figure 2: An example of soc.tcl.

Figure 2 shows a typical soc.tcl. Bold texts are the major configurations that Preloader exposes.

In this example, line 1 includes a pre-built NOR SPI flash database, `parts_for_nor_spi.tcl`. It is another text-base file containing configurations of all verified NOR SPI flashes. The file will be discussed in the later chapter.

After including the database, line 2 selects a specific NOR SPI flash model for the target platform. As line 3 indicates, there can be multiple "add_model" commands. Note that, though there are lots of verified NOR SPI flash model in the database, only the selected models will be recognized by Preloader in run time. While the number of selected models is unlimited to Preloader, please note that each inclusion increases the size of Preloader.

Line 4 and 5 is similar to line 1 ~ 3, except they are used to select the DRAM configurations from a DRAM database, `parts_for_dram_gen2.tcl`. The file will be discussed in the later chapter.

Line 8 ~ 11 configure the clock rates for different components. Line 8 decides to change clock rates during run time or not. If it is "**pin**", Preloader skips the PLL configuration in run time, and system runs at the default frequency. However, if it is configured at "**software**", Preloader applies the desired clock rates of CPU, DRAM, and LX bus given in line 9 ~ 11 to PLL controller during booting sequence. Note that, even if "pin" is chosen (the default frequency), one still has to configure line 9 ~ 11 to reflect the default clock rates. Preloader uses them to calculate several timing information for SoC components, such as UART and memory controller.

Line 15 gives the desired baud rate. Preloader combines this value with lx_clock_mhz to configure UART controller.

Line 20 ~ 24 organize the flash layout as illustrated by Figure 3. There are two parts in flash layout: image part (line 20) and run time part (line 22 ~ 24). The two parts are separated by the keyword "**--runtime**" as line 21. Image part configures how Preloader conducts the image, while run time part helps Preloader to recognize the other data stored in flash during run time. Furthermore, other applications, such as U-Boot and Linux kernel, can leverage this information for a better integrity. In this example, line 20 tells Preloader that it should combine U-Boot image to conduct the final image. The keyword "**follow**" indicates that U-Boot image should be attached right after Preloader's (with a padding length given in line 30, which is 4K). One can also use an "**absolute offset**" instead of "**follow**" to configure flash layout. For example, line 22 indicates that the environment parameters is stored at the position (*flash base* + 256K), and line 23 tells that Linux kernel is stored at (*flash base* + 1M). There is an optional keyword, "**-reserve**". It stands for the size of a section. For example, line 22 means that environment data is stored at the range between (*flash_base* + 256K) to (*flash_base* + 256K + 16K). To simplify the address calculation, one can also use an expression instead of a specific value to configure address, as indicated by line 24.

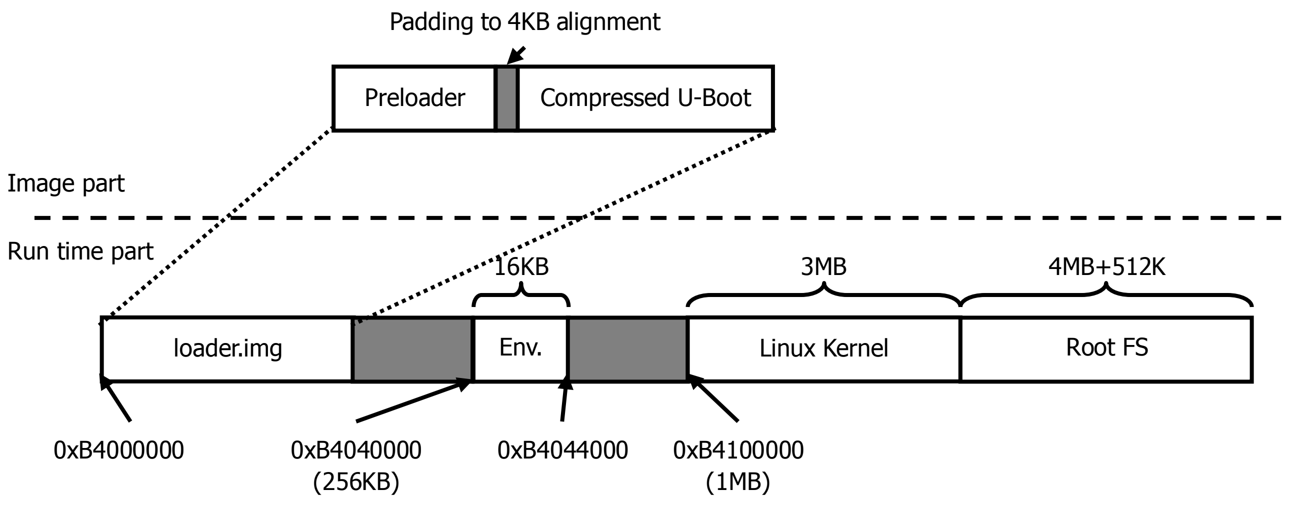


Figure 3: Example of Preloader and NOR SPI flash layout of 錯誤! 找不到參照來源。**.**

3.2 *Inside "parts_for_nor_spi.tcl"*

This file is a text-based database keeping necessary information for NOR SPI flash driver of Preloader to work properly. To add a new NOR SPI flash to the database, one has to edit this database, and provide necessary parameters. Figure 4 shows an example from a model in the database. One can add more models basing on this example.

For each model, one has to provide a model name as line 0 indicates. In this example, it is MXIC's MX25L12845E. The "add model" command in soc.tcl uses this name to include the correspondence parameters. The suggested naming convention is "manufacturer_model".

Line 1 sets the number of NOR SPI flash chips on board.

Line 2 sets the clock rate divisor related to memory clock for this NOR SPI flash chip. Since clock rate of NOR SPI flash relates to memory clock, if you have to change memory clock rate, please also verify this divisor. Please refer your NOR SPI flash spec for a proper clock rate. An overloaded clock rate may cause malfunction of NOR SPI flash.

Line 3 sets the size per NOR SPI flash chips.

Line 4 ~ 7 tweak the delays of RX signals.

Line 9 ~ 13 set parameters related to READ operation of NOR SPI flash. Since a NOR SPI flash chip could support different modes of READ operations, please refer you NOR SPI flash spec to fill the correct READ command, command IO mode, address IO mode, data IO mode, and number of dummy cycle. **Note!** If the desired READ command requires switching to QIO mode, please also sets line 25 ~ 31 for a proper switching method.

Similar to line 9 ~ 13, line 15 ~ 20 set parameters for PROGRAM operation. The only difference is line 20, **wr_boundary**, due to Most NOR SPI flashes can only write a limited size of data to flash for each PROGRAM command. While the common constraint is 256 bytes, please refer your NOR SPI flash spec for the correct value.

Line 22 ~ 23 configure ERASE operation. While most NOR SPI flashes support sector erase, block erase, and chip erase operations, user has to decide which erase operation that Preloader should use and the number of byte that the operation erases.

Line 25 ~ 31 are for enabling QIO mode of NOR SPI flash. Since different manufacturers require different methods to enable QIO, user has to set **pm_method** to make Preloader works. Currently, **pm_method** supports following options:

1. NONE: This option indicates that there is no need to switch to QIO mode. In this case, parameters in line 26 ~ 31 are ignored.
2. RWSR: Some NOR SPI flashes enable QIO mode by setting some bits in a status register to 1s. In this case, one has to provide **pm_rdsr_cmd**, **pm_wrsr_cmd**, **pm_enable_bits**, and **pm_status_len**. Where pm_rdsr_cmd and pm_wrsr_cmd are commands for NOR SPI flash, pm_enable_bits is a bit mask, pm_status_len is the number of byte of status. In run time, Preloader first issues pm_rsdr_cmd to get pm_status_len-bytes of status, masks it with pm_enable_bit to enable the QIO bits, and then writes the result to NOR SPI flash with pm_wrsr_cmd. Usually, MXIC's NOR SPI flash uses this method.
3. CMD: Some NOR SPI flashes enter QIO mode after explicitly received a special command. In this case, please configure **pm_enable_cmd** correctly. Other parameters are ignored. During initialization of NOR SPI flash, Preloader will issue the given command to enable QIO mode. Some EON's flashes utilize this method.
4. R2W1SR: This option is similar to RWSR, except that NOR SPI flashes read 16-byte status information from 2 status registers (8-byte status each), activate the QIO bit, and write back once to these status registers. In this case, user has to fill **pm_rdsr_cmd**, **pm_rdsr2_cmd**, **pm_wrsr_cmd**, and **pm_enable_bits**. In run time, Preloader first issues pm_rdsr_cmd and pm_rdsr2_cmd to retrieve status information. Note that Preloader keeps status from pm_rdsr_cmd to bit 15 ~ 8, and the one from pm_rdsr2_cmd to bit 7 ~ 0. After that, Preloader will `OR' pm_enable_bit to status, and then issue pm_wrsr_cmd to write back the updated status.

In our best knowledge, MXIC utilizes RWSR method, EON uses CMD method, and Winbond uses R2W1SR method. However, please refer your NOR SPI flash spec for the exact method. **Note!** If all above methods does not fit your NOR SPI flash, please contact us for technical support.

Line 33 ~ 36 defines how Preloader queries the availability of a NOR SPI flash. Where **rdbusy_cmd** stands for the command that Preloader uses, **rdbusy_len** is the number of bytes that rdbusy_cmd returns, **rdbusy_loc** is the position of busy bit, and **rdbusy_polling_period** is the minimal interval time in micro-second between two rdbusy_cmd. During each ERASE and PROGRAM operations, Preloader will issue rdbusy_cmd continuously to check whether the bit in rdbusy_loc is 1 or not. Since some NOR SPI flashes cannot handle such a burst of polling, Preloader may wait couple micro-seconds between the issuing of two rdbusy_cmd.

Finally, line 38 defines the ID of a NOR SPI flash. During initialization, Preloader probes ID of the NOR SPI flash on board, and traverses all selected (by "add_model" command in soc.tcl) for a matching configuration.

```
namespace eval MXIC_MX25L12845E {
    variable num_chips          1
    variable prefer_divisor     16
    variable size_per_chip      16M
    variable prefer_rx_delay0   0
    variable prefer_rx_delay1   0
    variable prefer_rx_delay2   0
    variable prefer_rx_delay3   0

    variable prefer_rd_cmd      0xeb
    variable prefer_rd_cmd_io   SIO
    variable prefer_rd_dummy_c  6
    variable prefer_rd_addr_io  QIO
    variable prefer_rd_data_io  QIO

    variable wr_cmd             0x02
    variable wr_cmd_io          SIO
    variable wr_dummy_c         0
    variable wr_addr_io         SIO
    variable wr_data_io         SIO
    variable wr_boundary        256

    variable erase_cmd          0x20
    variable erase_unit         4K

    variable pm_method          RWSR
    variable pm_rdsr_cmd        0x05
    variable pm_rdsr2_cmd       0x00
    variable pm_wrsr_cmd        0x01
    variable pm_enable_cmd      0x00
    variable pm_enable_bits     0x40
    variable pm_status_len      1

    variable rdbusy_cmd         0x05
    variable rdbusy_len         0x2
    variable rdbusy_loc         0
    variable rdbusy_polling_period  1

    variable id                0xc22018
}

namespace eval EON_EN25Q64 {
    :
    :
```

Figure 4: An example of NOR SPI flash model in parts_for_nor_spi.tcl.

3.3 Inside "parts_for_dram_gen2.tcl"

Similar to parts_for_nor_spi.tcl, this is a text-base database of verified DRAM chips. **Note!** Since memory controller (MEMCTL) evolves from time to time, different platforms may be built with different generation of MEMCTLs. As a result, the necessary parameters for configuring a MEMCTL may then be different. Table 1 summarizes the equipped MEMCTL of different SoC models.

Since there is only 8380 test chip utilizes Gen.1 MEMCTL, and its DRAM configuration is hard-coded, Preloader does **NOT** provide a database for Gen.1 MEMCTL.

Figure 5 and Figure 6 extract Nanya's NT5CB128MHP from the database for an example. To add a new model, simply use it as a template for revision, and append it to this database.

For each model, one has to provide a model name as line 0 indicates. In this example, it is Nanya's NT5CB128MHP. The "add model" command in soc.tcl uses this name to include the correspondence parameters. The suggested naming convention is "manufacturer_model".

Line 1 ~ 2 configure the instruction and data prefetch feature in MCR register of the Gen.2 MEMCTL. One can enable this feature by setting this option to 1.

Line 3 ~ 7 set bank number, data width, row number, and column number of a DRAM chip, respectively. Please refer you DRAM spec for the correct values. An inappropriate setting causes either wrong DRAM size determination or system fails to boot.

Line 8 configures the number of DRAM chips in system. Set it to 0 if only one DRAM chip on board; set it to 1 if there are two DRAM chip on board.

Line 9 and 10 control the FAST_RX and BSTREF feature in DCR register of Gen.2 MEMCTL. 1 for activation, and 0 for inactivation.

Line 12 ~ 21 define the timing constraint of the given DRAM chip. Please refer your DRAM spec for proper values.

Line 23 ~ 124 fill correspondence registers of MEMCTL directly. Which cover several MEMCTL registers, including MPMR0, MPMR1, DIDER, D23OSCR, DACCR, DACSPCR, DACSPSR, DACDQ0RR ~ DACD15RR, DACDQ0FR ~ DACD15FR, and DCDR.

Line 126 decides to use auto DRAM size detection mechanism or not. This mechanism is still under development, so please keep it disable.

Line 127 is allowed to choose a DRAM calibration method. Its goal is to initiate DACDQ0RR ~ DACD15RR, DACDQ0FR ~ DACD15FR, and DCDR. It can be **software** or **static**. If "software" is selected, Preloader applies an algorithm to calibrate DRAM signals during boot, and fills DACDQ0RR ~ DACD15RR, DACDQ0FR ~ DACD15FR, and DCDR. It will slightly increase boot time. On the other hand, selecting "static" omits the calibration procedure. In this case, one has to supply DACDQ0RR ~ DACD15RR, DACDQ0FR ~ DACD15FR, and DCDR explicitly.

Line 129 controls the auto calibration mechanism of MEMCTL. The goal of this options is to continuously tune DACDQ0RR ~ DACD15RR, DACDQ0FR ~ DACD15FR, and DCDR and against temperature and noise for system stability after these registers are initiated. However, this feature is still under development so please keep it disable.

zq_calibration and zq_impedance can enable/disable ZQ calibration with the given ZQ impedance.

drv_strength sets driving strength to **normal** or **reduced**.

mrs_dll_enable, mrs_drv_strength, mrs_odt, and mrs_additive_latency affect related options in mode registers of DRAM chips. Please refer your DRAM spec for detail.

Memory Controller Generation	**SoC Model**
Generation 1	8380 test chip
Generation 2	8380 real chip, 8390, 8686

Table 1: Memory controller generation of different SoC models.

```
namespace eval NANYA_NT5CB128MHP {
    variable IPREF              0
    variable DPREF              0

    variable BANKCNT            2
    variable DBUSWID            1
    variable ROWCNT             3
    variable COLCNT             2
    variable DCHIPSEL           0
    variable FAST_RX            0
    variable BSTREF             0

    variable refi_ns            7800
    variable rp_ns              15
    variable rcd_ns             15
    variable ras_ns             45
    variable rfc_ns             328
    variable wr_ns              15
    variable rrd_ns             10
    variable fawg_ns            50
    variable wtr_ns             8
    variable rtp_ns             8
#For MPMR0 register
    variable PM_MODE            0x0
    variable T_CKE              0xf
    variable T_RSD              0x3ff
    variable T_XSREF            0x3ff
#For MPMR1 register
    variable T_XARD             0xf
    variable T_AXPD             0xf
#For DIDER register
    variable DQS0_EN_HCLK        0x0
    variable DQS0_EN_TAP         0x0
    variable DQS1_EN_HCLK        0x0
    variable DQS1_EN_TAP         0x0
#For D23OSCR register
    variable ODT_ALWAYS_ON       0x0
    variable TE_ALWAYS_ON        0x0
#For DACCR register
    variable AC_MODE            0x1
    variable DQS_SE             0x1
    variable DQS0_GROUP_TAP      0x0
    variable DQS1_GROUP_TAP      0x0
    variable AC_DYN_BPTR_CLR_EN   0x0
    variable AC_BPTR_CLEAR       0x1
    variable AC_DEBUG_SEL        0x0
#For DACSPCR register
    variable AC_SILEN_PERIOD_EN   0x0
    variable AC_SILEN_TRIG       0x0
    variable AC_SILEN_PERIOD_UNIT  0x0
    variable AC_SILEN_PERIOD      0x0
    variable AC_SILEN_LEN        0x7f
#For DACSPSR register
    variable AC_SPS_DQ15R         0
    variable AC_SPS_DQ14R         0
    variable AC_SPS_DQ13R         0
    variable AC_SPS_DQ12R         0
    variable AC_SPS_DQ11R         0
    variable AC_SPS_DQ10R         0
    variable AC_SPS_DQ9R          0
    variable AC_SPS_DQ8R          0
    variable AC_SPS_DQ7R          0
    variable AC_SPS_DQ6R          0
    variable AC_SPS_DQ5R          0
    variable AC_SPS_DQ4R          0
    variable AC_SPS_DQ3R          0
    variable AC_SPS_DQ2R          0
    variable AC_SPS_DQ1R          0
    variable AC_SPS_DQ0R          0
    variable AC_SPS_DQ15F         0
    variable AC_SPS_DQ14F         0
    variable AC_SPS_DQ13F         0
    variable AC_SPS_DQ12F         0
    variable AC_SPS_DQ11F         0
    variable AC_SPS_DQ10F         0
    variable AC_SPS_DQ9F          0
    variable AC_SPS_DQ8F          0
    variable AC_SPS_DQ7F          0
    variable AC_SPS_DQ6F          0
    variable AC_SPS_DQ5F          0
    variable AC_SPS_DQ4F          0
    variable AC_SPS_DQ3F          0
    variable AC_SPS_DQ2F          0
    variable AC_SPS_DQ1F          0
    variable AC_SPS_DQ0F          0
#For DACDQ(0/8)RR~DACD(Q7/15)RR and
#DACDQ(0/8)FR~DACD(Q7~15)FR registers
    variable static_cal_data_0     0x0f110401
    variable static_cal_data_1     0x0f130501
    variable static_cal_data_2     0x0f100401
    variable static_cal_data_3     0x0f130501
    variable static_cal_data_4     0x0f130501
    variable static_cal_data_5     0x0f100401
    variable static_cal_data_6     0x0f030501
    variable static_cal_data_7     0x0f100401
    variable static_cal_data_8     0x00000001
    variable static_cal_data_9     0x00000001
    variable static_cal_data_10    0x00000001
    variable static_cal_data_11    0x00000001
    variable static_cal_data_12    0x00000001
    variable static_cal_data_13    0x00000001
    variable static_cal_data_14    0x00000001
    variable static_cal_data_15    0x00000001
    variable static_cal_data_16    0x00110401
```

Figure 5: An example of DRAM model in parts_for_dram_gen2.tcl.

```
    variable static_cal_data_17     0x00130501
    variable static_cal_data_18     0x00000001
    variable static_cal_data_19     0x00130501
    variable static_cal_data_20     0x00030501
    variable static_cal_data_21     0x00100401
    variable static_cal_data_22     0x00130501
    variable static_cal_data_23     0x00100401
    variable static_cal_data_24     0x00000001
    variable static_cal_data_25     0x00000001
    variable static_cal_data_26     0x00000001
    variable static_cal_data_27     0x00000001
    variable static_cal_data_28     0x00000001
    variable static_cal_data_29     0x00000001
    variable static_cal_data_30     0x00000001
    variable static_cal_data_31     0x00000001
    variable static_cal_data_32     0x10100000
#For DCDR register
    variable TX_CLK_PHS_DELAY       0x0
    variable CLKM_DELAY             0xa
    variable CLKM90_DELAY           0xf

    variable size_auto_detection    disable
    variable calibration_type       software

    variable auto_calibration       disable

    variable zq_calibration         disable
    variable zq_impedance           50

    variable drv_strength           normal

    variable mrs_dll_enable         dll_enable
    variable mrs_drv_strength       normal
    variable mrs_odt                120
    variable mrs_additive_latency   0
}
```

Figure 6: An example of DRAM model in parts_for_dram_gen2.tcl. (Cont.)

4 Software Patch Mechanism

Software patch mechanism (SWP, for short.) provides a simple way to hook functions into Preloader. With SWP, one can insert their own code into Preloader without knowing how Preloader works. To use this mechanism, one have to create a file with "swp_" as its prefix of file name under $(preloader)/preloader/platform/current. For example, swp_my_patch.c. This chapter uses code in 錯誤! 找不到參照來源。 for an example. The code is quite simple: prints "Preloader rocks." if MY_DEFINE equals 1, otherwise, prints "Amazing Preloader".

```
//OTTO_SWP_CFLAGS=-DMY_DEFINE=1

#include <preloader.h>

PATCH_REG(my_patch_func);
void my_patch_func(void) {
#if (MY_DEFINE == 1)
  printf("Preloader rocks.\n");
#else
  printf("Amazing Preloader.\n");
#endif

  return;
}
```

Figure 7: Example of software patch code.

One can notice that the first line is comment line as

```
//OTTO_SWP_CFLAGS=-DMY_DEFINE=1
```

The purpose of this line is to provide a chance to append extra compilation flags for building SWP, in addition to original flags come from Preloader. While the length and position of this line in code is not important, this line must start with "//OTTO_SWP_CFALGS=", otherwise it won't work.

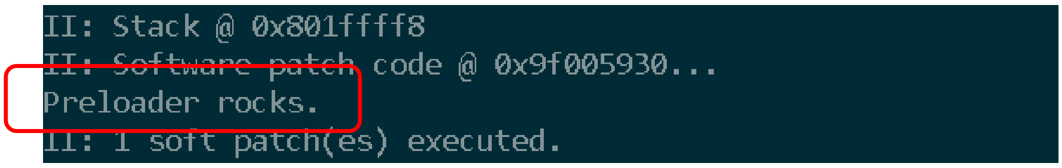
Another key point for SWP is to use PATCH_REG() macro to inform Preloader the entry point of this SWP file. In this example,

```
PATCH_REG(my_patch_func);
```

tells Preloader to start execution of this patch from the function named "my_patch_func". In run time, Preloader traverses all functions registered with PATCH_REG(). Note that the order of different SWP files is not guaranteed.

During execution, Preloader simply jumps to each SWP entry function and returns. Therefore, the prototype of the entry function (my_patch_func() in this example) must be void in both return type and argument. However, functions invoked by the entry function have no such a limitation.

Following is the execution log after applied this SWP file:



```
II: Stack @ 0x801ffff8
II: Software patch code @ 0x9f005930...
Preloader rocks.
II: 1 soft patch(es) executed.
```

5 Directory Arrangement of Preloader

Figure 7 illustrates the directory structure of Preloader. On the top level, there are **"./preloader"**, **"./util"**, and **"./release"**. Where "./preloader" includes the major source code of Preloader, "./util" contains composer and model DB, and "./release" are used to store the conducted Preloader image and header files for other applications that build upon Preloader.

"./preloader" includes **"./platform"** and **"./soc"**, along with platform-independent source files. These platform-independent source files define the generic control flow of Preloader. In contrast, "./platform" contains directories for each platform to keep platform-dependent code. Finally, "./soc" includes drivers of several common IPs, such as CPU, cache, UART, memory controller, PLL controller, NOR SPI flash, and NAND flash. Code under "./platform" can then leverage these drivers to ease the porting overhead.

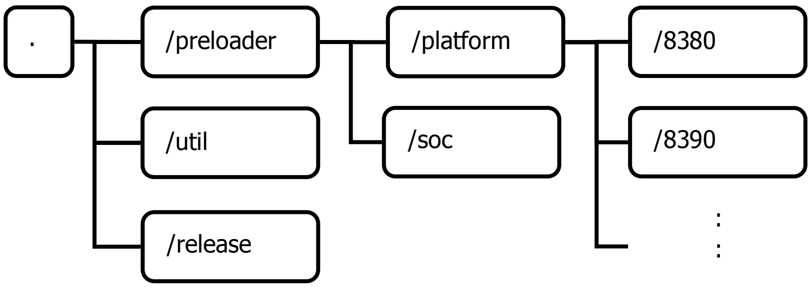


Figure 8: Directory arrangement of Preloader

6 Control Flow of Preloader

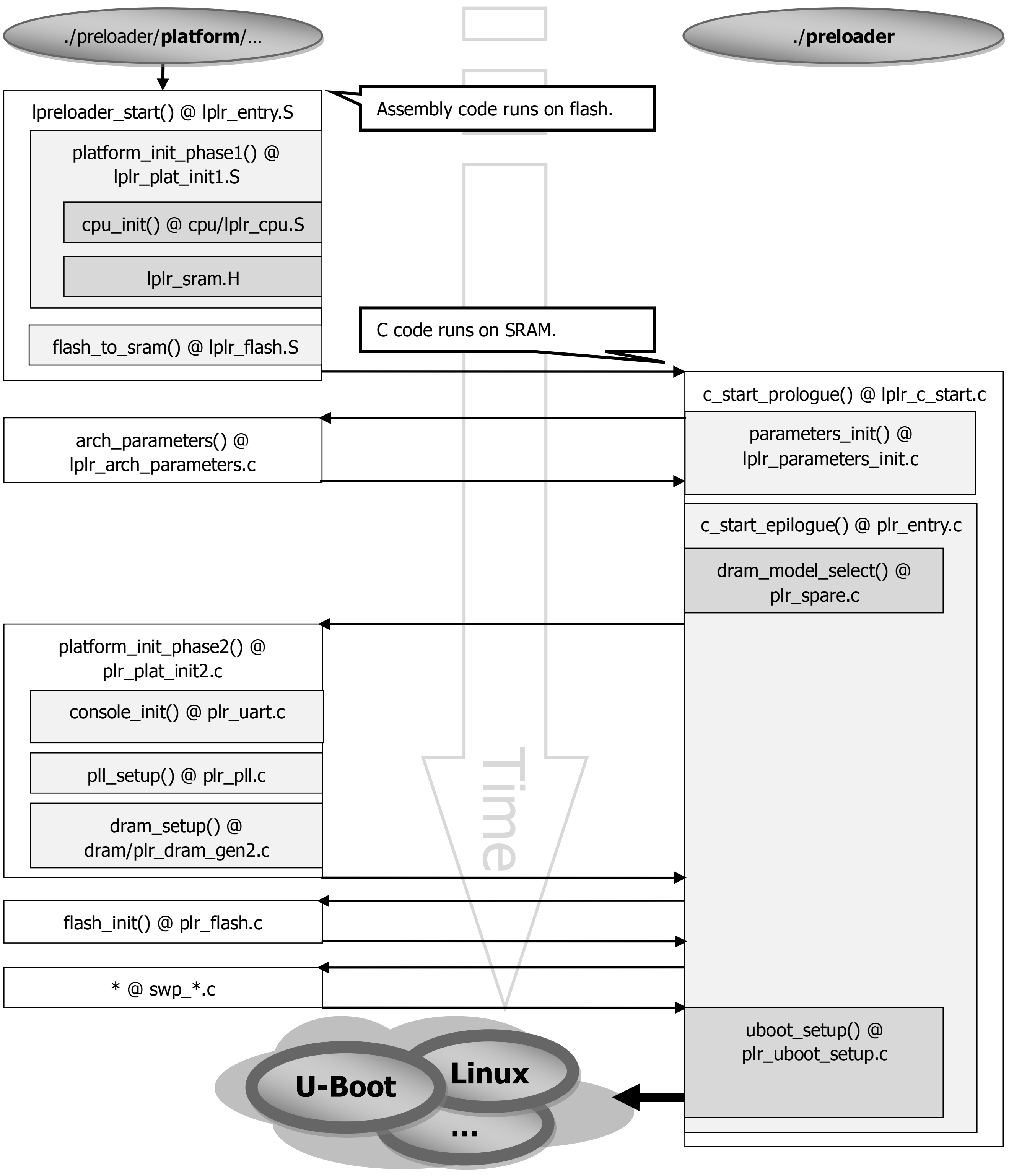

./preloader/**platform**/…
./**preloader**
lpreloader_start() @ lplr_entry.S
Assembly code runs on flash.
platform_init_phase1() @ lplr_plat_init1.S
cpu_init() @ cpu/lplr_cpu.S
lplr_sram.H
C code runs on SRAM.
flash_to_sram() @ lplr_flash.S
c_start_prologue() @ lplr_c_start.c
arch_parameters() @ lplr_arch_parameters.c
parameters_init() @ lplr_parameters_init.c
c_start_epilogue() @ plr_entry.c
dram_model_select() @ plr_spare.c
platform_init_phase2() @ plr_plat_init2.c
console_init() @ plr_uart.c
pll_setup() @ plr_pll.c
dram_setup() @ dram/plr_dram_gen2.c
Time
flash_init() @ plr_flash.c
* @ swp_*.c
uboot_setup() @ plr_uboot_setup.c
U-Boot
Linux
…

Figure 9: A typical control flow of Preloader

Figure 8 shows a typical control flow of Preloader framework. Conceptually, platform-dependent code are under ./preloader/platform, and platform-independent one are in ./preloader immediately.

Each platform includes a `lplr_entry.S' written in assembly code. It contains a serves as an entry point of Preloader. Its main purpose is to initiate and setup CPU, cache, and SRAM. Furthermore, it prepares environment to meet calling convention of C programs. Finally, it copies Preloader from flash to SRAM, and starts a platform-independent function, c_start_prologue(), from SRAM. Note that, before jumping to c_start_prologue(), all programs are written in assembly code and run on flash.

In c_start_prologue(), it setups the internal structure of Preloader, then invokes c_start_epilogue() to initialize UART, DRAM, PLL, and flash.

First of all, c_start_epilogue() invokes dram_model_select() to prepare the desired configuration of DRAM parameters, recall that users can select multiple DRAM models in soc.tcl. Unlike NOR SPI flash, DRAM does not come with any ID information. As a result, it is impossible for Preloader to "probe" DRAM chips for correspondence settings. Therefore **it is user's responsibility to provide dram_model_select() to tell Preloader which set of DRAM parameters should be used.** This function should return an integer as an index of the desired DRAM model. If this function is not provided, Preloader uses the first model for default. In the next step, Preloader invokes console_init(), pll_setup(), dram_setup(), and flash_init() to initialize UART, PLL, DRAM, and flash controllers, respectively. After that, c_start_epilogue() runs all functions declared as "software patch" in swp_*.c. Preloader provides such a mechanism so users can "hook" their code into Preloader without modifying source code of Preloader. This mechanism is discussed in Chapter 7. Finally, c_start_epilogue() invokes uboot_setup() to prepare environment and optionally decompress for bootloader (or kernel), then jump to it.

Generally, users can control those component drivers by editing soc.tcl. For users who are interested in porting these drivers, following chapters highlight some major IPs.

6.1 Data Structure

For the sake of managing these SoC components, Preloader defines a structure, `parameter_to_bootloader_t` (Parameters for short), to centralize these information. Each driver in Preloader accesses Parameters to configure correspondence components, and communicates with other components. One can find this structure in $(preloader)/release/soc.h.

Parameters contains a structure named *soc* (SOC for short). SOC encapsulates attributes of flash, DRAM, PLL, and other peripherals in flash_info, dram_info, pll_info, and peri_info, respectively.

Beyond attributes, Parameters also defines a set of APIs for other applications, e.g., U-Boot, to access. These APIs cover flash, UART, cache, and PLL. To invoke these APIs beyond Preloader, one can include "$(preloader)/release/pblr.h" in source code, and uses the predefined macro "parameters" to access the desired function. For example, to invoke "_nor_spi_erase()", one can use "parametes._nor_spi_erase()" along with proper arguments to erase data on SPI flash. Since Preloader is enhanced continuously, please refer $(preloader)/release/soc.h. for a complete list of APIs.

6.2 CPU and Cache

The major object of this phase is to enable cache and reset several CPU control registers.

For a reference of these fundamental settings, please check

```
cpu_init @ $(preloader)/preloader/platform/current/cpu/lplr_cpu.S
```

for CPU initialization, and

```
mips_cache_reset @ $(preloader)/preloader/platform/current/cpu/lplr_cache.S
```

for cache reset process.

Generally, simply invoke these two assembly functions should do the tricks. However, if more advanced configurations are needed, please refer the manual of CPU for more detail.

6.3 UART Controller

The UART controller in our platform is compatible with NS16550. One can refer

`otto_NS16550_init(int baudrate_divisor)` @ *`$(preloader)/preloader/platform/current/plr_uart.c`*

for an example of setup. Note that the argument of the function is "baud rate divisor", **NOT** "baud rate". Following equation translates baud rate to divisor:

Divisor = [LX Clock / (16 * Baud Rate)] – 1

This driver refers `parameters.soc.peri_info.baudrate_divisor` for baud rate divisor.

6.4 Memory Controller

The object of this phase is to setup memory controller along with DRAM chip parameters, so that memory controller interacts with DRAM chip correctly. Furthermore, this phase applies a software calibration sequence to fine tune timing parameters for better stability.

Our implementation of this phase can be found at

`dram_setup()` @ *`$(preloader)/preloader/platform/current/dram/plr_dram_gen2.c`*[2]

Due to the complexity of DRAM calibration, we strongly recommend that one to leverage our implementation of calibration instead of rewriting one. Furthermore, to ensure the correctness of calibration, we suggest running this code from cache.

This driver refers `parameters.soc.dram_info` to setup memory controller.

6.5 NOR SPI Flash Controller

The default setting of NOR SPI flash controller already provides the ability to do "MMIO read operation". That is, one can read flash from the mapped address. Due to the variety of NOR SPI flash chips, if a more advance I/O is necessary, one have to configure NOR SPI flash controller according to the spec. of the target NOR SPI flash chip. For detail of the register set of NOR SPI flash controller, please refer SFCR(0xb8001200), SFCR2(0xb8001204), SFCSR(0xb8001208), and SFDR(0xb800120c) in SoC spec.

An example of initializing NOR SPI flash controller in our driver can be found at

`flash_init(void)` @ *`$(preloader)/preloader/platform/current/plr_flash.c`*

First of all, the driver sets SFCR. Note, please refer the limitation of the selected NOR SPI flash chip to set SPI_CLK_DIV. An overloaded clock may cause NOR SPI flash to work inappropriately. Next, the driver enables QIO mode of the NOR SPI flash chip. NOR SPI flash chips from different manufactures may need different methods to trigger QIO mode. However, all these methods involve issuing NOR SPI flash commands to flash chip through controller with PIO. A proper sequence to issue a command from flash controller to flash chip can be found at

`spi_cmd()` @ *`$(preloader)/preloader/platform/current/plr_flash.c`*

NOTE, NOR SPI flash controller of RTL8380 does not support QIO.

Finally, the driver sets SFCR2 for flash size, and the desired read command for MMIO. Please refer spec. of the NOR SPI flash chip for proper command IO, address IO, data IO, and dummy cycles.

Furthermore, our driver has encapsulated three common NOR SPI flash operations, erase, write, and read operations, as C functions. Once one provides the desired erase, write, and read commands, these functions help to access NOR SPI flash in a PIO manner along with proper parameters to these operations. A brief of these functions is listed below:

[2] The actual file name depends on the memory controller. However, it always contains a function with the following prototype: `void dram_setup(void)`.

- `int flash_unit_erase(u32_t cid, u32_t offset)`: applies the predefined erase command to the given *offset* of the given *flash chip ID*.
- `int flash_read(u32_t cid, u32_t offset, u32_t len, void *buf)`: reads *len* bytes of data from the given *offset* of the given *flash chip ID* with the predefined read command, and writes them to *buf*.
- `int flash_unit_write(u32_t cid, u32_t offset, u32_t len, void *buf)`: writes *len* bytes of data from *buf* to the given *offset* of the given *flash chip ID* with the predefined write command.

These functions can be found at

`$(preloader)/preloader/platform/current/plr_flash.c`

This driver refers `parameters.soc.flash_info` to setup flash controller.

7 Composer

While soc.tcl provides a human-readable interface of configuration, it is inefficient for Preloader to use such information directly. That is why Composer inbounds. Composer is an essential tool in Preloader framework. It does not only pack relevant images together, but also encode information in soc.tcl into Parameters structure (chapter 6.1). If one decides to port drivers in Preloader, it is a good start to observe how Composer translates human-readable parameters in soc.tcl into a C structure as Parameters.

To see that, after loader.img is conducted, apply following command under $(preloader):

```
$ make dumpcfg
```

Composer will parse the *current* loader.img and show the encoded information, i.e., Parameters, similar to Figure 9 and Figure 10. Note: please make sure that loader.img is runnable before extracting Parameters for porting purpose.

Among this information, one can notice the first few lines indicate the size of correspondence structures.

What after size information are parameters for NOR SPI flash. Note that parameters of NOR SPI flash in an one-to-one mapping between soc.tcl and Parameters.

Following parameters of NOR SPI flash is flash layout. One can see that the human-readable layout setting in soc.tcl is translated to hex-based addresses.

Parameters after flash_layout_info are for memory controller. One can find that most of DRAM parameters are already translated to register-level parameters. For example, bank number, bus width, row and column counts are encoded into *dcr*; timing information such as refi, rp, rcd, ras, rfc, wr, rrd, fawg, wtr, and rtp become *dtr0*, *dtr1*, and *dtr2*. Similarly, PM_MODE, T_CKE, T_RSD, and T_XSREF are encoded to *mpmr0*; T_XARD and T_AXPD are encoded to *mpmr1*; DQS0_EN_HCLK, DQS0_EN_TAP, DQS1_EN_HCLK, and DQS1_EN_TAP are encoded to *dider*; ODT_ALWAYS_ON and TE_ALWAYS_ON are encoded to *d23oscr*; AC_MODE, DQS_SE, DQS0_GROUP_TAP, DQS1_GROUP_TAP, AC_DYN_BPTR_CLR_EN, and AC_DEBUG_SEL are encoded to *daccr*, AC_SILEN_PERIOD_EN, AC_SILEN_TRIG, AC_SILEN_PERIOD_UNIT, AC_SILEN_PERIOD, and AC_SILEN_LEN are encoded to *dacspcr*, AC_SPS_DQ15R ~ AC_SPS_DQ0R and AC_SPS_DQ15F ~ AC_SPS_DQ0F are encoded to *dacspar*. In addition to registers of MEMCTL listed above, Composer also generates proper values for mode registers of DRAM chips. For example, for DDR3, they are *DDR3_mr0*, *DDR3_mr1*, *DDR3_mr2*, and *DDR3_mr3*. The rest parameters, i.e., static_cal_data_0 ~ 32, size_auto_detection, calibration_type, TX_CLK_PHS_DELAY, CLKM_DELAY, CLKM90_DELAY, auto_calibration, zq_calibration, zq_impedance, drv_strength, mrs_dll_enable, mrs_drv_strength, mrs_odt, mrs_additive_latency are one-to-one mapping between soc.tcl and Parameters[3].

Clock rate setting in soc.tcl, i.e., cpu_clock_mhz, dram_clock_mhz, and lx_clock_mhz, are also converted to proper values for PLL control registers as sys_cpu_pll_ctl0, sys_cpu_pll_ctl1, sys_mem_pll_ctl0, sys_mem_pll_ctl1, sys_lx_pll_ctl0, and sys_lx_pll_ctl1[4].

"baudrate_divisor" is converted from uart_baudrate in soc.tcl. UART driver applies baudrate_divisor to UART controller during initialization.

"address" of mac_info comes from "address" of rtk_mac in soc.tcl. It is currently not used in Preloader.

The rest of parameters are used to maintain internal structures of Preloader: signature1, header_ver, and signature2, are used for sanity check, while spare_headers points to the address of spare models parameters of flash and DRAM on flash.

[3] Due to the variety of memory controller, the exact registers Composer translates to may differ in your environment.

[4] Due to the variety of PLL controller, the exact registers Composer translates to may differ in your environment.

```
....nor_spi_info_t has 40 bytes
....flash_layout_info_t has 52 bytes
....dram_gen2_info_t has 260 bytes
....pll_gen2_info_t has 28 bytes
....peri_info_t has 4 bytes
....mac_info_t has 8 bytes
....soc_t=404 bytes
nor_spi_info:
                  num_chips: 0x01
                  addr_mode: 0x00
              prefer_divisor: 0x10
               size_per_chip: 0x18
            prefer_rx_delay0: 0x00
            prefer_rx_delay1: 0x00
            prefer_rx_delay2: 0x00
            prefer_rx_delay3: 0x00
               prefer_rd_cmd: 0xeb
            prefer_rd_cmd_io: 0x00
           prefer_rd_dummy_c: 0x06
           prefer_rd_addr_io: 0x02
           prefer_rd_data_io: 0x02
                     wr_cmd: 0x02
                  wr_cmd_io: 0x00
                 wr_dummy_c: 0x00
                 wr_addr_io: 0x00
                 wr_data_io: 0x00
                wr_boundary: 0x08
                  erase_cmd: 0x20
                 erase_unit: 0x0c
                  pm_method: 0x01
                pm_rdsr_cmd: 0x05
               pm_rdsr2_cmd: 0x00
                pm_wrsr_cmd: 0x01
              pm_enable_cmd: 0x00
             pm_enable_bits: 0x0040
              pm_status_len: 0x01
                 rdbusy_cmd: 0x05
                 rdbusy_len: 0x02
                 rdbusy_loc: 0x00
       rdbusy_polling_period: 0x00
                         id: 0x00c22018
flash_layout_info:
            bootloader1_addr: 0x0000a000
            bootloader2_addr: 0x00000000
                kernel1_addr: 0x00100000
                kernel2_addr: 0x00000000
                rootfs1_addr: 0x00500000
                rootfs2_addr: 0x00000000
                    env_addr: 0x00040000
                    env_size: 0x00004000
```

```
                opt1_addr: 0x00000000
                opt2_addr: 0x00000000
                opt3_addr: 0x00000000
                opt4_addr: 0x00000000
                 end_addr: 0x00a00000
dram_gen2_info:
                      mcr: 0x00000000
                      dcr: 0x21320000
                DDR1_dtr0: 0x00001234
                DDR1_dtr1: 0x00005678
                DDR1_dtr2: 0x0000abcd
                DDR2_dtr0: 0x54422830
                DDR2_dtr1: 0x0404030f
                DDR2_dtr2: 0x0630d000
                DDR3_dtr0: 0x54433830
                DDR3_dtr1: 0x0404030f
                DDR3_dtr2: 0x0630d000
                    mpmr0: 0x0f3ff3ff
                    mpmr1: 0xff000000
                    dider: 0x00000000
                  d23oscr: 0x00000000
                    daccr: 0xc0000010
                  dacspcr: 0x0000007f
                  dacspar: 0x00000000
                  DDR1_mr: 0x00100061
                 DDR1_emr: 0x00110000
                  DDR2_mr: 0x00100862
                DDR2_emr1: 0x00110044
                DDR2_emr2: 0x00120000
                DDR2_emr3: 0x00130000
                 DDR3_mr0: 0x00101220
                 DDR3_mr1: 0x00110040
                 DDR3_mr2: 0x00120000
                 DDR3_mr3: 0x00130000
        static_cal_data_0: 0x0f110401
        static_cal_data_1: 0x0f130501
        static_cal_data_2: 0x0f100401
        static_cal_data_3: 0x0f130501
        static_cal_data_4: 0x0f130501
        static_cal_data_5: 0x0f100401
        static_cal_data_6: 0x0f030501
        static_cal_data_7: 0x0f100401
        static_cal_data_8: 0x00000001
        static_cal_data_9: 0x00000001
       static_cal_data_10: 0x00000001
       static_cal_data_11: 0x00000001
       static_cal_data_12: 0x00000001
       static_cal_data_13: 0x00000001
       static_cal_data_14: 0x00000001
       static_cal_data_15: 0x00000001
```

Figure 10: Example of "`make dumpcfg`".

```
        static_cal_data_16: 0x00110401
        static_cal_data_17: 0x00130501
        static_cal_data_18: 0x00000001
        static_cal_data_19: 0x00130501
        static_cal_data_20: 0x00030501
        static_cal_data_21: 0x00100401
        static_cal_data_22: 0x00130501
        static_cal_data_23: 0x00100401
        static_cal_data_24: 0x00000001
        static_cal_data_25: 0x00000001
        static_cal_data_26: 0x00000001
        static_cal_data_27: 0x00000001
        static_cal_data_28: 0x00000001
        static_cal_data_29: 0x00000001
        static_cal_data_30: 0x00000001
        static_cal_data_31: 0x00000001
        static_cal_data_32: 0x10100000
       size_auto_detection: 0x00
          calibration_type: 0x01
          tx_clk_phs_delay: 0x00
                clkm_delay: 0x00
              clkm90_delay: 0x00
          auto_calibration: 0x00
            zq_calibration: 0x00
              zq_impedance: 0x32
              drv_strength: 0x00
            mrs_dll_enable: 0x00
          mrs_drv_strength: 0x00
                   mrs_odt: 0x78
      mrs_additive_latency: 0x00
pll_gen2_info:
                    set_by: 0x00000001
          sys_cpu_pll_ctl0: 0x000041ac
          sys_cpu_pll_ctl1: 0x00000004
          sys_mem_pll_ctl0: 0x0000414c
          sys_mem_pll_ctl1: 0x00000004
           sys_lx_pll_ctl0: 0x0000414c
           sys_lx_pll_ctl1: 0x00000007
peri_info:
          baudrate_divisor: 0x006c
mac_info:
                   address: 11:22:33:44:55:66
check up message
                signature1: 0x03710601
                header_ver: 0x0000091d
                signature2: 0x62668696
             spare_headers: 0x9fc073b4
```

Figure 11: Example of "`make dumpcfg`" (Cont.).