S32G RDB2 Linux 板级开发包 Uboot 定制

by John Li (nxa08200)

本文说明S32G RDB2板Linux板级开发包BSP30 的Uboot细节，以帮助客户了解S32G的Uboot是如何运行的，以及如何修改到客户的新板上。

阅读本文之前请先阅读文档Automotive SW – S32G2 reference Software\Linux\《S32G_LinuxBSP30.0.0_User_Manual.pdf》，预先熟悉一下S32G的编译环境，本文部分内容与之重复。

《S32G_LinuxBSP30.0.0_Release_Notes.pdf》，为release notes。

本文推荐必读有第1，2章，第三章的第3.6节，为平台相关必须了解的信息。第三章其余部分为Linux背景知识介绍，可以选择阅读。

注意本文是使用默认的no-security uboot直接启动的方式为说明的，security ATF boot的方式另文说明，注意使用ATF后部分需要定制的部分在ATF中，uboot会简单很多。

请注意本文为培训和辅助文档，本文不是官方文档的替代，请一切以官方文档为准。

历史	说明	作者
V1	● 创建本文	John.Li
V2	● 更新到 BSP28	John.Li
V3	● 更新到 BSP29 ● 更新 DDR 定制	John.Li
V4	● 更新到 BSP30	John.Li

目录

1 S32G Linux 文档说明

从 NXP 官网帐号下，Automotive SW-S32G2 reference software 下载以下 Linux 相关文档：

分类	名称	说明	备注
Binary	binaries_auto_linux_bsp30.0_s32g2_pfe.tgz	Linux binary 支持 pfe 功能	建议使用最新版本，目前为 V30
发布须知	S32G_BSP30.0_Release_Notes.pdf	Release Notes	
用户手册	S32G_BSP30.0_User_Manual.pdf	Linux BSP 驱动详情说明	
Manifest 文件	S32G274_LinuxBSP30.0.0_PFE_license.manifest	含 PFE 功能的 manifest 文件	
Quick Start	S32G_BSP30.0_Quick_Start.pdf	Quick Start	
Quality Package	S32G_BSP30.0_Quality_Package.zip		

根据文档搭建 Yocto 编译环境和 standalone 编译环境。参考 Release Noes 的 What's New 一章了解最新的 BSP 相对于前一版本的更新。

2 创建 S32G RDB2 Linux 板级开发包编译环境

2.1 创建 yocto 编译环境:

S32G2 RDB2 Linux Yocto 编译环境要求使用 Ubuntu 18.04 编译主机, 可以从 ubuntu 官方网站下载 ISO 镜像 ubuntu-18.04.5-desktop-amd64.iso。如果要安装在虚拟机中，可以网上搜索安装方法，比如如下：

https://jingyan.baidu.com/article/86f4a73e493a0076d7526976.html

VMware安装Ubuntu-百度经验 (baidu.com)

启动后选择安装 WMware Tools for Linux 解决全屏问题。

安装结束后启动，需要事先执行以下命令安装编译所需包：

```
sudo apt-get update //之前要执行一下 update
sudo apt-get install python git curl
```

```
git config --global user.name xxx
git config --global user.email xxx@xxx.com
git config --list
```

根据文档 Automotive SW – S32G2 reference Software\Linux\
《S32G_BSP30.0_User_Manual.pdf》创建 yocto 编译环境时，需要注意以下几点：

1. 如果是在非中国大陆地区，或使用代理服务器的情况下，可以按照文档《S32G274_BSP30.0_User_Manual.pdf》所说，下载 Google 的 repo 工具：

```
mkdir ~/bin
curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+x ~/bin/repo
PATH=${PATH}:~/bin
```

中国大陆地区无法从 google 的服务器下载 repo 工具，可以改成如下清华的服务器：

```
mkdir bin
 cd bin
 curl https://mirrors.tuna.tsinghua.edu.cn/git/git-repo -o repo
 chmod a+x repo
 export PATH=~/bin:$PATH
```

2. 下载

```
mkdir fsl-auto-yocto-bsp
cd fsl-auto-yocto-bsp
repo init -u https://source.codeaurora.org/external/autobsps32/auto_yocto_bsp -b release/bsp30.0
repo sync
```

以下为使用清华服务器的情况：

```
mkdir imx-yocto-bsp
cd imx-yocto-bsp
repo init -u https://source.codeaurora.org/external/autobsps32/auto_yocto_bsp -b release/bsp30.0
--repo-url=https://mirrors.tuna.tsinghua.edu.cn/git/git-repo
repo sync
```

3. Yocto 编译，和烧写 SDcard 镜像：

首次运行时，先执行：

```
./sources/meta-alb/scripts/host-prepare.sh
```

有可能打印出如下提示：

```
Verifying sudo permission to execute apt-get command.
I ran the command: sudo -S -l which returned:
```

Matching Defaults entries for “usename” on “pcname”:

env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin, env_keep+="ftp_proxy http_proxy https_proxy"

User “usename”may run the following commands on “pcname”:

(ALL) ALL

This means you don't have sudo permission without password to execute

apt-get command as root. This is needed to install host packages correctly.

To configure this, as root using the command "/usr/sbin/visudo",

and add the following line in the User privilege section:

按照提示执行命令：

sudo /usr/sbin/visudo

然后在文档中增加一项：

“username” ALL = NOPASSWD: /usr/bin/apt-get //usename 为你们 ubuntu 主机用户名

保存。

再次运行，会自动下载安装没有安装的软件包。

创建编译目录：

source nxp-setup-alb.sh -m <machine>

<machine>目前支持 s32g274aevb, s32g274ardb, s32g274ardb2，我们这儿使用 s32g274ardb2 为例：

source nxp-setup-alb.sh -m s32g274ardb2

阅读并敲 Y 同意，以下列出目前支持的镜像：

Targets specific to NXP:

fsl-image-mfgtool-initramfs

fsl-image-auto

fsl-image-base

fsl-image-base-fit

fsl-image-bluebox

fsl-image-blueboxbootflash

fsl-image-blueboxdt

fsl-image-blueboxls2sdfactory

fsl-image-fit

fsl-image-flash

fsl-image-itbflash

fsl-image-nfs-initramfs

fsl-image-sim

fsl-image-ubuntu-base

fsl-image-ubuntu

fsl-image-ubuntu-ros

bitbake image 默认建议使用 fsl-image-auto：相关编译配置，以及建议使用功能，编译结果输出，烧写命令，SDK 编译与安装如下列表：

Image name	fsl-image-base	fsl-image-auto
Target	fsl-image-base 基础镜像	增加汽车相关应用
Application feature	测试使用	产品使用
Bitbake sample	bitbake fsl-image-base	bitbake fsl-image-auto
Output folder and image	/fsl-auto-yocto-bsp/build_s32g274ardb2 /tmp/deploy/images/s32g274ardb2 : ● 整个 sdcard 镜像，可以直接烧写 sdcard: fsl-image-base-s32g274ardb2.sdcard -> fsl-image-base-s32g274ardb2 -<编译时间>.rootfs.sdcard ● Rootfs: fsl-image-base-s32g274ardb2.tar.gz -> fsl-image-base-s32g274ardb2 -<编译时间>.rootfs.tar.gz ● 内核： Image -> Image--5.10.41-r0-s32g274ardb2-<编译时间>.bin ● Uboot: u-boot.s32-qspi -> u-boot-qspi-2020.04-r0.s32 u-boot.s32-sdcard -> u-boot-sdcard-2020.04-r0.s32	/fsl-auto-yocto-bsp/ build_s32g274ardb2 /tmp/deploy/images/s32g274ardb2 : ● 整个 sdcard 镜像，可以直接烧写 sdcard: fsl-image-auto-s32g274ardb2.sdcard -> fsl-image-auto-s32g274ardb2 -<编译时间>.rootfs.sdcard ● Rootfs: fsl-image-auto-s32g274ardb2.tar.gz -> fsl-image-auto-s32g274ardb2 -<编译时间>.rootfs. tar.gz ● 内核： Image -> Image--5.10.41-r0-s32g274ardb2-<编译时间>.bin ● Uboot: u-boot.s32-qspi -> u-boot-qspi-2020.04-r0.s32 u-boot.s32-sdcard -> u-boot-sdcard-2020.04-r0.s32
Burning SDcard image	sudo dd if= fsl-image-base-s32g274ardb2.sdcard of=/dev/sd<partition> bs=1M conv=fsync	sudo dd if= fsl-image-auto-s32g274ardb2.sdcard of=/dev/sd<partition> bs=1M conv=fsync
Compiling	bitbake fsl-image-base	bitbake fsl-image-auto

SDK	-c populate_sdk	-c populate_sdk
Compiling SDK **output**	/fsl-auto-yocto-bsp/ build_s32g274ardb2ubuntu tmp/deploy/sdk fsl-base-glibc-x86_64-aarch64-toolchain-30.0.sh	/fsl-auto-yocto-bsp/ build_s32g274ardb2ubuntu tmp/deploy/sdk fsl-auto-glibc-x86_64-aarch64-toolchain-30.0.sh
Install SDK	创建 SDK 安装目录 pwd ~/fsl-auto-yocto-bsp/ build_s32g274ardb2 mkdir sdk 到 SDK 输出目录下运行安装脚本 ./ fsl-base-glibc-x86_64-aarch64-toolchain-30.0.sh 输入安装 sdk 的目标目录 (default: /opt/ fsl-auto/30.0): ~/fsl-auto-yocto-bsp/ s32g274ardb2ubuntu/sdk You are about to install the SDK to "~/fsl-auto-yocto-bsp/ build_s32g274ardb2ubuntu/sdk ". Proceed[Y/n]? Y	创建 SDK 安装目录 pwd ~/fsl-auto-yocto-bsp/ build_s32g274ardb2 mkdir sdk 到 SDK 输出目录下运行安装脚本 ./ fsl-auto-glibc-x86_64-aarch64-toolchain-30.0.sh 输入安装 sdk 的目标目录 (default: /opt/fsl-auto/30.0): ~/fsl-auto-yocto-bsp/ s32g274ardb2ubuntu/sdk You are about to install the SDK to "~/fsl-auto-yocto-bsp/ build_s32g274ardb2ubuntu/sdk ". Proceed[Y/n]? Y
Install SDK **output**	~/fsl-auto-yocto-bsp/ build_s32g274ardb2ubuntu/sdk ● environment-setup-aarch64-fsl-linux ● sysroots	

注意：

1. 以上 yocto 编译测试环境为：
   - 编译主机与主机操作系统：

     操作系统：

     Type : Linux

     Operating system : Ubuntu 18.04.4 LTS

     …

     内存:8G

     :~$ cat /proc/meminfo

     MemTotal: 8167360 kB

CPU:8 核

```
cat /proc/cpuinfo
…
processor       : 7
vendor_id       : GenuineIntel
cpu family      : 6
model           : 45
model name      : Intel(R) Xeon(R) Gold 6136 CPU @ 3.00GHz
```

硬盘：2T SSD

```
df -h
..
/dev/mapper/vg_data-lv_opt  2.0T  1.7T  181G  91% /opt
…
```

建议编译主机使用 SSD 硬盘，实际中发现使用 SSD 硬盘的主机编译 yocto 要快很多。

文档要求如下：

All the steps described below have been run and validated on Ubuntu-18.04 LTS (native or through a virtual machine).It is then recommended to install Ubuntu-18.04 LTS before going through the following sections.

A Yocto build needs at least 50GB of free space and takes a lot of time (2 to 10 hours, depending on the system configuration). It is recommended to use a powerful system with many cores and a fast storage media (SSD is recommended).The recommended RAM size is 8 GB.

- 测试网络环境：

  NXP 公司网络。

2. 编译问题解决：
   - 编译主机版本不对或安装包不全，如上所说，更新到 ubuntu1804，安装全部要求的安装包，执行前先 update 一下。
   - Yocto 的编译依赖社区服务器器的支持，对于非 NXP 提供的相关组件包的下载和编译失败的问题，请寻找社区支持，一般遇到 这种情况，可以多试几次或者使用代理服务器。
3. 如果只是需要 toolchain, 可以用命令 bitbake meta-toolchain 编译出来。
4. 总共时长：

   编译时长要根据编译主机配置，网络情况和 git 服务器的情况而定，一般是从几个小时到一天都有可能，建议在尽量使用代理服务器。

2.2 独立编译

有很多客户希望可以独立的编译 Linux Uboot/kernel 和应用程序，而不希望使用 Yocto 编译环境，如下说明，注意，第一次使用 Yocto 编译有可能也是必要的，因为：

- 可以获得编译出来的 rootfs 和原始 sdcard 镜像，当然，从网上下载默认 demo 镜像也可以。
- 可以编译 SDK 来获得编译器，当然，也可以如下单独下载。

1. 编译器说明：

   根据文档《S32G_LinuxBSP30.0.0_User_Manual.pdf》说明，可以如下单独下载 GCC10.2.0 的编译器：

https://developer:arm:com/-/media/Files/downloads/gnu-a/10:2-2020:11/binrel/gcc-arm-10:2-2020:11-x86_64-aarch64-none-linux-gnu:tar:xz?revision=972019b5-912f-4ae6-864a-f61f570e2e7e&la=en&hash=B8618949E6095C87E4C9FFA1648CAA67D4997D88

下载以下版本：

解压后如文档说所配置交叉编译器位置

CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-linux-gnu-

由于文档《S32G_LinuxBSP30.0.0_User_Manual.pdf》已经说明了如何使用，本文只说明如何使用 Yocto 编译出来的 SDK 来编译的方法：

2. 编译 uboot：

   打开一个终端

```
pwd
~/s32g/standalone
git clone  https://source.codeaurora.org/external/autobsps32/u-boot
cd u-boot
git tag |grep bsp30
 ....
 bsp30.0-2020.04
 ...
git checkout bsp30.0-2020.04
git status

HEAD detached at bsp30.0-2020.04
nothing to commit, working directory clean

ls configs |grep s32g
 …
```

s32g274abluebox3_defconfig

s32g274a_emu_defconfig

s32g274ardb2_defconfig

s32g274ardb2_qspi_defconfig

s32g274ardb_defconfig

s32g274ardb_qspi_defconfig

s32g274a_SECURE_BOOT.config

s32g274a_sim_defconfig

s32g2xxaevb_defconfig

s32g2xxaevb_qspi_defconfig

…

source ~ /fsl-auto-yocto-bsp/environment-setup-aarch64-poky-linux (sdk 安装地址)

make s32g274ardb2_defconfig

make

编译 log 如下：

…

LD   u-boot

  OBJCOPY u-boot-nodtb.bin

  OBJCOPY u-boot.srec

  SYM   u-boot.sym

start=$(aarch64-fsl-linux-nm u-boot | grep __rel_dyn_start | cut -f 1 -d ' '); end=$(aarch64-fsl-linux-nm u-boot | grep __rel_dyn_end | cut -f 1 -d ' '); tools/relocate-rela u-boot-nodtb.bin 0x340a0000 $start $end

  DTC   arch/arm/dts/fsl-s32g274ardb2.dtb

  FDTGREP dts/dt-spl.dtb

  SHIPPED dts/dt.dtb

  CAT   u-boot-dtb.bin

  COPY   u-boot.dtb

  COPY   u-boot.bin

  LD   u-boot.elf

0+1 records in

0+1 records out

18485 bytes (18 kB, 18 KiB) copied, 0.000262407 s, 70.4 MB/s

10+1 records in

10+1 records out

719397 bytes (719 kB, 703 KiB) copied, 0.000909532 s, 791 MB/s

```
 MKIMAGE u-boot.s32

IVT:                    Offset: 0x0           Size: 0x100
IVT (duplicate):        Offset: 0x1000        Size: 0x100
DCD:                    Offset: 0x1200        Size: 0x2000
AppBootCode Header:     Offset: 0x3200        Size: 0x40
U-Boot/FIP:             Offset: 0x3400        Size: 0xbfbe5
U-Boot Environment:     Offset: 0x1e0000      Size: 0x2000

 CFGCHK  u-boot.cfg
```

编译结束后的输出镜像为：

```
u-boot.s32
u-boot.bin
```

3. 编译 Linux 内核：

```
 pwd
~/s32g/standalone
git clone  https://source.codeaurora.org/external/autobsps32/linux
cd linux
git tag|grep bsp30
bsp30.0-5.10.41-rt
bsp30.0-5.4-rt
git checkout bsp30.0-5.10.41-rt
git status
 HEAD detached at BSP30.0-5.10.41-rt
source ~ /fsl-auto/sdk/environment-setup-aarch64-poky-linux
ls arch/arm64/configs/s32g*
arch/arm64/configs/s32gen1_defconfig
arch/arm64/configs/s32gen1_emu_defconfig
arch/arm64/configs/s32gen1_emu_mmc_delta_defconfig
```

For s32g274aevb, s32g254aevb, s32g233aevb, s32g274ardb and s32g274ardb2, the defconfig is s32gen1_defconfig.

```
make  s32gen1_defconfig
make
```

```
make dtbs clean
make dtbs //just make dtb
```

编译结束后的输出镜像为：

```
DTC     arch/arm64/boot/dts/freescale/fsl-s32g274a-rdb2.dtb
OBJCOPY arch/arm64/boot/Image
```

4. 更新镜像到已经烧写了*.sdcard 的 sdcard 里

bootloader:

```
vmuser@ubuntu:~$ cat /proc/partitions
major minor  #blocks  name
...
   8       32    7761920 sdc
   8       33      32768 sdc1
   8       34    6918144 sdc2
sudo dd if=u-boot.s32 of=/dev/sdc bs=256 count=1 seek=0
sync
sudo dd if=u-boot.s32 of=/dev/sdc bs=512 seek=1 skip=1
sync
```

kernel and btb:

```
cp Image to Boot
cp s32g274ardb2.dtb to Boot
```

注意，文档《S32G_LinuxBSP30.0.0_User_Manual.pdf》有说明如何将一张空的 sdcard 烧录上 uboot/kernel/rootfs 的方法， 本文不再重复。

3 FSL Uboot 定制

目前 5.10.x 内核对应的 uboot 2020_04 有两个重要功能 是：

- 如同内核一样，支持 FDT(Flatted device tree)
- 与内核类似，支持 DM(driver model)

以下详细说明这两个功能：

3.1 FDT 支持

3.1.1 说明

FDT，flatted device tree，扁平设备树。简单理解为将部分设备信息结构存放到 device tree 文件中。 uboot 最终将其 device tree 编译成 dtb 文件，使用过程中通过解析该 dtb 来获取板级设备信息。 uboot 的 dtb 和 kernel 中的 dtb 是一致的。

DTB 头结构体如下：

```
Scripts\dtc\libfdt\fdt.h
struct fdt_header {
        fdt32_t magic;                    /* magic word FDT_MAGIC */
        fdt32_t totalsize;          /* total size of DT block */
        fdt32_t off_dt_struct;            /* offset to structure */
        fdt32_t off_dt_strings;           /* offset to strings */
        fdt32_t off_mem_rsvmap;                   /* offset to memory reserve map */
        fdt32_t version;          /* format version */
        fdt32_t last_comp_version;        /* last compatible version */
        /* version 2 fields below */
        fdt32_t boot_cpuid_phys;          /* Which physical CPU id we're
                                             booting on */
        /* version 3 fields below */
        fdt32_t size_dt_strings;    /* size of the strings block */
        /* version 17 fields below */
        fdt32_t size_dt_struct;           /* size of the structure block */
};
```

其中，magic 是一个固定的值，0xd00dfeed（大端）

```
#define FDT_MAGIC 0xd00dfeed        /* 4: version, 4: total size */
```

只要提取待验证 dtb 的地址上的数据的前四个字节，与 0xd00dfeed（大端）或者 0xedfe0dd0（小端）进行比较，如果匹配的话，就说明对应待验证 dtb 就是一个合法的 dtb。

以下编译宏：

```
\arch\Kconfig
config ARM
…
        select SUPPORT_OF_CONTROL
```

\include\generated\autoconfig.h

#define CONFIG_OF_CONTROL 1

CONFIG_OF_CONTROL=y 用于表示是否使用了 dtb 的方式

3.1.2 dtb 在 uboot 中的位置

dtb可以以两种形式编译到uboot的镜像中。

- dtb 和 uboot 的 bin 文件分离
    - 如何使能
      需要打开 CONFIG_OF_SEPARATE 宏来使能。

\dts\ Kconfig

```
choice
    prompt "Provider of DTB for DT control"
    depends on OF_CONTROL
config OF_SEPARATE
    …
config OF_EMBED
```

默认选择： CONFIG_OF_SEPARATE=y // 是否将 dtb 和 uboot 分离表

- 编译说明

  在这种方式下，uboot 的编译和 dtb 的编译是分开的，先生成 uboot 的 bin 文件，然后再另外生成 dtb 文件。

- 最终位置

  dtb 最终会追加到 uboot 的 bin 文件的最后面。也就是 uboot.img 的最后一部分。 因此，可以通过 uboot 的结束地址符号，也就是_end 符号来获取 dtb 的地址。

- dtb 集成到 uboot 的 bin 文件内部
    - 如何使能

      需要打开 CONFIG_OF_EMBED 宏来使能。

    - 编译说明

      在这种方式下，在编译 uboot 的过程中，也会编译 dtb。

    - 最终位置

注意：最终 dtb 是包含到了 uboot 的 bin 文件内部的。dtb 会位于 uboot 的.dtb.init.rodata 段中，并且在代码中可以通过__dtb_dt_begin 符号获取其符号。 因为这种方式不够灵活，文档上也不推荐，所以后续也不具体研究，简单了解一下即可。

- 另外，也可以通过 fdtcontroladdr 环境变量来指定 dtb 的地址 可以通过直接把 dtb 加载到内存的某个位置，并在环境变量中设置 fdtcontroladdr 为这个地址，达到动态指定 dtb 的目的。在调试中使用。

  S32G RDB2 板的 DTS 源文件前文已经说明。

3.1.3 uboot 中如何获取 dtb

在uboot初始化过程中，需要对dtb做两个操作：

- 获取 dtb 的地址，并且验证 dtb 的合法性
- 因为我们使用的 dtb 并没有集成到 uboot 的 bin 文件中，也就是使用的 CONFIG_OF_SEPARATE 方式。因此，在 relocate uboot 的过程中并不会去 relocate dtb。因此，这里我们还需要自行为 dtb 预留内存空间并进行 reloca
- relocate 之后，还需要重新获取一次 dtb 的地址

  这部分过程是在 init_board_f 中实现, 对应代码 common/board_f.c

```
static const init_fnc_t init_sequence_f
        |->fdtdec_setup, /* 获取 dtb 的地址，并且验证 dtb 的合法性*/
int fdtdec_setup(void)
{
…
        /* FDT is at end of image */
        gd->fdt_blob = (ulong *)&_end; /* 当使用 CONFIG_OF_SEPARATE 的方式时，也就是 dtb 追加到 uboot 的 bin 文件后面时，通过_end 符号来获取 dtb 地址 */
…
        /* Allow the early environment to override the fdt address */
    /* 可以通过环境变量 fdtcontroladdr 来指定 gd->fdt_blob，也就是指定 fdt 的地址 */
        gd->fdt_blob = (void *)getenv_ulong("fdtcontroladdr", 16,
                                            (uintptr_t)gd->fdt_blob);
…
/* 最终都把 dtb 的地址存储在 gd->fdt_blob 中 */
/* 在 fdtdec_prepare_fdt 中检查 fdt 的合法性  */
```

```
// 判断 dtb 是否存在，以及是否有四个字节对齐
// 然后再调用 fdt_check_header 看看头部是否正常。fdt_check_header 主要是检查 dtb 的 magic 是否正确
        return fdtdec_prepare_fdt();
}
      |->reserve_fdt, /* 为 dtb 分配新的内存地址空间 */
static int reserve_fdt(void)
{
#ifndef CONFIG_OF_EMBED
        /*
         * If the device tree is sitting immediately above our image then we
         * must relocate it. If it is embedded in the data section, then it
         * will be relocated with other data.
         */
         /*
          当使用 CONFIG_OF_EMBED 方式时，也就是 dtb 集成在 uboot 中的时候，relocate uboot 过程中也会把 dtb 一起 relocate，所以这里就不需要处理。
   当使用 CONFIG_OF_SEPARATE 方式时，就需要在这里地方进行 relocate
    */
        if (gd->fdt_blob) {
                /* 获取 dtb 的 size */
                gd->fdt_size = ALIGN(fdt_totalsize(gd->fdt_blob) + 0x1000, 32);
        /* 为 dtb 分配新的内存空间 */
                gd->start_addr_sp -= gd->fdt_size;
                gd->new_fdt = map_sysmem(gd->start_addr_sp, gd->fdt_size);
                debug("Reserving %lu Bytes for FDT at: %08lx\n",
                      gd->fdt_size, gd->start_addr_sp);
        }
#endif

        return 0;
}
      |->reloc_fdt, /* relocate dtb  */
static int reloc_fdt(void)
{
```

```
#ifndef CONFIG_OF_EMBED
/*
当使用 CONFIG_OF_EMBED 方式时，也就是 dtb 集成在 uboot 中的时候，relocate uboot 过程中也会把 dtb 一起 relocate，所以这里就不需要处理。
当使用 CONFIG_OF_SEPARATE 方式时，就需要在这里地方进行 relocate

*/
 /* 检查 GD_FLG_SKIP_RELOC 标识 */
        if (gd->flags & GD_FLG_SKIP_RELOC)
                return 0;
        if (gd->new_fdt) {
                /* relocate dtb 空间 */
                memcpy(gd->new_fdt, gd->fdt_blob, gd->fdt_size);
                /* 切换 gd->fdt_blob 到 dtb 的新的地址空间上 */
                gd->fdt_blob = gd->new_fdt;
        }
#endif

        return 0;
}
```

3.1.4 uboot 中 dtb 解析的常用接口

gd->fdt_blob 已经设置成了 dtb 的地址了。 注意，fdt 提供的接口都是以 gd->fdt_blob（dtb 的地址）为参数的。以下只简单说明几个接口的功能，另外，用节点在 dtb 中的偏移地址来表示一个节点。也就是节点变量 node 中，存放的是节点的偏移地址。

- lib/fdtdec.c 中
    - **fdt_path_offset**
      int fdt_path_offset(const void *fdt, const char *path)
      eg：node = fdt_path_offset(gd->fdt_blob, “/aliases”);
      功能：获得 dtb 下某个节点的路径 path 的偏移。这个偏移就代表了这个节点。
    - **fdt_getprop**
      const void *fdt_getprop(const void *fdt, int nodeoffset, const char *name, int *lenp)
      eg： mac = fdt_getprop(gd->fdt_blob, node, “mac-address”, &len);
      功能：获得节点 node 的某个字符串属性值。

- **fdtdec_get_int_array、fdtdec_get_byte_array**
  int fdtdec_get_int_array(const void *blob, int node, const char *prop_name, u32 *array, int count)
  eg： ret = fdtdec_get_int_array(blob, node, “interrupts”, cell, ARRAY_SIZE(cell));
  功能：获得节点 node 的某个整形数组属性值。

- **fdtdec_get_addr**
  fdt_addr_t fdtdec_get_addr(const void *blob, int node, const char *prop_name)
  eg：fdtdec_get_addr(blob, node, “reg”);
  功能：获得节点 node 的地址属性值。

- **fdtdec_get_config_int、fdtdec_get_config_bool、fdtdec_get_config_string**
  功能：获得 config 节点下的整形属性、bool 属性、字符串等等。

- **fdtdec_get_chosen_node**
  int fdtdec_get_chosen_node(const void *blob, const char *name)
  功能：获得 chosen 下的 name 节点的偏移

- **fdtdec_get_chosen_prop**
  const char *fdtdec_get_chosen_prop(const void *blob, const char *name)
  功能：获得 chosen 下 name 属性的值

- lib/fdtdec_common.c 中

  - **fdtdec_get_int**
    int fdtdec_get_int(const void *blob, int node, const char *prop_name, int default_val)
    eg： bus->udelay = fdtdec_get_int(blob, node, “i2c-gpio,delay-us”, DEFAULT_UDELAY);
    功能：获得节点 node 的某个整形属性值。

  - **fdtdec_get_uint**
    功能：获得节点 node 的某个无符号整形属性值。

3.2 DM(driver model)支持

3.2.1 说明

uboot 引入了驱动模型（driver model），这种驱动模型为驱动的定义和访问接口提供了统一的方法。提高了驱动之间的兼容性以及访问的标准型。 uboot 驱动模型和 kernel 中的设备驱动模型类似，但是又有所区别。

具体细节建议参考./doc/driver-model/README.txt

在 configs/s32g274ardb2_defconfig 中定义了如下：

```
CONFIG_DM=y
```

DM 和 uclass 是息息相关的，如果我们希望在某个模块引入 DM，那么就需要使用相应模块的 uclass driver 来代替旧版的通用 driver:

CONFIG_DM_I2C/MMC/ SPI_FLASH/ETH/PCI/PMIC/ PMIC_VR5510/ PMIC_PF5020/ PMIC_FS560/SPI

REGULATOR/ REGULATOR_FIXED/ REGULATOR_GPIO/ THERMAL=y

看 driver/serial/Makefile

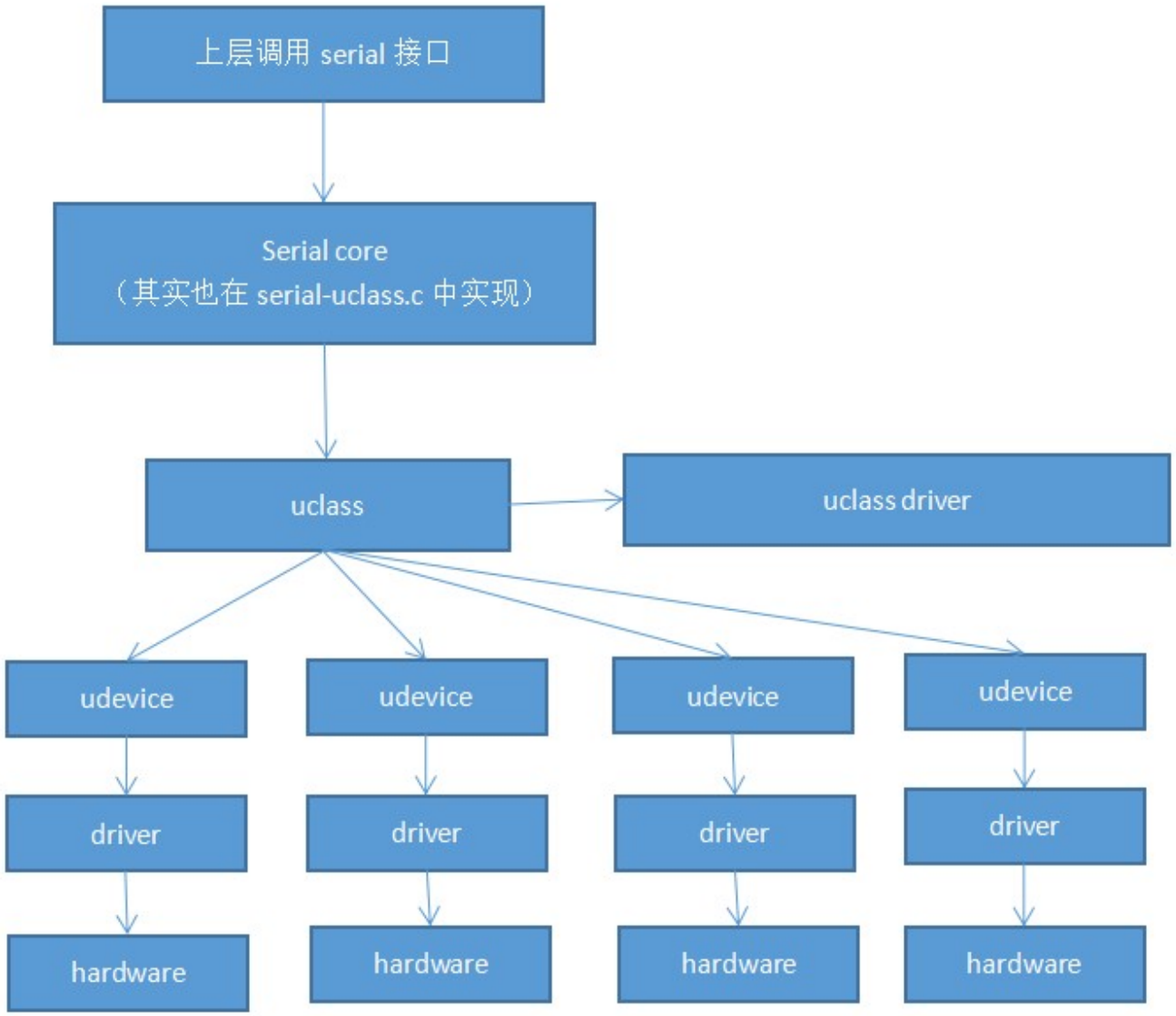
```
ifdef CONFIG_DM_SERIAL
        obj-y += serial-uclass.o ## 引入 dm 的 serial core 驱动
else
        obj-y += serial.o ## 通用的 serial core 驱动
endif
```

3.2.2 uboot DM 整体架构

DM 的四个组成部分

- **udevice**：简单就是指设备对象，可以理解为 kernel 中的 device
- **driver**：udevice 的驱动，可以理解为 kernel 中的 device_driver。和底层硬件设备通信，并且为设备提供面向上层的接口
- **uclass**：uclass，使用相同方式的操作集的 device 的组。相当于是一种抽象。uclass 为那些使用相同接口的设备提供了统一的接口。
  例如，GPIO uclass 提供了 get/set 接口。再例如，一个 I2C uclass 下可能有 10 个 I2C 端口，4 个使用一个驱动，另外 6 个使用另外一个驱动
- **uclass_driver**：对应 uclass 的驱动程序。主要提供 uclass 操作时，如绑定 udevice 时的一些操作

调用关系框架图



相互之间的关系

结合上图来看：

- 上层接口都是和 uclass 的接口直接通讯。
- uclass 可以理解为一些具有相同属性的 udevice 对外操作的接口，uclass 的驱动是 uclass_driver，主要为上层提供接口。
- udevice 的是指具体设备的抽象，对应驱动是 driver，driver 主要负责和硬件通信，为 uclass 提供实际的操作集。

- udevice 找到对应的 uclass 的方式主要是通过：udevice 对应的 driver 的 id 和 uclass 对应的 uclass_driver 的 id 是否匹配。
- udevice 会和 uclass 绑定。driver 会和 udevice 绑定。uclass_driver 会和 uclass 绑定。

uclass和udevice都是动态生成的。在解析fdt中的设备的时候，会动态生成udevice。 然后找到udevice对应的driver，通过driver中的uclass id得到uclass_driver id。从uclass链表中查找对应的uclass是否已经生成，没有生成的话则动态生成uclass

3.2.3 uboot DM 的初始化

Uboot DM 初始化的主要工作包括：

- DM 的初始化
  - 创建根设备 root 的 udevice，存放在 gd->dm_root 中。
    根设备其实是一个虚拟设备，主要是为 uboot 的其他设备提供一个挂载点。
  - 初始化 uclass 链表 gd->uclass_root
- DM 中 udevice 和 uclass 的解析
  - udevice 的创建和 uclass 的创建
  - udevice 和 uclass 的绑定
  - uclass_driver 和 uclass 的绑定
  - driver 和 udevice 的绑定
  - 部分 driver 函数的调用

DM 初始化的接口在 dm_init_and_scan 中。 可以发现在 uboot relocate 之前的 initf_dm 和之后的 initr_dm 都调用了这个函数,主要区别在于参数。 首先说明一下 dts 节点中的“u-boot,dm-pre-reloc”属性，当设置了这个属性时，则表示这个设备在 relocate 之前就需要使用。当 dm_init_and_scan 的参数为 true 时，只会对带有“u-boot,dm-pre-reloc”属性的节点进行解析。而当参数为 false 的时候，则会对所有节点都进行解析。 由于“u-boot,dm-pre-reloc”的情况比较少，

所以这里只学习参数为 false 的情况。也就是 initr_dm 里面的 dm_init_and_scan(false)

|->dm_init(); // DM 的初始化

```
int dm_init(void)
{
        if (gd->dm_root) {
                // 根设备已经存在，说明 DM 已经初始化过了
```

```
        }
        // 初始化 uclass 链表
        INIT_LIST_HEAD(&DM_UCLASS_ROOT_NON_CONST);

        ret = device_bind_by_name(NULL, false, &root_info, &DM_ROOT_NON_CONST);
// DM_ROOT_NON_CONST 是指根设备 udevice，root_info 是表示根设备的设备信息
// device_bind_by_name 会查找和设备信息匹配的 driver，然后创建对应的 udevice 和 uclass 并进行绑定，最后放在 DM_ROOT_NON_CONST 中。
// device_bind_by_name 后续我们会进行说明，这里我们暂时只需要了解 root 根设备的 udevice 以及对应的 uclass 都已经创建完成。

        ret = device_probe(DM_ROOT_NON_CONST);
// 对根设备执行 probe 操作，
    // device_probe 后续再进行说明
/*
这里就完成的 DM 的初始化了
（1）创建根设备 root 的 udevice，存放在 gd->dm_root 中。
（2）初始化 uclass 链表 gd->uclass_root

*/
```

|->dm_scan_platdata(pre_reloc_only); // 从平台设备中解析 udevice 和 uclass,

|->dm_scan_fdt(gd->fdt_blob, pre_reloc_only); // 从 dtb 中解析 udevice 和 uclass

```
int dm_scan_fdt(const void *blob, bool pre_reloc_only)
{
// 此时传进来的参数
// parent=gd->dm_root，表示以 root 设备作为父设备开始解析
// blob=gd->fdt_blob，指定了对应的 dtb
// offset=0，从偏移 0 的节点开始扫描
// pre_reloc_only=0，不只是解析 relotion 之前的设备

        return dm_scan_fdt_node(gd->dm_root, blob, 0, pre_reloc_only);
}
```

| |->dm_scan_fdt_node

```
int dm_scan_fdt_node(struct udevice *parent, const void *blob, int offset,
                     bool pre_reloc_only)
{
        int ret = 0, err;
/*  以下步骤相当于是遍历每一个 dts 节点并且调用 lists_bind_fdt 对其进行解析 */
        for (offset = fdt_first_subnode(blob, offset);
        // 获得 blob 设备树的 offset 偏移下的节点的第一个子节点
             offset > 0;

             offset = fdt_next_subnode(blob, offset)) {
                        // 循环查找下一个子节点
                if (pre_reloc_only &&
                    !fdt_getprop(blob, offset, "u-boot,dm-pre-reloc", NULL))
                        continue;
                if (!fdtdec_get_is_enabled(blob, offset)) {
                        // 判断节点状态是否是 disable，如果是的话直接忽略
                        dm_dbg("   - ignoring disabled device\n");
                        continue;
                }
                // 解析绑定这个节点，dm_scan_fdt 的核心，下面具体分析
                err = lists_bind_fdt(parent, blob, offset, NULL);

                }
        }
}
```

lists_bind_fdt 是从 dtb 中解析 udevice 和 uclass 的核心。 其具体实现如下：

// parent 指定了父设备，通过 blob 和 offset 可以获得对应的设备的 dts 节点，对应 udevice 结构通过 devp 返回

```
int lists_bind_fdt(struct udevice *parent, const void *blob, int offset,
                   struct udevice **devp)
struct driver *driver = ll_entry_start(struct driver, driver); // 获取 driver table 地址
const int n_ents = ll_entry_count(struct driver, driver); // 获取 driver table 长度
```

```
name = fdt_get_name(blob, offset, NULL);
    dm_dbg("bind node %s\n", name);
    // 打印当前解析的节点的名称
for (entry = driver; entry != driver + n_ents; entry++) {
                        // 遍历 driver table 中的所有 driver
                        ret = driver_check_compatible(entry->of_match, &id,
                                                      compat);
                        if (!ret)
                                break;
                }
    // 找到对应的 driver，调用 device_bind 进行绑定，会在这个函数中创建对应 udevice 和 uclass 并切进行绑定，后面继续说明
                ret = device_bind_with_driver_data(parent, entry, name,
                                                   id->data, offset, &dev);
    // 将 udevice 设置到 devp 指向的地方中，进行返回
                        if (devp)
                                *devp = dev;
```

在 device_bind_common 中实现了 udevice 和 uclass 的创建和绑定以及一些初始化操作，这里专门学习一下 device_bind_common。 device_bind_common 的实现如下(去除部分代码)
driver/core/device.c

```
static int device_bind_common(struct udevice *parent, const struct driver *drv,
                              const char *name, void *platdata,
                              ulong driver_data, int of_offset,
                              uint of_platdata_size, struct udevice **devp)


// parent:父设备
// drv：设备对应的 driver
// name：设备名称
// platdata：设备的平台数据指针
// of_offset：在 dtb 中的偏移，即代表了其 dts 节点
// devp：所创建的 udevice 的指针，用于返回
{
    // 获取 driver id 对应的 uclass，如果 uclass 原先并不存在，那么会在这里创建 uclass 并其 uclass_driver 进行绑定
```

```
    ret = uclass_get(drv->id, &uc);
// 分配一个 udevice
    dev = calloc(1, sizeof(struct udevice));
dev->platdata = platdata; // 设置 udevice 的平台数据指针
    dev->driver_data = driver_data;
    dev->name = name; // 设置 udevice 的 name
    dev->of_offset = of_offset; // 设置 udevice 的 dts 节点偏移
    dev->parent = parent; // 设置 udevice 的父设备
    dev->driver = drv; // 设置 udevice 的对应的 driver，相当于 driver 和 udevice 的绑定
    dev->uclass = uc; // 设置 udevice 的所属 uclass
dev->platdata = calloc(1,
                                        drv->platdata_auto_alloc_size);
                        // 为 udevice 分配平台数据的空间，由 driver 中的 platdata_auto_alloc_size 决定
// 添加到父设备的子设备链表中
    /* put dev into parent's successor list */
    if (parent)
            list_add_tail(&dev->sibling_node, &parent->child_head);
 // uclass 和 udevice 进行绑定，主要是实现了将 udevice 链接到 uclass 的设备链表中
    ret = uclass_bind_device(dev);
    if (ret)
            goto fail_uclass_bind;

    /* if we fail to bind we remove device from successors and free it */
    if (drv->bind) {
            // 执行 udevice 对应 driver 的 bind 函数
            ret = drv->bind(dev);
            if (ret)
                    goto fail_bind;
    }
    if (parent && parent->driver->child_post_bind) {
            // 执行父设备的 driver 的 child_post_bind 函数
            ret = parent->driver->child_post_bind(dev);
            if (ret)
```

```
                goto fail_child_post_bind;
        }
        if (uc->uc_drv->post_bind) {
                // 执行所属 uclass 的 post_bind 函数
                ret = uc->uc_drv->post_bind(dev);
                if (ret)
                        goto fail_uclass_post_bind;
        }

        if (parent)
                dm_dbg("Bound device %s to %s\n", dev->name, parent->name);
        // 将 udevice 进行返回
        if (devp)
                *devp = dev;
    // 设置已经绑定的标志
        // 后续可以通过 dev->flags & DM_FLAG_ACTIVATED 或者 device_active 宏来判断设备是否已经被激活
    //上述就完成了 dtb 的解析，udevice 和 uclass 的创建，以及各个组成部分的绑定关系
    //注意，这里只是绑定，即调用了 driver 的 bind 函数，但是设备还没有真正激活，也就是还没有执行设备的 probe 函数。
        dev->flags |= DM_FLAG_BOUND;
```

上述就完成了 dtb 的解析，udevice 和 uclass 的创建，以及各个组成部分的绑定关系。 注意，这里只是绑定，即调用了 driver 的 bind 函数，但是设备还没有真正激活，也就是还没有执行设备的 probe 函数。

3.2.4 DM 工作流程

经过前面的 DM 初始化以及设备解析之后，我们只是建立了 udevice 和 uclass 之间的绑定关系。但是此时 udevice 还没有被 probe，其对应设备还没有被激活。 激活一个设备主要是通过 device_probe 函数，所以在介绍 DM 的工作流程前，先说明 device_probe 函数。

主要工作归纳如下：

- 分配设备的私有数据
- 对父设备进行 probe

- 执行 probe device 之前 uclass 需要调用的一些函数
- 调用 driver 的 ofdata_to_platdata，将 dts 信息转化为设备的平台数据
- 调用 driver 的 probe 函数
- 执行 probe device 之后 uclass 需要调用的一些函数

driver/core/device.c

```
int device_probe(struct udevice *dev)
{
drv = dev->driver; // 获取这个设备对应的 driver
// 为设备分配私有数据
dev->priv = alloc_priv(drv->priv_auto_alloc_size, drv->flags);
// 为设备所属 uclass 分配私有数据
size = dev->uclass->uc_drv->per_device_auto_alloc_size;
dev->flags |= DM_FLAG_ACTIVATED; // 设置 udevice 的激活标志
ret = uclass_pre_probe_device(dev); // uclass 在 probe device 之前的一些函数的调用
// 调用 driver 中的 ofdata_to_platdata 将 dts 信息转化为设备的平台数据
                ret = drv->ofdata_to_platdata(dev);
// 调用 driver 的 probe 函数，到这里设备才真正激活了
                ret = drv->probe(dev);
}
```

通过 uclass 来获取一个 udevice 并且进行 probe 有如下接口：

```
driver/core/uclass.c
int uclass_get_device(enum uclass_id id, int index, struct udevice **devp)  //通过索引从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_get_device_by_name(enum uclass_id id, const char *name,
        struct udevice **devp) //通过设备名从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp) //通过序号从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_get_device_by_of_offset(enum uclass_id id, int node,
        struct udevice **devp) //通过 dts 节点的偏移从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_get_device_by_phandle(enum uclass_id id, struct udevice *parent,
        const char *name, struct udevice **devp) //通过设备的“phandle”属性从 uclass 的设备链表中获取 udevice，并且进行 probe
int uclass_first_device(enum uclass_id id, struct udevice **devp) //从 uclass 的设备链表中获取第一个 udevice，并且进行 probe
int uclass_next_device(struct udevice **devp) //从 uclass 的设备链表中获取下一个 udevice，并且进行 probe
```

这些接口主要是获取设备的方法上有所区别，但是 probe 设备的方法都是一样的，都是通过调用 uclass_get_device_tail->device_probe 来 probe 设备的。以 uclass_get_device 为例:

```
/*  通过索引从 uclass 中获取 udevice*/
int uclass_get_device(enum uclass_id id, int index, struct udevice **devp)
{
        ret = uclass_find_device(id, index, &dev); //通过索引从 uclass 的设备链表中获取对应的 udevice
        return uclass_get_device_tail(dev, ret, devp); // 调用 uclass_get_device_tail 进行设备的 get，最终会调用 device_probe 来对设备进行 probe
}
int uclass_get_device_tail(struct udevice *dev, int ret,
                                                struct udevice **devp)
{
        ret = device_probe(dev); // 调用 device_probe 对设备进行 probe，这个函数在前面说明过了
        *devp = dev;
}
```

3.2.5 DM 工作流程实例 DM-I2C

Drivers\i2c\i2c-uclass.c 定义了一个 uclass_driver

```
UCLASS_DRIVER(i2c) = {
        .id                     = UCLASS_I2C,
        .name                   = "i2c",
        .flags                  = DM_UC_FLAG_SEQ_ALIAS,
        .post_bind              = i2c_post_bind,
        .init                   = i2c_uclass_init,
        .priv_auto_alloc_size = sizeof(struct i2c_priv),
        .pre_probe      = i2c_pre_probe,
        .post_probe             = i2c_post_probe,
        .per_device_auto_alloc_size = sizeof(struct dm_i2c_bus),
        .per_child_platdata_auto_alloc_size = sizeof(struct dm_i2c_chip),
        .child_post_bind = i2c_child_post_bind,
};
```

I2C DTS support: \uboot\arch\arm\dts\fsl-s32-gen1.dtsi

```
        i2c4: i2c@402DC000 {
                compatible = "fsl,vf610-i2c";
…
        };
/i2c/2/3/
```

S32G I2C 设备驱动：

/drivers/i2c/mxc_i2c.c

```
static const struct udevice_id mxc_i2c_ids[] = {
        { .compatible = "fsl,vf610-i2c", .data = I2C_QUIRK_FLAG, },
};
    U_BOOT_DRIVER(i2c_mxc) = {
        .name = "i2c_mxc",
        .id = UCLASS_I2C,
        .of_match = mxc_i2c_ids,
        .probe = mxc_i2c_probe,
        .priv_auto_alloc_size = sizeof(struct mxc_i2c_bus),
        .ops = &mxc_i2c_ops,
        .flags = DM_FLAG_PRE_RELOC,
    };
```

在 DM 初始化的过程中 uboot 自己创建对应的 udevice 和 uclass，并自己实现将 udevice 绑定到对应的 uclass 中。

udevice 的 probe 则通过 i2c core API 调用实现：

i2c core API 接口说明如下：其中既提供了兼容老版本的接口，也提供了 DM 框架下的接口。

- DM 框架下的接口:

```
int dm_i2c_read(struct udevice *dev, uint offset, uint8_t *buffer, int len);
int dm_i2c_write(struct udevice *dev, uint offset, const uint8_t *buffer,
```

```
                        int len)
int dm_i2c_probe(struct udevice *bus, uint chip_addr, uint chip_flags,
                        struct udevice **devp);
int dm_i2c_reg_read(struct udevice *dev, uint offset);
int dm_i2c_reg_write(struct udevice *dev, uint offset, unsigned int val);
int dm_i2c_xfer(struct udevice *dev, struct i2c_msg *msg, int nmsgs);
int dm_i2c_set_bus_speed(struct udevice *bus, unsigned int speed);
int dm_i2c_get_bus_speed(struct udevice *bus);
```

实例说明如下：

Drivers/power/pmic/vr5510.c

dm_i2c_read

dm_i2c_write

3.3 Uboot 目录 结构

一般定制需要使用的目录有源代码如下：

\uboot

|-> arch

| |->arm

| | |->cpu

| | | |->armv8

| | | | |->s32:cpu.c, fdt.c

| | | | | |->s32-gen1: ncore.c, soc.c, s32g274a.c

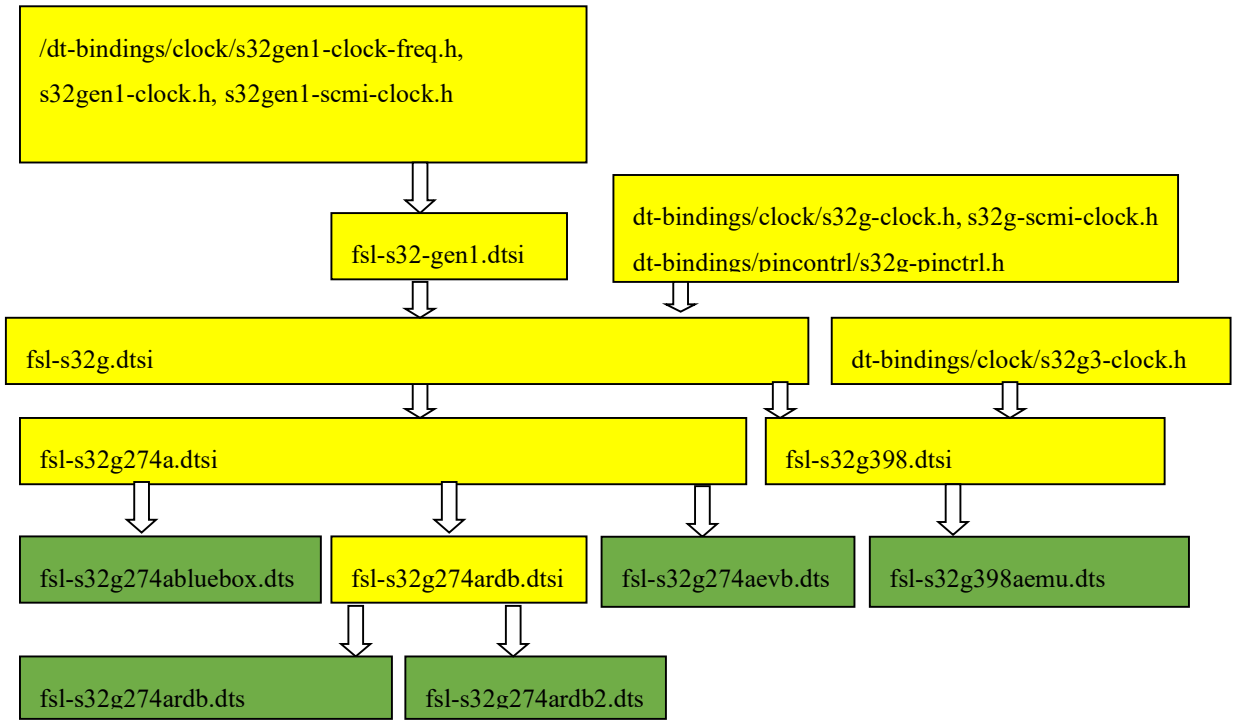
| | | |->u-boot.lds

| | |->dts

| | | |-> fsl-s32g.dtsi, fsl-s32g274a.dtsi, fsl-s32g274abluebox3.dts, fsl-s32g274aemu.dts, fsl-s32g274aevb.dts, fsl-s32g274ardb.dts, fsl-s32g274ardb.dtsi, fsl-s32g274ardb2.dts, fsl-s32g274a-sec-boot.dts, fsl-s32g398a.dtsi, fsl-s32g398aemu.dts, fsl-s32-gen1.dtsi

| | | | |->include: dt-bindings

Dts 的相关关系如下图：



| | |->include

| | | |->asm

| | | | |->arch-s32

| | | | | |->s32-gen1

|->board //board 以下的代码与板级配置相关，定制需要修改的部分基本在这些文件中

| |->freescale

| | |->s32-gen1：ddr_init.c, ddr_utils.c, ddrss_cfg.c, eth.c, s32g274a.c, sja1105_cfg_rdb.c

| | | |->s32g274a: ddrc_cfg.c, dmem_cfg.c, phy_cfg.c, pie_cfg.c,dp_swap_cfg.c

|->configs

| |-> s32g2xxaevb_defconfig, s32g2xxaevb_qspi_defconfig, s32g274a_emu_defconfig, s32g274a_SECURE_BOOT.config, s32g274a_sim_defconfig, s32g274abluebox3_defconfig, s32g274ardb_defconfig, s32g274ardb_qspi_defconfig, s32g274ardb2_defconfig, s32g274ardb2_qspi_defconfig, s32g398a_emu_defconfig

|->common

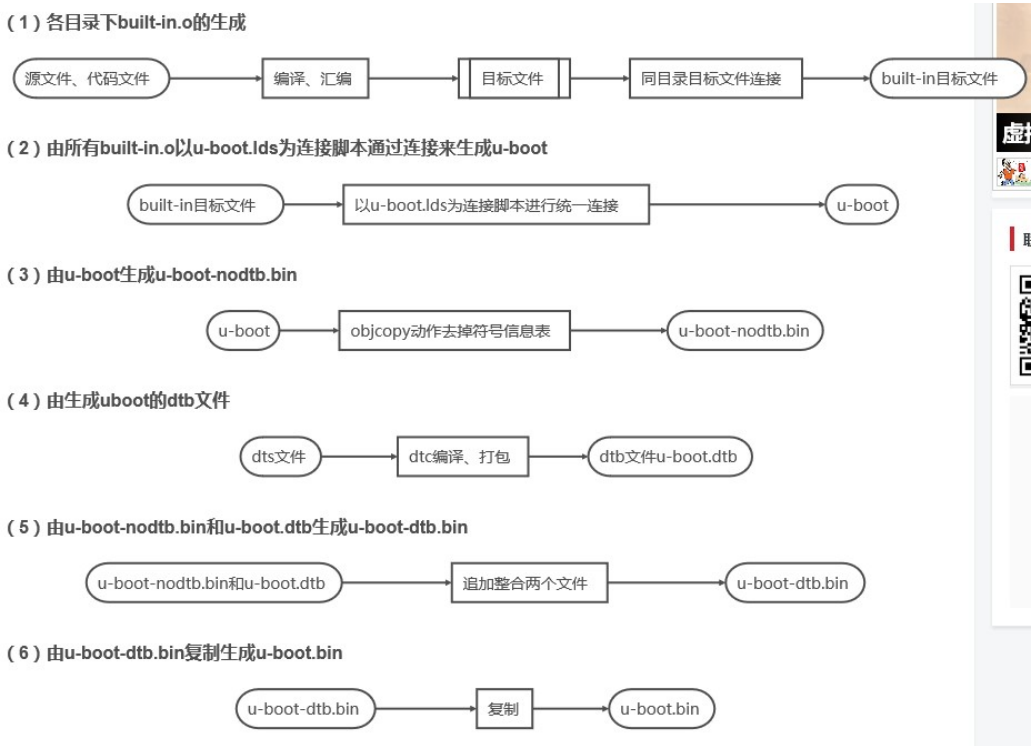
| |->board_f.c, board_r.c

|->drivers

| |->i2c
| | |->i2c-uclass.c,
| |->mmc: mmc-uclass.c
| |->net
| | |->phy:
| |->power
| | |->pmic:
|->include//以下配置文档控制了 uboot 的配置相关宏，特别重要
| |->s32g274a_common.h (board/freescale/s32-gen1)//s32g274ardb.c include
| | |->include config.h
| | | |->#include <configs/s32g274a.h>
| | | | |->#include <configs/s32-gen1.h>
| | | | | |->#include <configs/s32.h>

3.4 Uboot 编译

3.4.1 编译流程



S32G Uboot

3.4.2 生成文件：

文件	说明
u-boot	初步链接后得到的 uboot 文件
u-boot-nodtb.bin	在 u-boot 的基础上，经过 objcopy 去除符号表信息之后的可执行程序
u-boot.dtb	dtb 文件
u-boot-dtb.bin	将 u-boot-nodtb.bin 和 u-boot.dtb 打包在一起的文件
u-boot.bin	在需要 dtb 的情况下，直接由 u-boot-dtb.bin 复制而来，也就是编译 u-boot 的最终目标
u-boot.lds	uboot 的连接脚本
u-boot.map	连接之后的符号表文件
u-boot.cfg	由 uboot 配置生成的文件

3.5 Uboot 初始化流程

板级初始化的流程

_main

|->board_init_f_alloc_reserve :堆栈、GD、early malloc空间的分配
|->board_init_f_init_reserve:堆栈、GD、early malloc空间的初始化
|->board_init_f:uboot relocate前的板级初始化以及relocate的区域规划
|->relocate_code、relocate_vectors:进行uboot和异常中断向量表的重定向 ,旧堆栈的清空
|->board_init_r:uboot relocate后的板级初始化
|->run_main_loop:进入命令行状态，等待终端输入命令以及对命令进行处理

如下详细代码说明:

(\arch\arm\cpu\lib\vector.S): _start

_start:

…

|-> b reset

arch\arm\cpu\armv8\start.S

reset:

….

/* Processor specific initialization */

| |-> bl lowlevel_init

| |->bl _main

Arch\arm\lib\crt0_64.S

ENTRY(_main)

板级初始化的流程

| | |->board_init_f_alloc_reserve

| | |->board_init_f_init_reserve

| | |->board_init_f

Common\board_f.c

if (initcall_run_list(init_sequence_f))

setup_mon_len

fdtdec_setup, /* 获取 dtb 的地址，并且验证 dtb 的合法性，之前已经说明*/

trace_early_init

initf_malloc

log_init

initf_bootstage

arch_cpu_init,(arch/arm/cpu/armv8/s32/cpu.c) /* basic arch cpu dependent setup */

| | | |-> enable_early_clocks()

/* Platforms with Concerto/Ncore have to explicitly initialize

* the interconnect before any cache operations are performed.

* Also, ensure that clocks are initialized before the interconnect.

*

* Note: TF-A has already initialized these, so don't do it again if

* we're running at EL2.

*/

ret = enable_early_clocks(); //initial clock of a53/xbar/lin/ddr

| | | | |-> enable_ddr_clock //此处为 DDR clock 的设置，改频与展频可能需要修改，详情请参考相关文档。

ddr_pll_freq = S32GEN1_DDR_PLL_VCO_FREQ;

ddr_freq = S32GEN1_DDR_FREQ;

//Ver B 设置 DDR clock

#define S32GEN1_DDR_PLL_VCO_FREQ (1333333333UL)

#define S32GEN1_DDR_FREQ (666666666)

#define S32G274A_REV2_DDR_PLL_VCO_FREQ (1600 * MHZ)

#define S32G274A_REV2_DDR_FREQ (800 * MHZ)

```
#ifdef CONFIG_NXP_S32G2XX
        if (!is_s32gen1_soc_rev1()) {
                ddr_pll_freq = S32G274A_REV2_DDR_PLL_VCO_FREQ;
                ddr_freq = S32G274A_REV2_DDR_FREQ;
        }
#endif
//打开 DDR

            s32gen1_enable(&ddr);

            s32_gentimer_init();

            mach_cpu_init
        /* include Open IPC channel , power on SMMU*/
        initf_dm /*此处仅初始化 relocal 之前的驱动*/
        arch_cpu_init_dm,
        board_early_init_f/*此外用于初始化外设 iomux，非常重要
| | |  |->setup_iomux_uart();//此处为调试串口的 iomux，如果修改了调试串口，需要修改
        get_clocks, /*get sdhc clock to gd*/
        timer_init,                     /* initialize timer */
        board_postclk_init,
        env_init,                       /* initialize environment */
        init_baud_rate,                 /* initialze baudrate settings */
        serial_init,                    /* serial communications setup *//*此处用于真实的初始化调试串口，在此之后，
串口打印才有输出，之前的代码就需要用调试器调试。
* stage 1 init of console */
        display_options,  /* say that we are here */
        display_text_info,              /* show debugging info if required */
        checkcpu,
        print_resetinfo,
        //打印 cpu 信息
        int print_cpuinfo(void)
        {
        #ifdef CONFIG_S32G274A
```

```
        printf("CPU:\tNXP S32G274A");
        #ifdef CONFIG_TARGET_TYPE_S32GEN1_SIMULATOR
        printf("\n");
        #else
        printf(" rev. %d.%d.%d\n",
                    get_siul2_midr1_major() + 1,
                    get_siul2_midr1_minor(),
                    get_siul2_midr2_subminor());
        #endif  /* CONFIG_TARGET_TYPE_S32GEN1_SIMULATOR */
       #elif defined(CONFIG_S32R45)
        printf("CPU:\tNXP S32R45\n");
       #elif defined(CONFIG_S32V344)
        printf("CPU:\tNXP S32V344\n");
       #endif
        printf("Reset cause: %s\n", get_reset_cause());

        return 0;
    show_board_info
       misc_init_f,
    init_func_i2c//初始化 i2c 口
    init_func_vid
    announce_dram_init
      dram_init /* configure available RAM banks */
| | |  |->ddr_init（board/freescale/s32g-gen1/ddr_init.c）
| | | |  |->/* Init DDR controller based on selected parameter values */
                ret = ddrc_init_cfg(&configs[i]);此外为具体的 DDRC 寄存器配置，修改新的 DDR 主要是修改此
处
| | | |  |-> ret = set_axi_parity();          /* Setup AXI ports parity */
| | | |  |->execute_training(&configs[i]); /* Init PHY module */
| | | |  |->post_train_setup(); /* Execute post training setup */
    post_init_f,
    testdram,
    init_post
      setup_dest_addr,
```

```
        reserve_round_4k /* Round memory pointer down to next 4 kB limit */
        reserve_video,
        reserve_trace,
        reserve_uboot,
        reserve_malloc,
        reserve_board,
        setup_machine,
        reserve_global_data,
        reserve_fdt,
        reserve_bootstage,
        reserve_bloblist,
        reserve_arch,
        reserve_stacks,
        dram_init_banksize,
        show_dram_config,
display_new_sp,
        reloc_fdt, /* relocate dtb  */
        reloc_bootstage,
        reloc_bloblist,
        setup_reloc,
         clear_bss,
| | |->relocate_code
| | |->board_init_r
Common\board_r.c
if (initcall_run_list(init_sequence_r))
                    hang();
static init_fnc_t init_sequence_r[] = {
        initr_trace,
        initr_reloc,
        initr_caches,   /*caches initial*/
      initr_reloc_global_data,
        initr_barrier,
        initr_malloc,
        log_init,
```

```
        initr_bootstage,  /* Needs malloc() but has its own timer */
        initr_console_record,
        initr_noncached,
        initr_dm, /*之前已经分析*/
…
        board_init,         /* Setup chipselects */ //由于 board_init_f 已经初始化了外设的 iomux，所以此处没有代码
set_cpu_clk_info, /* Setup clock information */
initr_binman,
        initr_dm_devices,
        stdio_init_tables,
        initr_serial, //此外通过 DM 初始化串口
        initr_announce,
initr_watchdog,
initr_addr_map,
int board_early_init_r(void)
{
        release_resets();
        return 0;
}
initr_post_backlog,
initr_pci,
arch_early_init_r,
power_init_board,
INIT_FUNC_WATCHDOG_RESET
initr_nand,
initr_mmc,//此外初始化 emmc
initr_env,
initr_malloc_bootparams,
initr_secondary_cpu,
        /*
         * Do pci configuration
         */
        initr_pci,
        stdio_add_devices,
```

```
        initr_jumptable,
console_init_r,                 /* fully init console as a device */
        console_announce_r,
        show_board_info,
arch_misc_init,                 /* miscellaneous arch-dependent init */
misc_init_r,            /* miscellaneous platform-dependent init */
        interrupt_init,
initr_enable_interrupts,
timer_init,             /* initialize timer */
initr_ethaddr,//以太网 mac 地址传入
board_late_init,
        initr_scsi,
initr_bbmii,
initr_net,//此外初始化 网口
| | | | ->eth_initialize();
| | | | ->reset_phy();
initr_post,
initr_ide,
last_stage_init,
initr_bedbug,
initr_mem,
blkcache_init,
run_main_loop,//此外进入 uboot 主循环
```

3.6 Uboot 定制

根据以上的 Uboot 启动过程分析，涉及到红字部分，是有可能需要定制的。

3.6.1 修改 DDR 配置

在 Uboot 中修改 DDR 需要修改 DDR 配置和大小两个部分：

根据以下代码分析：

```
\common\board_f.c\init_sequence_f
|->dram_init(arch\arm\cpu\armv8\s32\s32-gen1\soc.c)
```

| |-> ddr_init(board\freescale\s32-gen1\ddr_init.c);

| | |->init_image_sizes

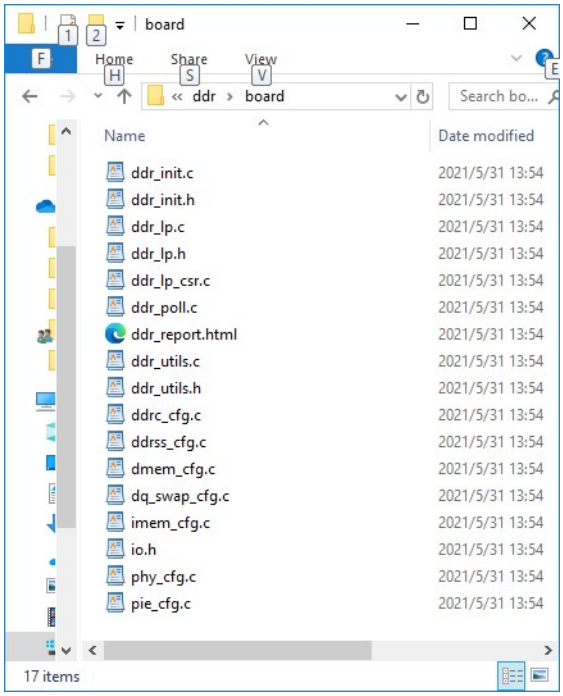
/* Init DDR controller based on selected parameter values */

| | |->ret = ddrc_init_cfg(&configs[i]);//此处实际设置 DDRC 寄存器，初始化 DDR 控制器

| | | |-> ret = load_register_cfg(config->ddrc_cfg_size, config->ddrc_cfg);

…

S32 DS 的 DDR 配置工具运行成功后会输出以下文件：



需要替换的配置文件包括：

u-boot/board/freescale/s32-gen1/ddr_init.c/h; ddr_utils.c/h; ddrss_cfg.c，imem_cfg.c

u-boot/board/freescale/s32-gen1/s32g274a/ ddrc_cfg.c; dmem_cfg.c; dq_swap_cfg.c; phy_cfg.c; pie_cfg.c

共 11 个文件,注意：

- 原始代码中：S32G B version 芯片是使用的\u-boot\board\freescale\s32-gen1\s32g274a\rev2 中的配置文件，使用宏 CONFIG_NXP_S32G2XX 来区分，如果直接替换掉文件，则不需要区分 VerA 和 VerB 的芯片，原文件已经直接更新了。
- 原始代码中，使用宏 CONFIG_S32GEN1_DRAM_INLINE_ECC 来区分是否在配置中打开 ECC，如果直接替换掉文件，则不需要区分，原文件已经直接更新了。
- 新的 ddrc_init.c 中的函数调用：

  uint32_t load_register_cfg_16(size_t size, struct regconf_16 cfg[])

```
{
    size_t i;
    for (i = 0; i < size; i++)
        write_16(cfg[i].data, (uintptr_t)cfg[i].addr);
    return NO_ERR;
}
```

在 BSP 源文件中没有定义，而是定义在导出的 io.h 文件中，将相关代码 porting 到 \arch\arm\include\asm\io.h 中：

```
/* 16 bits*/
#define __arch_put_16(v, a)  (*(volatile uint16_t *)((size_t)(a)) = (v))
#define __arch_get_16(a)     (*(volatile uint16_t *)((size_t)(a)))
/* 16 bits*/
#define write_16(v, c) \
    __extension__({ uint16_t __v = (v); dmb(); __arch_put_16(__v, (c)); __v; })
#define read_16(c) \
    __extension__({ uint16_t __v = __arch_get_16(c); dmb(); __v; })
```

除此之外，memory 的大小还需要 uboot 传递给内核，其中 DTS 中控制 gd->db 传入内核大小，如下：

```
\common\board_f.c\init_sequence_f
|->dram_init_banksize, (arch\arm\cpu\armv8\s32\cpu.c)
#ifdef CONFIG_S32_GEN1 //定义在(Board\freescale\s3-gen1\Kconfig config S32_GEN1= default y)
|    |->ret = fdtdec_setup_memory_banksize();
#else
|    |    |->fdtdec_setup_memory_banksize_fdt
\arch\arm\dts\fsl-s32g274ardb.dtsi:
/ {
    memory@80000000 {
            device_type = "memory";
            reg = <0 0x80000000 0 0x80000000>; //0x8000_0000 开头 2GB
    };
    memory@880000000 {
            device_type = "memory";
            reg = <0x8 0x80000000 0 0x80000000>;//0x8_8000_0000 开头 2GB
    };
```

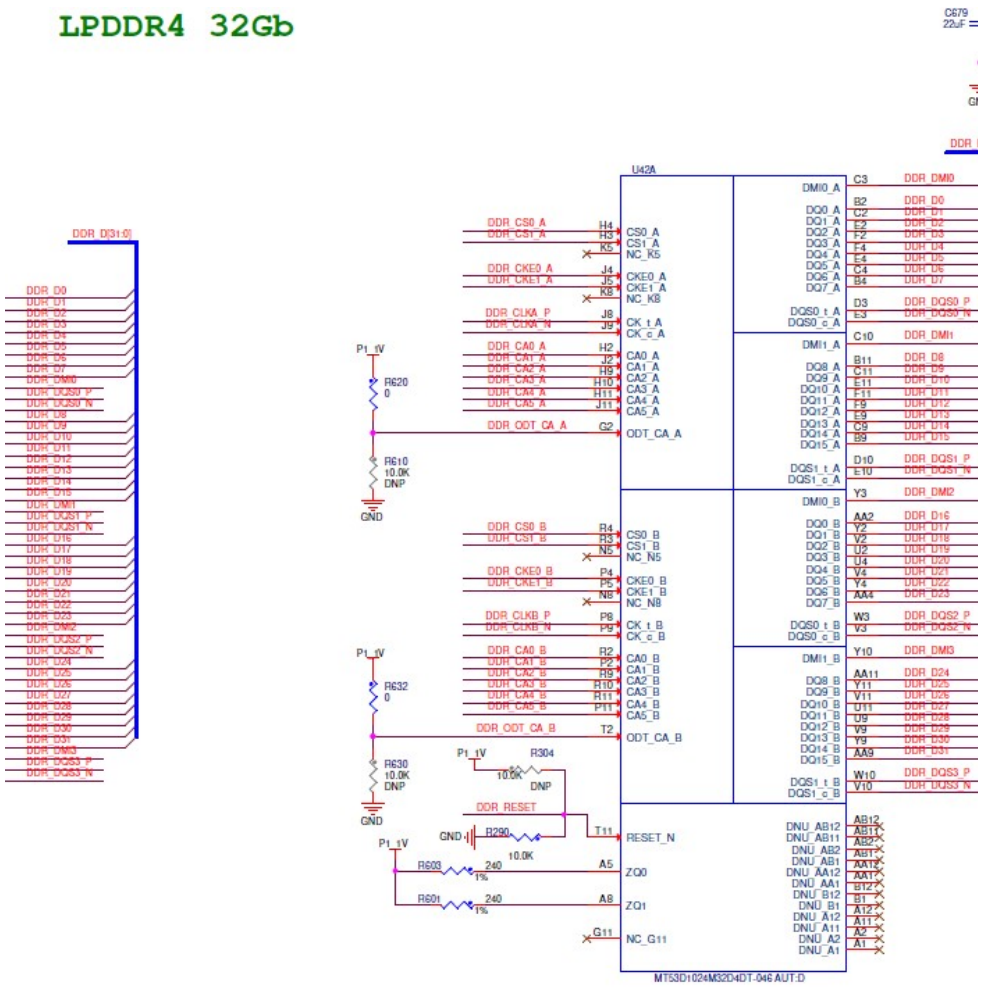
```
        sram@34000000 {
                device_type = "memory";
                reg = <0 0x34000000 0 0x800000>;
        };
};
```

对比芯片手册 附件：S32G_Memory_Map.xlsx

0x08_0000_0000	0x08_3FFF_FFFF	1	DDR_0 (0 - 1G bytes) Always mirrored
0x08_4000_0000	0x08_7FFF_FFFF	1	DDR_0 (1G - 2G bytes) Mirror in Mode 2+0
0x08_8000_0000	0x08_FFFF_FFFF	2	DDR_0 (2G - 4G bytes)

0x00_3400_0000	0x00_353F_FFFF	20480	Internal SRAM (first 8Mbyte)

S32G RDB2 连接外部 4GB LPDDR4:



而后再根据是否打开的 ECC 来减去 1/8 的内部内存大小：

```
int dram_init_banksize(void)
{…
ret = fdtdec_setup_memory_banksize();//从 dts 中读出内存大小
…
#ifdef CONFIG_SYS_FSL_DDRSS
        s32_exclude_ecc_from_dram();
#endif
…
}
```

另外：

```
phys_size_t __weak get_effective_memsize(void)
{…
```

```
        size = CONFIG_SYS_FSL_DRAM_SIZE1;
#ifdef CONFIG_SYS_FSL_DDRSS
        s32_exclude_ecc_range(CONFIG_SYS_FSL_DRAM_BASE1, &size);
#endif
…
}
```

而在文件 include\configs\s32-gen.h 中的宏：(实际测试中发现不修改没有影响)

```
#define CONFIG_SYS_FSL_DRAM_BASE1       0x80000000
#define CONFIG_SYS_FSL_DRAM_SIZE1       0x80000000
#define CONFIG_SYS_FSL_DRAM_BASE2       0x880000000
#define CONFIG_SYS_FSL_DRAM_SIZE2       0x80000000
```

则控制了 MMU 的 Mapping 大小和位置：

arch\arm\cpu\armv8\s32\cpu.c

```
    static struct mm_region final_map[] = {..
          {
           CONFIG_SYS_FSL_DRAM_BASE1, CONFIG_SYS_FSL_DRAM_BASE1,
           CONFIG_SYS_FSL_DRAM_SIZE1,
           PTE_BLOCK_MEMTYPE(MT_NORMAL) | PTE_BLOCK_OUTER_SHARE
          },…
          {
           CONFIG_SYS_FSL_DRAM_BASE2, CONFIG_SYS_FSL_DRAM_BASE2,
           CONFIG_SYS_FSL_DRAM_SIZE2,
           PTE_BLOCK_MEMTYPE(MT_NORMAL) | PTE_BLOCK_OUTER_SHARE
          },….}
```

所以修改 DDR 的大小要注意：

- 只需要修改 DTS 中内存大小，头文件中内存大小可以不修改，实际测试无影响，也可以修改。
- 如果打开了 ECC，则在头文件中内存大小还要在真实的大小上减少 1/8。

\u-boot\drivers\ddr\fsl\Kconfig

```
config SYS_FSL_DDRSS
          bool "DDR Subsystem support"
          help
            Select DDR Subsystem support.
          default y if (S32_GEN1 && !TARGET_TYPE_S32GEN1_SIMULATOR)
```

所以默认是打开的，如果要关闭，可以 vi .config 去掉，或是从

\u-boot\configs\s32g274ardb2_defconfig

CONFIG_ SYS_FSL_DDRSS is not set

以下为 512M，1G 和 2G 大小的内存大小 修改方法对比:注意修改 CONFIG_SYS_FSL_DRAM_BASE2 的目的是保证实际 memory mapping 的物理地址连续。

		512M(1CS)	1G(1CS)	2G(1CS)

fsl-s32g274a.dtsi		memory@80000000 {
		device_type = "memory";
		reg = <0 0x80000000 0 0x20000000>; //0x8000_0000 开头 2GB
		};
		/*注掉 memory mapping 2
		memory@880000000 {
		device_type = "memory";
		reg = <0x8 0x80000000 0 0x80000000>;//0x8_8000_0000 开头 2GB
		};
		*/
			memory@80000000 {
			device_type = "memory";
			reg = <0 0x80000000 0 0x40000000>; //0x8000_0000 开头 1GB
			};
			/*注掉 memory mapping 2
			memory@880000000 {
			device_type = "memory";
			reg = <0x8 0x80000000 0 0x80000000>;//0x8_8000_0000 开头 2GB
			};
			*/
				memory@80000000 {
				device_type = "memory";
				reg = <0 0x80000000 0 0x80000000>; //0x8000_0000 开头 2GB
				};
				/*注掉 memory mapping 2
				memory@880000000 {
				device_type = "memory";
				reg = <0x8 0x80000000 0 0x80000000>;//0x8_8000_0000 开头 2GB
				};
				*/

s32-gen.h 此处可以不修改	ECC enable	#define CONFIG_SYS_FSL_DRAM_BASE1 0x80000000 #define CONFIG_SYS_FSL_DRAM_SIZE1 0x16000000 #define CONFIG_SYS_FSL_DRAM_BASE2 0x820000000 #define CONFIG_SYS_FSL_DRAM_SIZE2 0x00000000	#define CONFIG_SYS_FSL_DRAM_BASE1 0x80000000 #define CONFIG_SYS_FSL_DRAM_SIZE1 0x32000000 #define CONFIG_SYS_FSL_DRAM_BASE2 0x840000000 #define CONFIG_SYS_FSL_DRAM_SIZE2 0x00000000	#define CONFIG_SYS_FSL_DRAM_BASE1 0x80000000 #define CONFIG_SYS_FSL_DRAM_SIZE1 0x70000000 #define CONFIG_SYS_FSL_DRAM_BASE2 0x880000000 #define CONFIG_SYS_FSL_DRAM_SIZE2 0x00000000
	ECC Disable	#define CONFIG_SYS_FSL_DRAM_BASE1 0x80000000	#define CONFIG_SYS_FSL_DRAM_BASE1 0x80000000	#define CONFIG_SYS_FSL_DRAM_BASE1 0x80000000

		#define CONFIG_SYS_FSL_DRAM_SIZE1 0x20000000 #define CONFIG_SYS_FSL_DRAM_BASE2 0x820000000 #define CONFIG_SYS_FSL_DRAM_SIZE2 0x00000000	#define CONFIG_SYS_FSL_DRAM_SIZE1 0x40000000 #define CONFIG_SYS_FSL_DRAM_BASE2 0x840000000 #define CONFIG_SYS_FSL_DRAM_SIZE2 0x00000000	#define CONFIG_SYS_FSL_DRAM_SIZE1 0x80000000 #define CONFIG_SYS_FSL_DRAM_BASE2 0x880000000 #define CONFIG_SYS_FSL_DRAM_SIZE2 0x00000000

总结需要修改或更新的文件包括：

git status

…

modified: arch/arm/dts/fsl-s32g274a.dtsi

modified: arch/arm/include/asm/io.h

modified: board/freescale/s32-gen1/ddr_init.c

modified: board/freescale/s32-gen1/ddr_init.h

modified: board/freescale/s32-gen1/ddr_utils.c

modified: board/freescale/s32-gen1/ddr_utils.h

modified: board/freescale/s32-gen1/ddrss_cfg.c

modified: board/freescale/s32-gen1/imem_cfg.c

modified: board/freescale/s32-gen1/s32g274a/ddrc_cfg.c

modified: board/freescale/s32-gen1/s32g274a/dmem_cfg.c

modified: board/freescale/s32-gen1/s32g274a/dq_swap_cfg.c

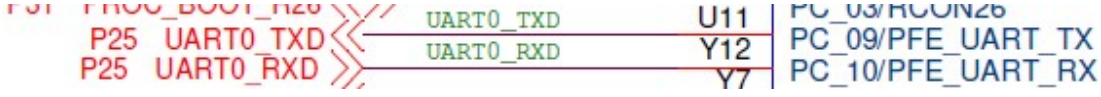
modified: board/freescale/s32-gen1/s32g274a/phy_cfg.c

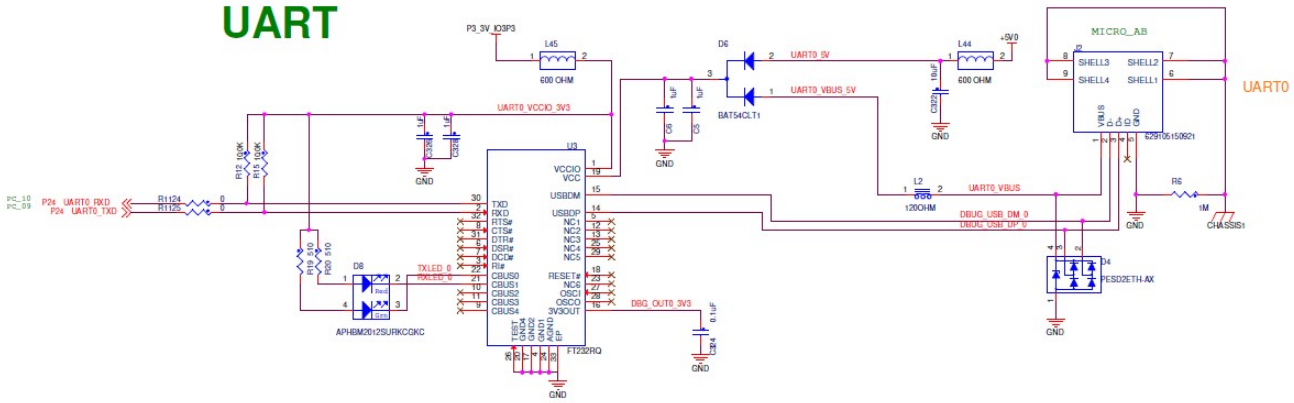
modified: board/freescale/s32-gen1/s32g274a/pie_cfg.c

modified: include/configs/s32-gen1.h //此处可以不修改

3.6.2 修改调试串口与 IOMUX 说明

一般情况下，不建议修改 A 核 的调试串口（Uboot/Kernel 均使用同一下调试串口）：如下 S32G RDB2 板设计的调试串口：





参考芯片手册中的 S32G2_IOMUX.xls:

Port	Address	Function	Module	Description	BGA	Chiptopport
PC_09	0x4009C2E4	2=LIN0_TX	2=LINFlex_0	LIN TX Data	U11	Pad_041
PC_10	0x4009CA40	4=LIN0_TX	4=LINFlex_0	LIN RX Data	U11	Pad_042

及芯片手册 Boot 一章：

30.11.4 UART

BootROM supports boot from the LINFlexD_0 module, configured in UART mode. The communication between the MCU and the host (transmitting utility) is done by exchange of data through UART packets. LINFlexD_0 is configured to communicate in 8E1 configuration.

30.11.4.1 UART IOMUX configuration

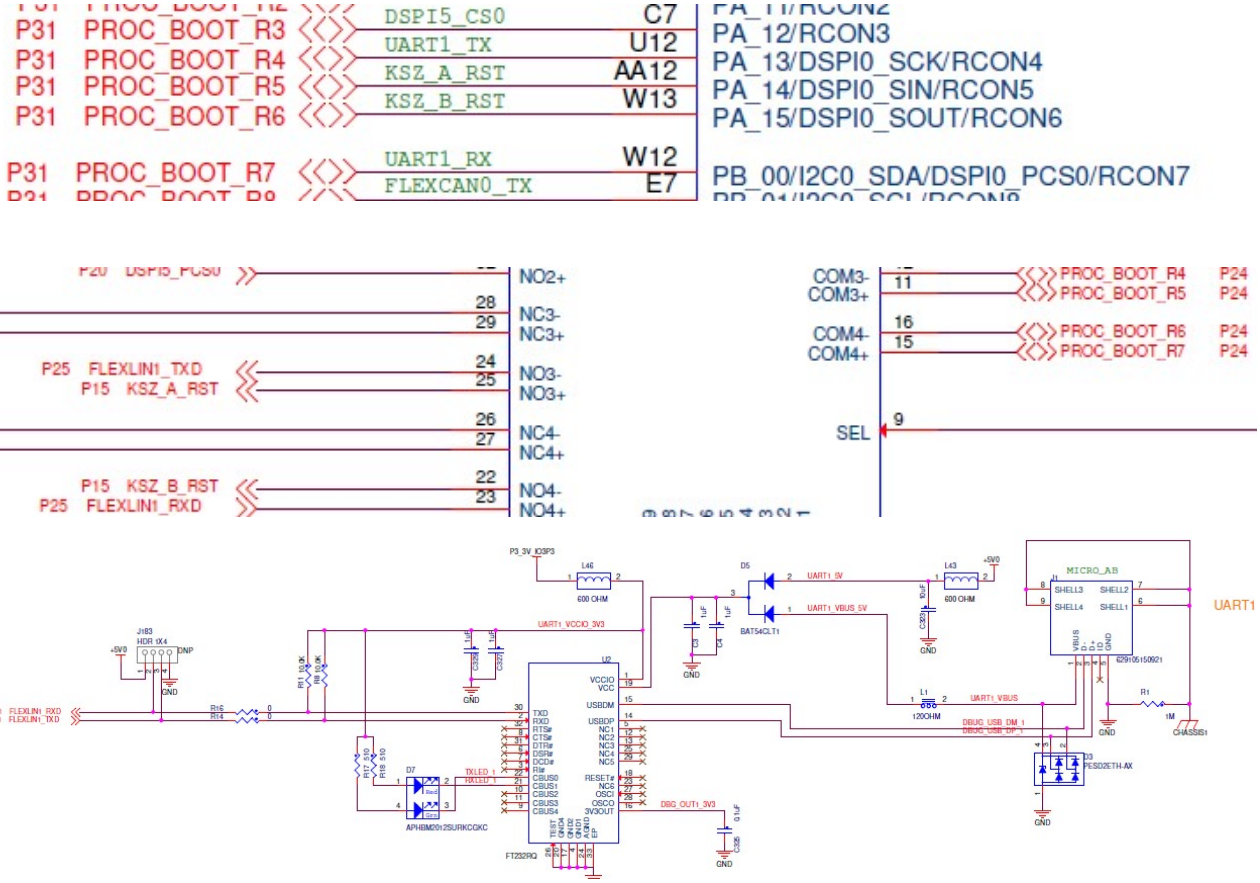
The IOMUX pads for UART communication are configured as shown in the following table.

Table 161. UART external I/O

Signal	PAD name
RX	PAD[42]
TX	PAD[41]

可以看出在 ROM codes 中，默认使用这个 iomux 和这个串口来做为 serial boot mode 的串口，所以这个口除了用做调试串口，又用于下载最初镜像使用的下载串口。

如果希望保留下载串口，而修改调试串口成另一个串口，比如说修改为如下 RDB1 的 UART1:



步骤如下：

首先，目前 Uboot 的串口驱动支持情况：

\configs\s32g274ardb2_defconfig

CONFIG_DM_SERIAL //未定义

CONFIG_FSL_LINFLEXUART=y

\drivers\serial\Makefile

```
ifeq ($(CONFIG_$(SPL_TPL_)BUILD)$(CONFIG_$(SPL_TPL_)DM_SERIAL),yy)
obj-y += serial-uclass.o
else
obj-y += serial.o
endif
ifdef CONFIG_DM_SERIAL
obj-$(CONFIG_FSL_LINFLEXUART) += serial_linflexuart.o
else
obj-$(CONFIG_FSL_LINFLEXUART) += serial_linflexuart_nondm.o
endif
```

所以目前串口驱动没有使用DM，根据以上Uboot启动流程，涉及到的串口初始化分为IOMUX配置和串口初始化部分， 如下 IOMUX 配置部分：注意从 BSP30 开始 Uboot 已经支持 DTS 的 IOMUX 配置，但是串口驱动没有使用，仍然是直接配置寄存器方式

Common\board_f.c

```
init_sequence_f
|->board_early_init_f /board/freescale/s32-gen1/common.c
|   |->setup_iomux_uart(); /board/freescale/s32-gen1/s32g274ardb.c
void setup_iomux_uart(void)
{
#if (CONFIG_FSL_LINFLEX_MODULE == 0)
        /* Muxing for linflex0 */
        setup_iomux_uart0_pc09_pc10();

#elif (CONFIG_FSL_LINFLEX_MODULE == 1)
        /* Muxing for linflex1 */
        /* set PA13 - MSCR[13] - for UART1 TXD */
        writel(SIUL2_MSCR_S32G_G1_PORT_CTRL_UART1_TXD,
            SIUL2_0_MSCRn(SIUL2_PA13_MSCR_S32_G1_UART1));
        /* set PB00 - MSCR[16] - for UART1 RXD */
        writel(SIUL2_MSCR_S32G_G1_PORT_CTRL_UART_RXD,
            SIUL2_0_MSCRn(SIUL2_PB00_MSCR_S32_G1_UART1));
        /* set PB00 - MSCR[736]/IMCR[224] - for UART1 RXD */
        writel(SIUL2_IMCR_S32G_G1_UART1_RXD_to_pad,
            SIUL2_1_IMCRn(SIUL2_PB00_IMCR_S32_G1_UART1));
}
|   |   |->setup_iomux_uart0_pc09_pc10(void)
void setup_iomux_uart0_pc09_pc10(void)
{
        /* Muxing for linflex0 */
        /* set PC09 - MSCR[41] - for UART0 TXD */
        writel(SIUL2_MSCR_S32G_G1_PORT_CTRL_UART0_TXD,
            SIUL2_0_MSCRn(SIUL2_PC09_MSCR_S32_G1_UART0));

        /* set PC10 - MSCR[42] - for UART0 RXD */
```

```
        writel(SIUL2_MSCR_S32G_G1_PORT_CTRL_UART_RXD,
               SIUL2_0_MSCRn(SIUL2_PC10_MSCR_S32_G1_UART0));
        /* set PC10 - MSCR[512]/IMCR[0] - for UART0 RXD */
        writel(SIUL2_IMCR_S32G_G1_UART0_RXD_to_pad,
               SIUL2_0_IMCRn(SIUL2_PC10_IMCR_S32_G1_UART0));
}
```

通过以上代码可以看出，宏 CONFIG_FSL_LINFLEX_MODULE 决定是使用 uart1 或 uart2。

如下串口初始化部分：

```
init_sequence_r(/common/board_r.c)
|->initr_serial
|   |->serial_initialize
|   |   |->serial_assign(default_serial_console()->name);
|   |   |-> default_serial_console(/driver/serial/serial_inflexuart_nondm.c(非 DM 使用此代码))//设备注册

init_sequence_f(/common/board_f.c)
|-> serial_init /* serial communications setup */  /driver/serial/serial.c(非 DM 使用此代码)
return get_current()->start();
|   丨-> /driver/serial/serial_inflexuart_nondm.c(非 DM 使用此代码)
static struct serial_device linflex_serial_drv = {
        .name = "linflex_serial",
        .start = linflex_serial_init,//调用注册的设备初始化文件
```

所以在文件 serial_inflexuart_nondm.c 中，串口驱动的寄存器基地址为：

```
struct linflex_fsl *base = (struct linflex_fsl *)LINFLEXUART_BASE;
```

定义在：

/include/configs/s32-gen1.h

```
#if CONFIG_FSL_LINFLEX_MODULE == 0
#define LINFLEXUART_BASE  LINFLEXD0_BASE_ADDR
#elif CONFIG_FSL_LINFLEX_MODULE == 1
#define LINFLEXUART_BASE  LINFLEXD1_BASE_ADDR
#else
#define LINFLEXUART_BASE  LINFLEXD2_BASE_ADDR
#endif
```

所以宏 CONFIG_FSL_LINFLEX_MODULE 决定调试串口是使用串口 0 还是串口 1。

CONFIG_FSL_LINFLEX_MODULE 来源自：

```
/drivers/serial/Kconfig
config FSL_LINFLEX_MODULE
        int "LINFLEX Active Module"
        depends on FSL_LINFLEXUART
        default 0
        help
          Select the LINFLEX Module used as serial
```

所以可以通过 uboot 的 menuconfig 来修改：如果在 make s32g274ardb2_defconfig 后，直接修改用于编译的.config 也可以：

```
vi .config
CONFIG_FSL_LINFLEX_MODULE=1
make clean //clean 强制重新编译
make -j8
```

将调试串口 usb 线插到 J1 Uart1 的 micro USB 接口上，可见启动打印信息。

最后回过头来看 S32G 的 IOMUX 和 PAD 配置的方法：

串口 0 的 IOMUX ：

```
void setup_iomux_uart0_pc09_pc10(void)
{
    /* Muxing for linflex0 */
    /* set PC09 - MSCR[41] - for UART0 TXD */
    writel(SIUL2_MSCR_S32G_G1_PORT_CTRL_UART0_TXD,
           SIUL2_0_MSCRn(SIUL2_PC09_MSCR_S32_G1_UART0));
    /* set PC10 - MSCR[42] - for UART0 RXD */
    writel(SIUL2_MSCR_S32G_G1_PORT_CTRL_UART_RXD,
           SIUL2_0_MSCRn(SIUL2_PC10_MSCR_S32_G1_UART0));
    /* set PC10 - MSCR[512]/IMCR[0] - for UART0 RXD */
    writel(SIUL2_IMCR_S32G_G1_UART0_RXD_to_pad,
           SIUL2_0_IMCRn(SIUL2_PC10_IMCR_S32_G1_UART0));
}
#define PER_GROUP0_BASE                                 (0x40000000UL)
#define SIUL2_0_BASE_ADDR                       (PER_GROUP0_BASE + 0x009C000)
#define SIUL2_0_MSCR_BASE               (SIUL2_0_BASE_ADDR + 0x00000240)
```

#define SIUL2_0_MSCRn(i) (SIUL2_0_MSCR_BASE + 4 * (i))

所以 SIUL2_0_MSCRn=0x4009c240+4*i

UART0 的 TX/RX 管脚序号如下：

/* TXD */

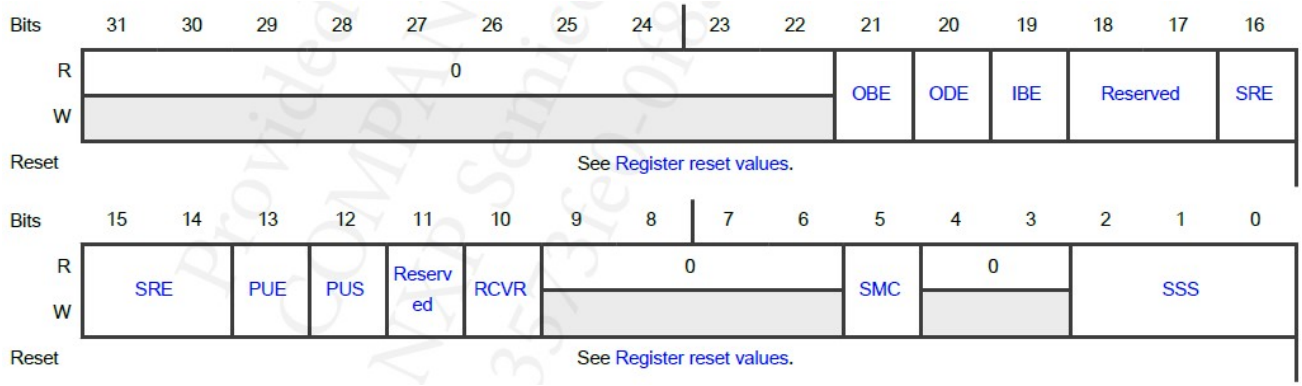
#define SIUL2_PC09_MSCR_S32_G1_UART0 41//所以对应 MSCR 寄存器地址是：0x4009c240+Hex(4*41)= 0x4009c2E4

/* RXD */

#define SIUL2_PC10_MSCR_S32_G1_UART0 42//0x4009c2E8

与之上的 S32G2_IOMUX.xlsx 中的 IO Signal Table 中 IO 的寄存器对应一致。

再看看 MSCR 寄存器定义：



对应源代码：

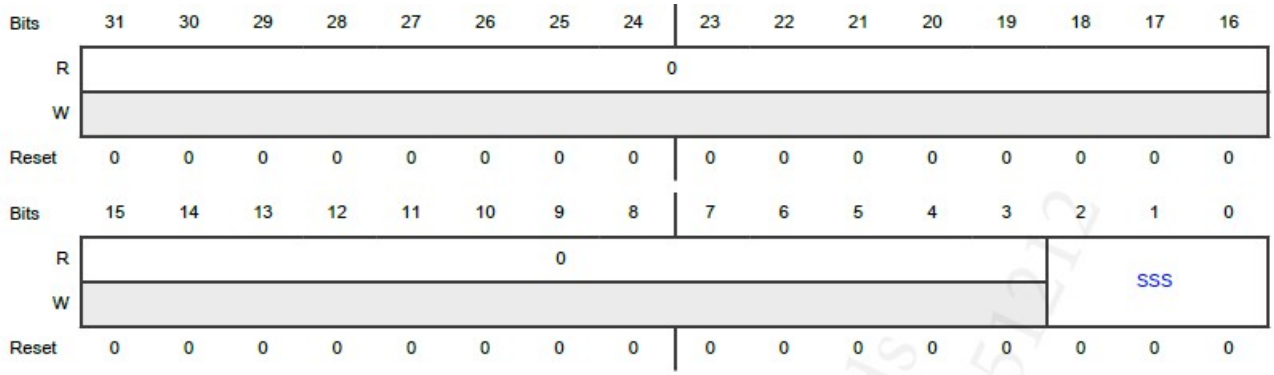
#define SIUL2_MSCR_S32G_G1_PORT_CTRL_UART0_TXD \

(SIUL2_MSCR_S32_G1_SRC_100MHz | // //#define SIUL2_MSCR_S32_G1_SRC_100MHz (5 << 14) =— 101b: Fmax=150 MHz (at 1.8V), 133 MHz (at 3.3V)

SIUL2_MSCR_S32_G1_OBE_EN | //#define SIUL2_MSCR_S32_G1_OBE_EN (1 << 21): 1b - Output driver enabled

SIUL2_MSCR_MUX_MODE_ALT1) //#define SIUL2_MSCR_MUX_MODE_ALT1 (0x1) Selects a function for the pad. Refer to "SSS" column of the 'IO Signal Table' tab of the IOMUX spreadsheet attachment.

RXD 与 UART1 IOMUX 方法一样，注意一下对于输入管脚，还需要说明从输入管脚连接到内部的哪个模块，所以需要多设置一下 IMCR 寄存器，其寄存器定义 如下：



Fields

Field	Function
31-3 —	Reserved
2-0 SSS	Source Signal Select Selects which source signal is connected to the associated destination (chip pin).

只需要设置 SSS 位就可以了，具体的寄存器地址和值请参考 S32G_IOMUX.xlsx 中的 Input Muxing 一章，如下：

Instance	Function	CR	Input CR number	Input SSS value	Source	Signal
LINFlex_0	LIN0_RX	SLUL_CC	512	0		Disable_low
				1		Disable_high
				2	IO_PAD	PC_10
				3	IO_PAD	PA_14
				4	IO_PAD	PL_00

所以如果从 PC_10 输入 UART0_RXD 信号，需要设置寄存器

```
SIUL2_0_IMCRn(SIUL2_PC10_IMCR_S32_G1_UART0));
#define SIUL2_PC10_IMCR_S32_G1_UART0          (512 - 512) //input 的 IMCR 寄存器从 512 开始计数，所以为 512-512=0，第 0 个 IMCR 寄存器
/* IMCR 0-83 */
#define SIUL2_0_IMCRn(i)              (SIUL2_0_IMCR_BASE + 4 * (i))
```

值为：

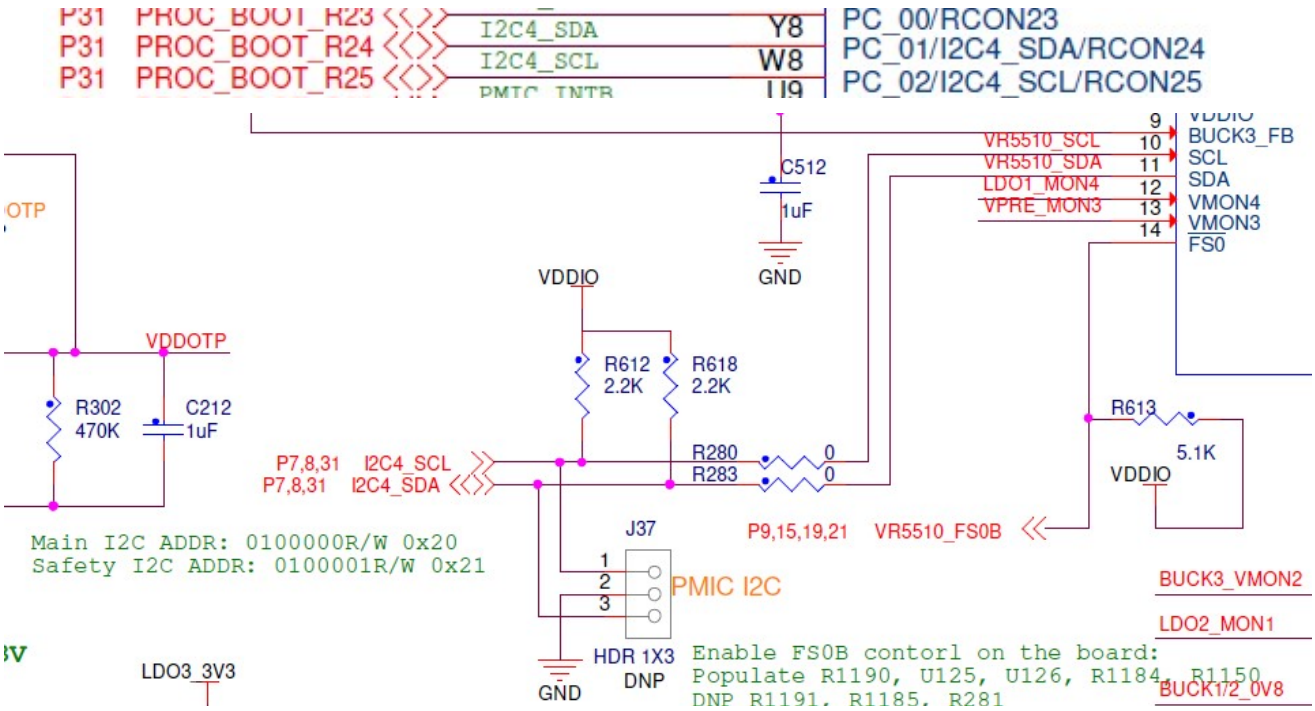
```
#define SIUL2_IMCR_S32G_G1_UART0_RXD_to_pad      (SIUL2_MSCR_MUX_MODE_ALT2)
#define SIUL2_MSCR_MUX_MODE_ALT2 (0x2) //=2=PC_10
```

以上说明了 S32G IOMUX，PAD 设置的方法，之后不再详述，对于默认没有配置的 IO，设置的经验是：

1. 参考相似的模块 IO 来设置新的 IO，比如说，如果是增加一个新的 I2C 口，就参考已有的 I2C 口的配置，根据硬件实际连接的情况，修改相应的 IOMUX 的 IO 管脚的 MSCR 寄存器的地址，SSS 值和输入 INPUT 管脚的 IMCR 寄存器的地址和 SSS 值，当然尽量使用宏定义便于代码阅读。
2. MSCR 中关于 IOPAD 的设置，可以先参考拷贝，比如说新 usdhc 口就参考已有的接口配置来设置，然后再根据实际的接口电平，频率来调整，比如说已有接口可能是 SDcard 3.3V 50mHz 的，而新的是 eMMC 1.8V 200mHZ 的，则需要相应调整驱动能力。

3.6.3 DM I2C 与 PMIC 初始化

I2C 驱动在 Uboot 中的主要应用 就是连接 PMIC，所以通常来讲不建议修改 S32G 系统的电源设计，包括电源芯片和连接接口。注意一下在一些需要 ASIL-B~D 的应用场景中，PMIC 的驱动管理可能会移到 M7 内部中去，所以 A 核心通过 uboot 或内核访问 PMIC 的驱动可能需要去掉，则以下讨论可以帮忙客户了解如何从内核配置和 DTS 中移掉 PMIC 驱动甚至 I2C 驱动。S32G2 RDB2 连接如下：



默认连接到 I2C4,分析其 I2C4 驱动与 PMIC 驱动，以方便客户可能修改 I2C IOMUX 接口或 PMIC.

首先，目前 Uboot 的 I2C 驱动支持情况：

\configs\s32g274ardb2_defconfig

```
CONFIG_CMD_I2C=y
CONFIG_DM_I2C=y //所以I2C实现了DM
CONFIG_SYS_I2C_MXC=y //I2C与i.MX使用同一IP，同一驱动
```

\drivers\i2c\Makefile

```
obj-$(CONFIG_DM_I2C) += i2c-uclass.o
obj-$(CONFIG_SYS_I2C_MXC) += mxc_i2c.o //平台驱动源代码
```

其次： I2C 接口的 IOMUX 已经可以使用 DTS 配置支持：

\arch\arm\dts\fsl-s32g27ardb.dtsi

```
  &pinctrl0 {
                    pinctrl0_i2c4: pinctrl0_i2c4 {
                                fsl,pins = <PC01_MSCR_S32G2XX PC01_I2C4_SDA_CFG
                                            PC02_MSCR_S32G2XX PC02_I2C4_SCL_CFG
                                            >;
                    };
}
&pinctrl1 {
                    pinctrl1_i2c4: pinctrl1_i2c4 {
                                fsl,pins = <I2C4_SDA_IMCR PC01_I2C4_SDA_IN
                                            I2C4_SCL_IMCR PC02_I2C4_SCL_IN
                                            >;
                    };
}

/* PB05 */
#define PB05_MSCR_S32G2XX (21)
#define PB05_I2C2_SCL_CFG    (SIUL2_MSCR_S32_G1_MUX_ID_1 | \
          I2C_PIN_CFG)
#define PB05_I2C2_SCL_IN      (SIUL2_MSCR_S32_G1_MUX_ID_2)

/* PB06 */
#define PB06_MSCR_S32G2XX (22)
#define PB06_I2C2_SDA_CFG   (SIUL2_MSCR_S32_G1_MUX_ID_1 | \
          I2C_PIN_CFG)
#define PB06_I2C2_SDA_IN      (SIUL2_MSCR_S32_G1_MUX_ID_2)
```

```
#define SIUL2_MSCR_S32_G1_MUX_ID_1      (1)
#define SIUL2_MSCR_S32_G1_MUX_ID_2      (2)

#define I2C_PIN_CFG             (SIUL2_MSCR_S32_G1_OBE | \
        SIUL2_MSCR_S32_G1_ODE | SIUL2_MSCR_S32_G1_IBE | \
        SIUL2_MSCR_S32_G1_SRC_25MHz)
```

最后，参考 RDB2 的 DTS 配置

```
/arch/arm/dts/fsl-s32g274ardb2.dts
&i2c4 {pf5020_a, pf5020_b, fs5600
#include "fsl-s32g274ardb.dtsi"
&i2c4 {
        clock-frequency=<100000>; //I2C4 时钟频率设置为 100K
        pinctrl-0 = <&pinctrl0_i2c4 &pinctrl1_i2c4>;
        pinctrl-1 = <&pinctrl0_i2c4_gpio &pinctrl1_i2c4_gpio>;
        scl-gpios = <&gpio0 34 (GPIO_ACTIVE_HIGH | GPIO_OPEN_DRAIN)>;
        sda-gpios = <&gpio0 33 (GPIO_ACTIVE_HIGH | GPIO_OPEN_DRAIN)>;
        pinctrl-names = "default";
        status = "okay";
        vr5510 {
                compatible = "fsl,vr5510"; //常用功能寄存器
                reg = <0x20>;
                status = "okay";
        };
        vr5510_fsu { //function saftey PMIC 地址
                compatible = "fsl,vr5510";
                reg = <0x21>;
                status = "okay";
        };
};
#include "fsl-s32g274a.dtsi" /#include "fsl-s32-gen1.dtsi"
        i2c4: i2c@402DC000 {
                compatible = "fsl,vf610-i2c";
driver/i2c/mxc_i2c.c
```

```
static const struct udevice_id mxc_i2c_ids[] = {
        { .compatible = "fsl,imx21-i2c", },
        { .compatible = "fsl,vf610-i2c", .data = I2C_QUIRK_FLAG, },
        {}
};
```

所以在 S32G RDB2 的设计中，使用 I2C4 来连接所有电源芯片，建议保持此设计，由于 I2C0/1/2/4 的 IOMUX 与 DTS 配置都已经有了，如果需要连接到其它 I2C 接口修改相应的 DTS 就可以，如果需要连接到 I2C4 的不同的 IOMUX，修改其 IOMUX 就可以了。

注意从 BSP30 开始，对设备端拉死 I2C 总线的情况，已经有解除代码如下：

```
driver/i2c/mxc_i2c.c
mxc_i2c_read
|-> bus_i2c_read
|   |-> i2c_init_transfer
|   |   |->i2c_idle_bus  //详细算法请参考以下代码
/*
 * See Linux Documentation/devicetree/bindings/i2c/i2c-imx.txt
 * "
 *  scl-gpios: specify the gpio related to SCL pin
 *  sda-gpios: specify the gpio related to SDA pin
 *  add pinctrl to configure i2c pins to gpio function for i2c
 *  bus recovery, call it "gpio" state
 * "
 *
 * The i2c_idle_bus is an implementation following Linux Kernel.
 */
/*
                        * GPIO pinctrl for i2c force idle is not a must,
                        * but it is strongly recommended to be used.
                        * Because it can help you to recover from bad
                        * i2c bus state. Do not return failure, because
                        * it is not a must.
                        */
            /*
             * In most cases it is just enough to generate 8 + 1 SCLK
```

```
        * clocks to recover I2C slave device from 'stuck' state
        * (when for example SW reset was performed, in the middle of
        * I2C transmission).
        *
        * However, there are devices which send data in packets of
        * N bytes (N > 1). In such case we do need N * 8 + 1 SCLK
        * clocks.
        */
…
```

PMIC 驱动如下：

\configs\s32g274ardb2_defconfig

```
CONFIG_CMD_PMIC=y
CONFIG_DM_PMIC=y //所以PMIC实现了DM
CONFIG_DM_PMIC_VR5510=y //以5510为例
CONFIG_DM_PMIC_PF5020=y
CONFIG_DM_PMIC_FS5600=y
```

\drivers\power\pmic\Makefile

```
obj-$(CONFIG_DM_PMIC) += pmic-uclass.o
obj-$(CONFIG_DM_PMIC_VR5510) += vr5510.o
obj-$(CONFIG_DM_PMIC_PF5020) += pf5020.o
obj-$(CONFIG_DM_PMIC_FS5600) += fs5600.o//平台驱动源代码
```

PMIC 属于 I2C4 的附属设备，所以其 DTS 配置为：

/arch/arm/dts/fsl-s32g274ardb2.dts #include "fsl-s32g274ardb.dtsi"

```
/* PMIC */
&i2c4 {
        vr5510 {
                compatible = "fsl,vr5510";
                reg = <0x20>;//注意与硬件设计的 I2C 寄存器地址匹配
                status = "okay";
        };
        vr5510_fsu {
                compatible = "fsl,vr5510";
                reg = <0x21>;//注意与硬件设计的 I2C 寄存器地址匹配，此为支持功能安全的 PMIC 版本地址
                status = "okay";
        };
};
```

源代码在：

```
    \drivers\power\pmic\vr5510.c
static const struct udevice_id vr5510_ids[] = {
        {.compatible = "fsl,vr5510"},
        {}
};
```

目前 Uboot 启动过程中并没有操作 PMIC,所以默认配置可以启动 S32G RDB2 板，由于已经支持 I2C 和 PMIC uboot 命令，可以如下操作 I2C 接口和 PMIC 接口：

```
=>i2c
i2c - I2C sub-system
Usage:
i2c bus [muxtype:muxaddr:muxchannel] - show I2C bus info
i2c crc32 chip address[.0, .1, .2] count - compute CRC32 checksum
i2c dev [dev] - show or set current I2C bus
i2c loop chip address[.0, .1, .2] [# of objects] - looping read of device
i2c md chip address[.0, .1, .2] [# of objects] - read from I2C device
i2c mm chip address[.0, .1, .2] - write to I2C device (auto-incrementing)
i2c mw chip address[.0, .1, .2] value [count] - write to I2C device (fill)
i2c nm chip address[.0, .1, .2] - write to I2C device (constant address)
i2c probe [address] - test for and show device(s) on the I2C bus
i2c read chip address[.0, .1, .2] length memaddress - read to memory
i2c write memaddress chip address[.0, .1, .2] length [-s] - write memory
        to I2C; the -s option selects bulk write in a single transaction
i2c flags chip [flags] - set or get chip flags
i2c olen chip [offset_length] - set or get chip offset length
i2c reset - re-init the I2C Controller
i2c speed [speed] - show or set I2C bus speed

=>pmic
pmic - PMIC sub-system
Usage:
pmic list          - list pmic devices
pmic dev [name]    - show or [set] operating PMIC device
```

```
pmic dump          - dump registers
pmic read address  - read byte of register at address
pmic write address - write byte to register at address
=> pmic list
| Name                    | Parent name        | Parent uclass @ seq
| vr5510                  | i2c@402DC000       | i2c @ 4
| vr5510_fsu              | i2c@402DC000       | i2c @ 4
| pf5020_a                | i2c@402DC000       | i2c @ 4
| pf5020_b                | i2c@402DC000       | i2c @ 4
| fs5600                  | i2c@402DC000       | i2c @ 4
```

3.6.4 通用 GPIO

通用 GPIO 在 Uboot 中常常用于一些输出应用：如调试灯，外设 reset 信号，外部电源使能信号等，或是不常用于一些输入应用：如二制信号判断，按键等：

S32G 平台 BSP30 默认没有使用 DM_GPIO 和 GPIO 相关驱动，但是有一些硬编码代码可以参考， 如：

/configs/s32g274abluebox3_defconfig

```
CONFIG_BOARD_EARLY_INIT_R=y
```

/common/board_r.c

```
init_sequence_r
#if defined(CONFIG_BOARD_EARLY_INIT_R)
        board_early_init_r,
#endif
|-> board_early_init_r,(board\freescale\s32-gen1\s32g274abluebox3.c)
|   |->release_resets();
static void release_resets(void)
{
        int i;
        static const u8 reset_pins[] = {//用于输出 reset 信号
                SIUL2_MSCR_S32G_PA_12, //12
                SIUL2_MSCR_S32G_PB_02, //18
                SIUL2_MSCR_S32G_PC_00,//32
                SIUL2_MSCR_S32G_PC_05//37
        };
```

```
        for (i = 0; i < sizeof(reset_pins); i++) {
                writeb(SIUL2_GPIO_VALUE1, SIUL2_PDO_N(reset_pins[i]));
/*
#define SIUL2_GPIO_VALUE1               (0x01) //表示拉高
#define SIUL2_PDO_N(i) \
        (i < SIUL2_0_MAX_GPIO ? \ //#define SIUL2_0_S32G274A_MAX_GPIO 112 所以 gpio bank0 有 112 个 gpio
        SIUL2_0_GPDO_N(SIUL2_GPDIO_FOR_GPIO(i)) + \ //#define SIUL2_0_GPDI_N(i)
        (SIUL2_0_GPDI_BASE + 4 * (i)) 表示相应 GPIO 的 GPDI 寄存器地址
// #define SIUL2_GPDIO_FOR_GPIO(i)              (((i) & (~0x3)) >> 2)
```

0 PDI_n	Pad Data In Stores the value of the external GPIO pad associated with this register. PDI_*n* represents PDI[*n*], where *n* is the instance of the register. 0b - Logic low 1b - Logic high

Diagram

Bits	7	6	5	4	3	2	1	0
R	0							PDI_n
W								
Reset	See Register reset values.							

Register reset values

Register	Reset value
GPDI0	01h
GPDI1–GPDI4	00h
GPDI5	01h
GPDI6–GPDI101	00h
GPDI102–GPDI103	Register not supported

GPDI 寄存器只有一个，高或低，默认重启时的拉高拉低如下表。

```
        SIUL2_GPDIO_PDIO_OFF_FOR_GPIO(i) : \
```

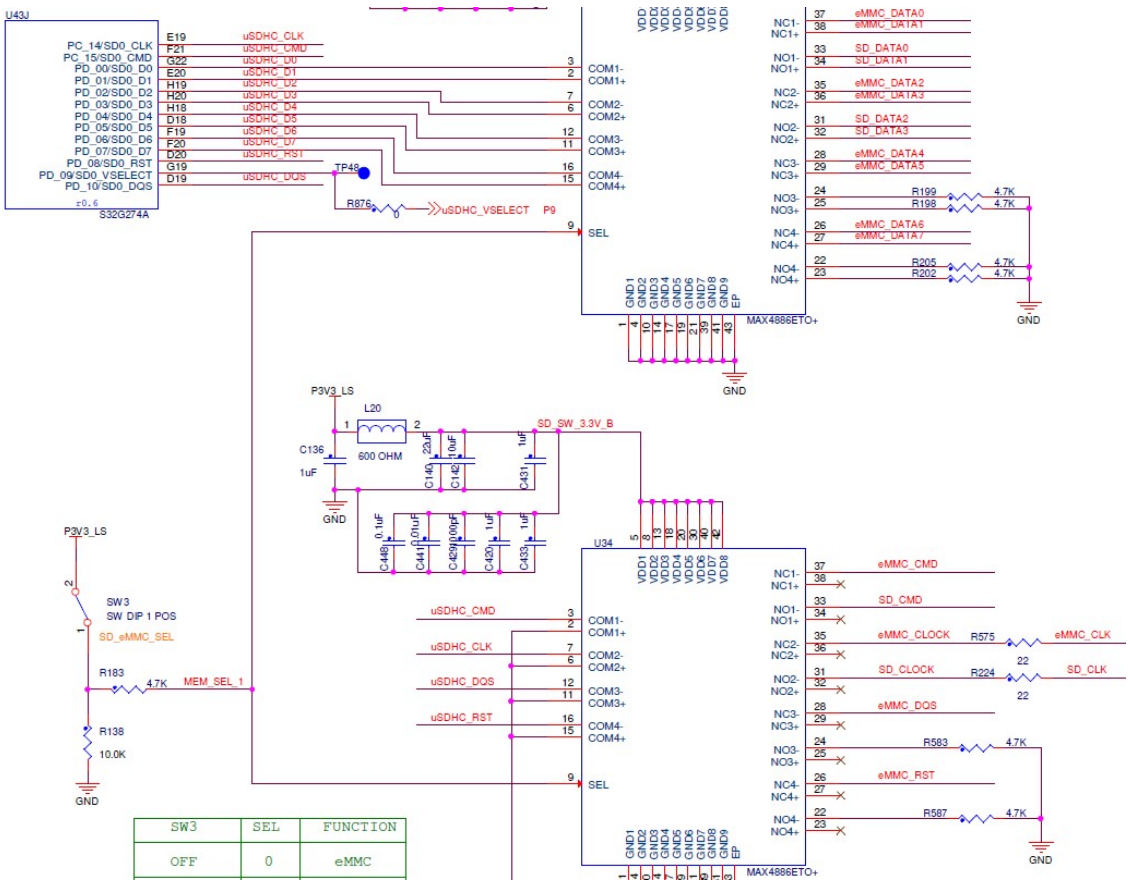
```
        SIUL2_1_GPDO_N(SIUL2_GPDIO_FOR_GPIO(i)) + \ //以下相同
        SIUL2_GPDIO_PDIO_OFF_FOR_GPIO(i)) */
//iomux 配置
                writel(SIUL2_MSCR_GPO_G1, SIUL2_0_MSCRn(reset_pins[i]));
/*
#define SIUL2_MSCR_GPO_G1 \
        (SIUL2_MSCR_MUX_MODE_ALT0 | \ 配置这样管脚 ALT0， ALT0 一般会 GPIO
         SIUL2_MSCR_S32_G1_OBE_EN)
*/
        }
}
```

//输入的示例如下，不过目前代码中只有 IO 配置，没有读取的代码：

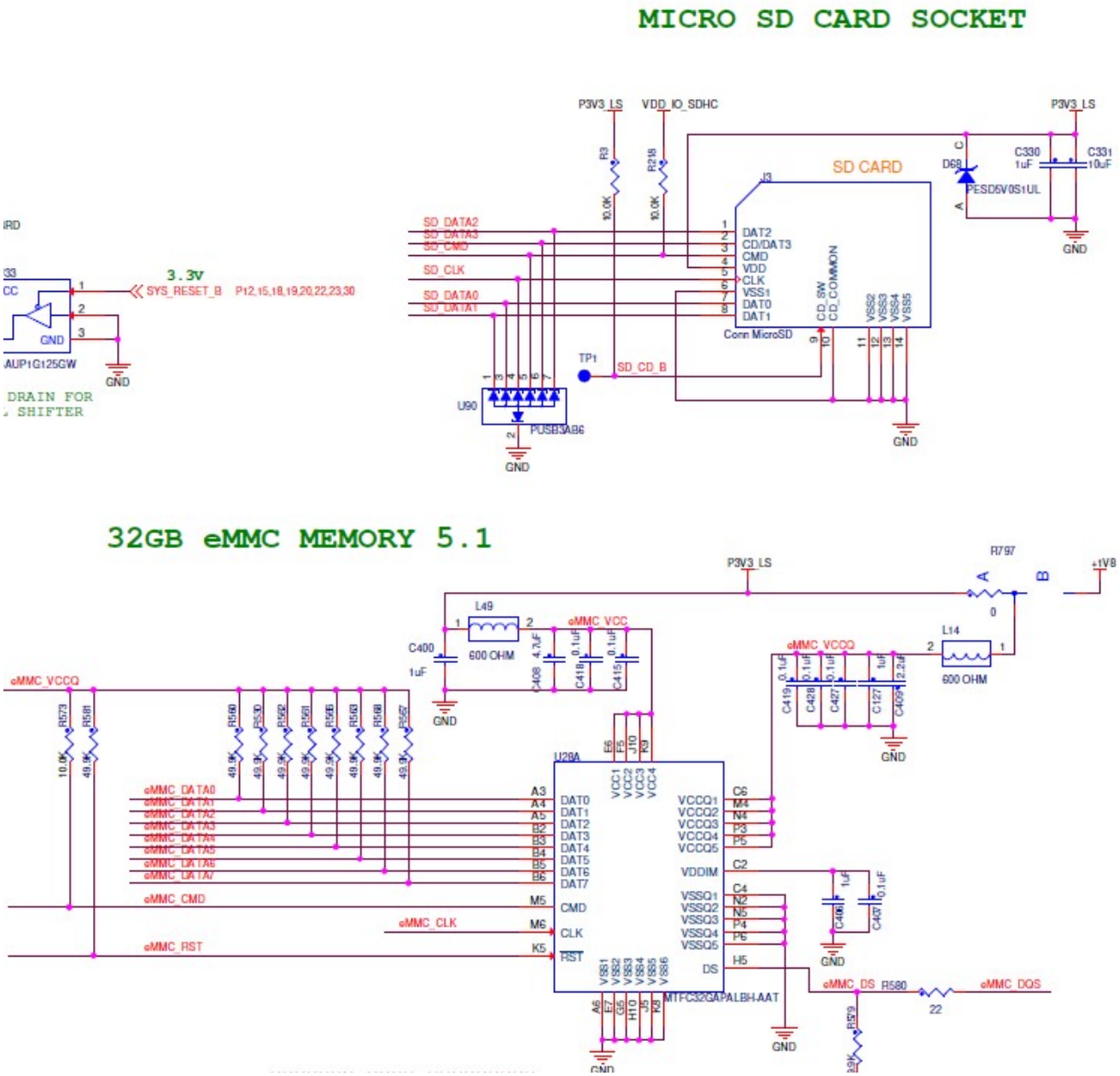
```
setup_iomux_input_gpios();
static void setup_iomux_input_gpios(void)
{
        int i;
        for (i = 0; i < input_pins_size ; i++) {
                if (input_pins[i].nr < SIUL2_0_MAX_GPIO)
                        writel(SIUL2_MSCR_GPI_G1,
                                SIUL2_0_MSCRn(input_pins[i].nr));
                else
                        writel(SIUL2_MSCR_GPI_G1,
                                SIUL2_1_MSCRn(input_pins[i].nr));
        }
}
static const struct pin input_pins[] = { //大部分用于检测电源 good 的 GPIO
        {SIUL2_MSCR_S32G_PA_09, "S32G_SD_CD_B"},
…
        {SIUL2_MSCR_S32G_PL_14, "S32G_FS85_LX_PGOOD"},
}
```

3.6.5 启动 eMMC 定制

S32G 仅支持一个 uSDHC 接口，所以不存在修改 uSDHC 的可能，S32G RDB2 是有一个分路器件来支持 SDcard/eMMC 的，如下：



SW3	SEL	FUNCTION
OFF	0	eMMC



仅有可能修改其 IO,其 IO 配置的代码从 BSP30 开始已经移动到 DTS 中配置，由于只有一个 uSDHC 口，所以一般不会修改其 IOMUX 配置，只是 IOPAD 配置需要注意：

\arch\arm\dts\fsl-s32g274ardb.dtsi

```
pinctrl0_sd0: pinctrl0_sd0 {
    fsl,pins = <PC14_MSCR_S32G2XX PC14_SD0_CLK_CFG
                PC15_MSCR_S32G2XX PC15_SD0_CMD_CFG
                PD00_MSCR_S32G2XX PD00_SD0_D0_CFG
```

```
…
                                PD07_MSCR_S32G2XX PD07_SD0_D7_CFG
                                PD08_MSCR_S32G2XX PD08_SD0_RST_CFG
                                PD09_MSCR_S32G2XX PD09_SD0_VSELECT_CFG
                                PD10_MSCR_S32G2XX PD10_SD0_DQS_CFG
                                SD0_CMD_IMCR PC15_SD0_CMD_IN
                                SD0_D0_IMCR PD00_SD0_D0_IN
…
                                SD0_D7_IMCR PD07_SD0_D7_IN
                                SD0_DQS_IMCR PD10_SD0_DQS_IN
                                >;
        };

#define PC14_SD0_CLK_CFG        (SIUL2_MSCR_S32_G1_MUX_ID_1 | \
    SIUL2_MSCR_S32_G1_SRC_208_1V8_166_3V3_MHZ | SIUL2_MSCR_S32_G1_OBE | \
    SIUL2_MSCR_S32_G1_PUE | SIUL2_MSCR_S32_G1_SMC_DIS)

#define PD00_SD0_D0_CFG                 (SIUL2_MSCR_S32_G1_MUX_ID_1 | \
    SD0_D_PIN_CFG)

#define SD0_D_PIN_CFG           (SIUL2_MSCR_S32_G1_SRC_208_1V8_166_3V3_MHZ | \
     SIUL2_MSCR_S32_G1_OBE | SIUL2_MSCR_S32_G1_IBE | \
     SIUL2_MSCR_S32_G1_PUE | SIUL2_MSCR_S32_G1_PUS | \
     SIUL2_MSCR_S32_G1_SMC_DIS)

#define SD0_CMD_PIN_CFG                 SD0_D_PIN_CFG

#define PD10_SD0_DQS_CFG        (SIUL2_MSCR_S32_G1_SRC_208_1V8_166_3V3_MHZ | \
    SIUL2_MSCR_S32_G1_IBE | SIUL2_MSCR_S32_G1_SMC_DIS)
```

总结一下就是

- CLK 是输出管脚，数据和命令是输入输出管脚，DQS 是输入 管脚，所以相应的 OBE/IBE 配置不同。

- 数据和命令线有上拉处理(原理图上也设计了外部上拉)，CLK 则是默认下拉，DQS 未设置上下拉(原理图上也设计了外部下拉)
- 最主要要考虑的是 SRE 的配置，如下：

16-14 SRE	Slew Rate Control • For "3.3 V/1.8 V" FAST pads: — 000b: Fmax= 208 MHz (at 1.8V), 166 MHz (at 3.3V)

Field	Function
	— 011b: Reserved — 100b: Fmax=166 MHz (at 1.8V), 150 MHz (at 3.3V) — 101b: Fmax=150 MHz (at 1.8V), 133 MHz (at 3.3V) — 110b: Fmax=133 MHz(at 1.8V), 100 MHz (at 3.3V) — 111b, Fmax=83 MHz (at 1.8V), 63 MHz (at 3.3V)

可见默认配置是针对高速 eMMC 的，对于低速 eMMC 或 SDcard 是否有过冲 EMI 或信号质量问题要考虑，可以根据实际频率需要来修改。一般来讲，此处是唯一要考虑定制的情况。

目前 S32GRDB2 板的设计的 eMMC 是与 SDcard 一样，用 3.3V 供电的，对于 3.3V 供电的 SDcard，目前在 3.3V 供电的情况下仅支持：

1.Default Speed mode: 3.3V 供电模式，频率上限 25MHz，速度上限 12.5MB/sec

2.High Speed mode: 3.3V 供电模式，频率上限 50MHz，速度上限 25MB/sec

所以如果是 Debug 情况下使用 SDcard,设置为：

#define SIUL2_MSCR_S32_G1_SRC_133_1V8_100_3V3_MHZ (6 << 14)

#define SIUL2_MSCR_S32_G1_SRC_83_1V8_63_3V3_MHZ (7 << 14)

更合理。

而产品中实际实用 eMMC 居多， 如下：

Table 4 — Bus Speed Modes

Mode Name	Data Rate	I/O Voltage	Bus Width	Frequency	Max Data Transfer (implies x8 bus width)
Backwards Compatibility with legacy MMC card	Single	3 V/1.8 V/ 1.2 V	1, 4, 8	0-26 MHz	26 MB/s
High Speed SDR	Single	3 V/1.8 V/ 1.2 V	1,4, 8	0-52 MHz	52 MB/s
High Speed DDR	Dual	3 V/1.8 V/ 1.2 V	4, 8	0-52 MHz	104 MB/s
HS200	Single	1.8 V/1.2 V	4, 8	0-200 MHz	200 MB/s
HS400	Dual	1.8 V/1.2 V	8	0-200 MHz	400 MB/s

所以如目前 S32GRDB2 板的设计的 eMMC 是与 SDcard 一样，用 3.3V 供电的，则理论上设置为：

```
#define SIUL2_MSCR_S32_G1_SRC_166_1V8_150_3V3_MHZ      (4 << 14)
#define SIUL2_MSCR_S32_G1_SRC_150_1V8_133_3V3_MHZ      (5 << 14)
```

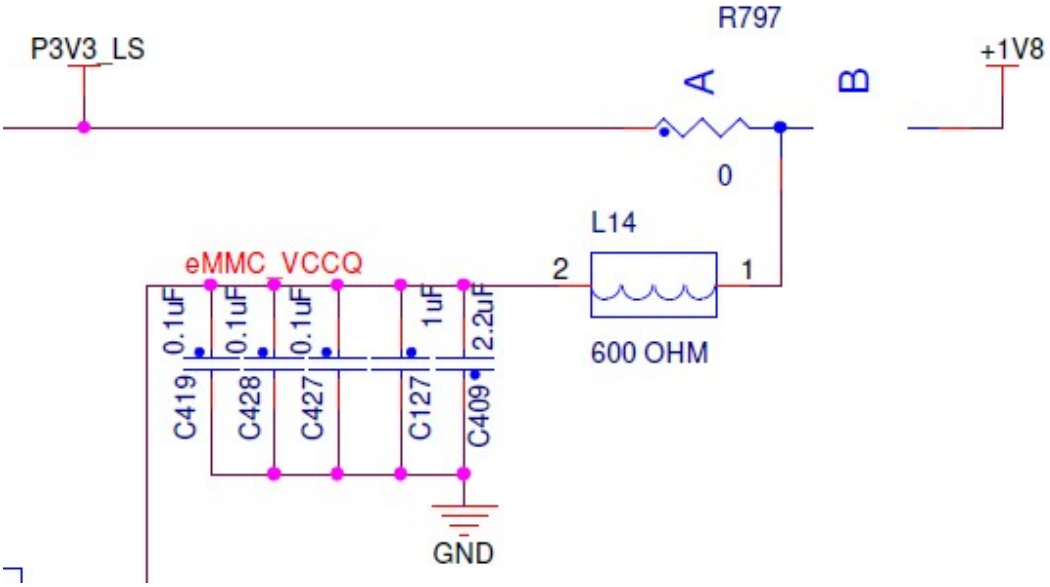
即可。

从 BSP30 开始，支持 eMMC 1.8V HS400 模式，则 IOPAD 需要设置为：

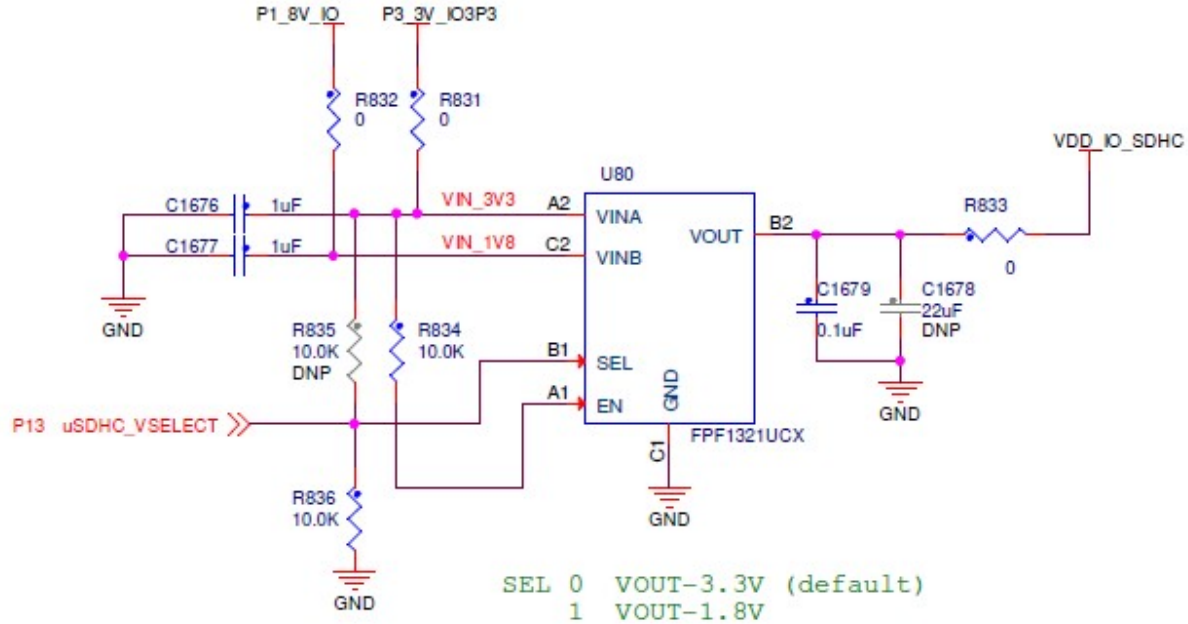
```
#define SIUL2_MSCR_S32_G1_SRC_208_1V8_166_3V3_MHZ      (0 << 14)
```

注意如 BSP30 参考手册说明，修改成 HS400 需要修改(注意由于默认 S32GRDB2 板上连接的是 SDcard，所以以下未测试)：

- 硬件：R797 去掉 A 连接 B,使用 1V8 给 IO 供电。



去掉 R836，连接 R835.



SD_CLK 上串联的电阻也需要去掉。

- DTS 修改为：

\arch\arm\dts\fsl-s32-gen1.dts

```
usdhc0: usdhc@402F0000{
        compatible = "fsl,s32gen1-usdhc";
        …
        bus-width = <8>;
        mmc-hs400-1_8v;
        /delete-property/ no-1-8-v;
        …
};
```

如源代码\drivers\mmc\mmc-uclass.c

```
if (dev_read_bool(dev, "mmc-hs400-1_8v"))
        cfg->host_caps |= MMC_CAP(MMC_HS_400);
    if (dev_read_bool(dev, "mmc-hs400-enhanced-strobe"))
        cfg->host_caps |= MMC_CAP(MMC_HS_400_ES);
```

所以增加 host controller 支持 HS400 功能， HS400_ES 还不支持。

- DTS 中的高频 IOPAD 也需要增加：

\arch\arm\dts\fsl-s32g274ardb.dtsi

```
&usdhc0 {
        /* By default sd0 pins were able to work at 100Mhz and 200Mhz */
        pinctrl-0 = <&pinctrl0_sd0>;
        pinctrl-1 = <&pinctrl0_sd0>; //add
        pinctrl-2 = <&pinctrl0_sd0>;//add
        pinctrl-names = "default", "state_100mhz", "state_200mhz";
        status = "okay";
};
```

在 debug 过程中如果要参考 Uboot 的 usdhc 驱动源代码，如下：

\configs\s32g274ardb2_defconfig

```
CONFIG_CMD_MMC=y
CONFIG_ENV_IS_IN_MMC=y
CONFIG_DM_MMC=y //所以 mmc 驱动实现了 DM
CONFIG_FSL_USDHC=y
```

\drivers\mmc\Makefile

```
obj-$(CONFIG_$(SPL_)DM_MMC) += mmc-uclass.o
obj-$(CONFIG_FSL_ESDHC) += fsl_esdhc.o /平台驱动源代码
```

\arch\arm\dts\fsl-s32g274ardb2.dts #include "fsl-s32g274ardb.dtsi" #include "fsl-s32g274a.dtsi"

```
        usdhc0: usdhc@402F0000{
                compatible = "fsl,s32gen1-usdhc", "fsl,esdhc";
                reg = <0x0 0x402F0000 0x0 0x1000>;
                interrupts = <0 36 4>;
                clocks = <&clks S32GEN1_SCMI_CLK_USDHC_MODULE>,
                         <&clks S32GEN1_SCMI_CLK_USDHC_AHB>,
                         <&clks S32GEN1_SCMI_CLK_USDHC_CORE>;
                clock-names = "ipg", "ahb", "per";
                clock-frequency = <0>;   /* updated dynamically if 0 */
                bus-width = <8>; //默认为 8，如果使用 SDcard 可以改成 4
                status = "disabled";
        };
```

源代码：

/drivers/mmc/fsl_esdhc_imx.c

```
static const struct udevice_id fsl_esdhc_ids[] = {
        { .compatible = "fsl,esdhc", },
        { /* sentinel */ }
};
```

因为 有 mmc 命令支持，所以可以使用 uboot 命令查看支持的情况，如下：

```
mmc
mmc - MMC sub system
Usage:
mmc info - display info of the current MMC device
mmc read addr blk# cnt
mmc write addr blk# cnt
mmc erase blk# cnt
mmc rescan
mmc part - lists available partition on current mmc device
mmc dev [dev] [part] - show or set current mmc device [partition]
mmc list - lists available devices
```

```
mmc hwpartition [args...] - does hardware partitioning
  arguments (sizes in 512-byte blocks):
    [user [enh start cnt] [wrrel {on|off}]] - sets user data area attributes
    [gp1|gp2|gp3|gp4 cnt [enh] [wrrel {on|off}]] - general purpose partition
    [check|set|complete] - mode, complete set partitioning completed
  WARNING: Partitioning is a write-once setting once it is set to complete.
  Power cycling is required to initialize partitions after set to complete.
mmc setdsr <value> - set DSR register value
=> mmc info
Device: FSL_SDHC
Manufacturer ID: 3
OEM: 5344
Name: SL16G
Bus Speed: 50000000
Mode: SD High Speed (50MHz)
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 14.8 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
```

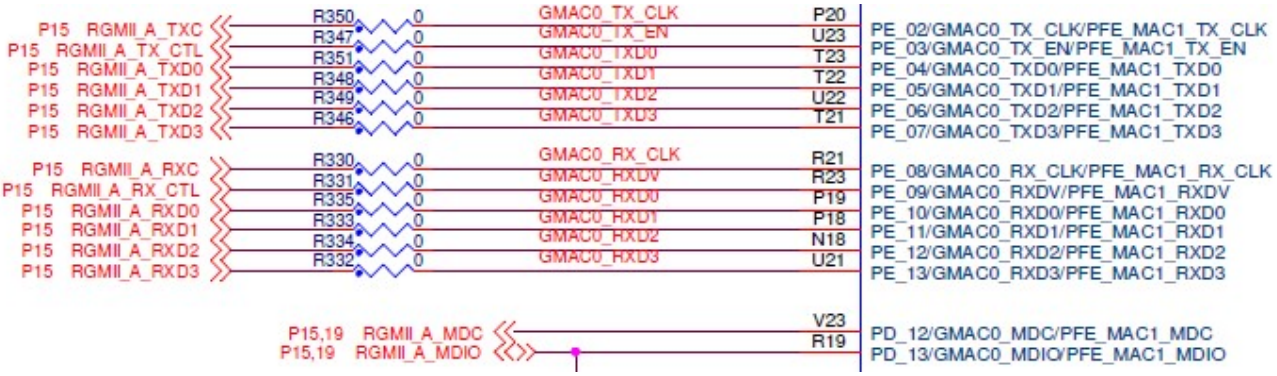
3.6.6 Ethernet 定制

做为汽车网关处理器，S32G 的以太网相关非常重要，不过在 uboot 中，通常仅用于调试或升级使用，所以此处仅介绍属于 A 核的 GMAC 的以太网驱动定制，目前 uboot 代码也仅支持 GMAC 连接的 PHY，不支持 PFE。

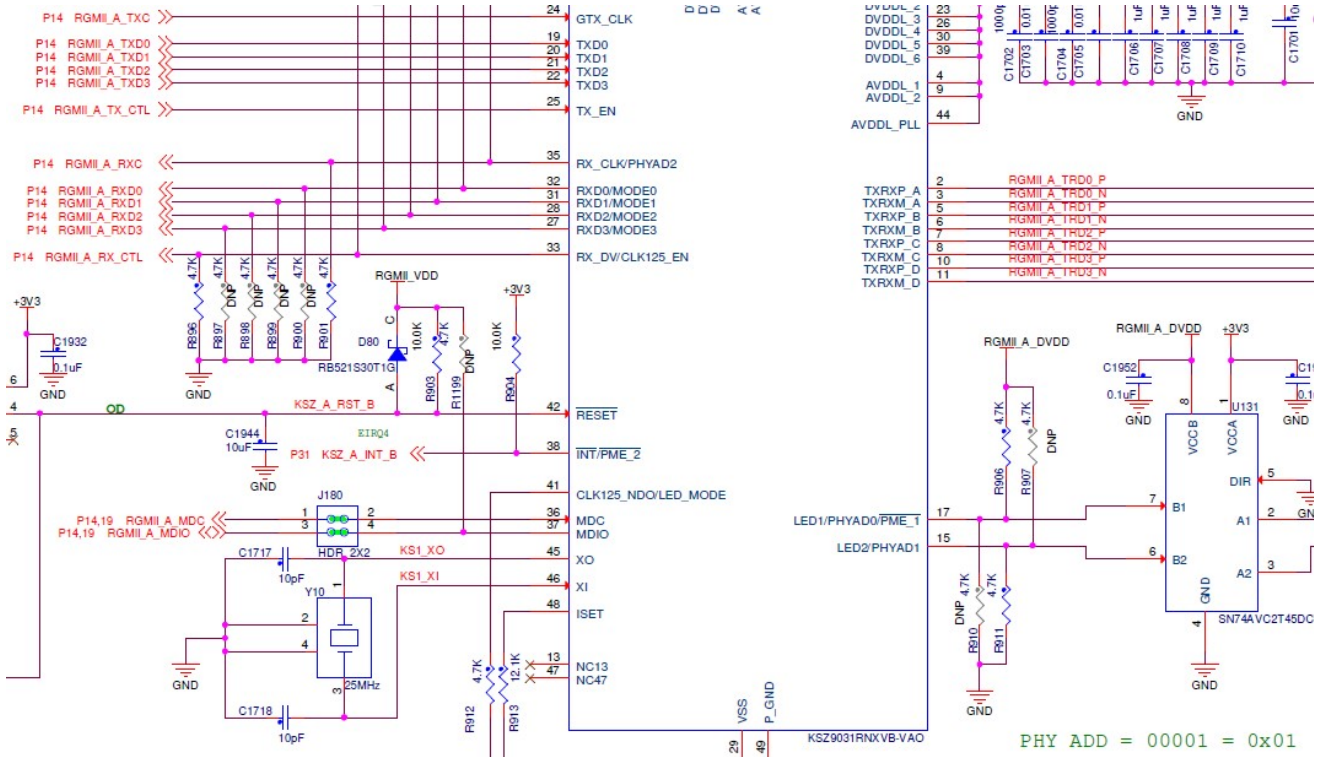
GMAC 通过 RGMII 连接到了一颗 PHY KSZ9031 上，通过 SGMII 连接到了一颗 PHY ARQ107 上。

S32G RDB2 的以太网 PHY 硬件连接如下：

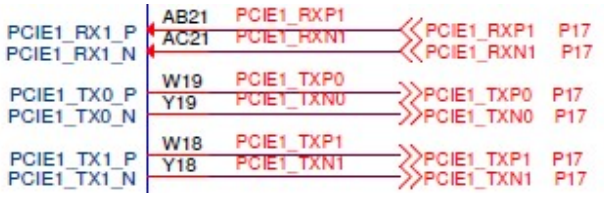
GMAC0 的 RGMII_A 接口及 MDIO:

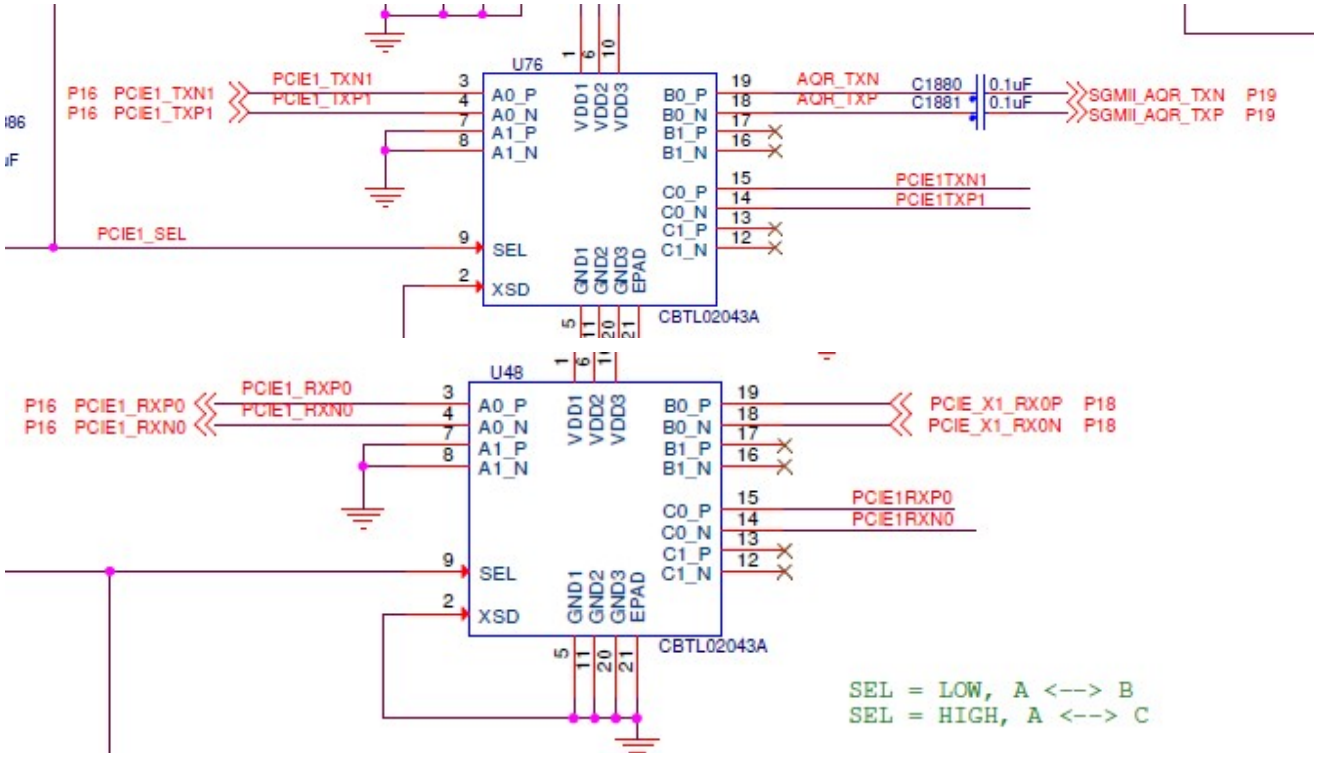


RGMII 接口+MDIO 连接到 KSZ9031 上，及这颗 KSZ9031 设置的 PHY 地址=0x1。



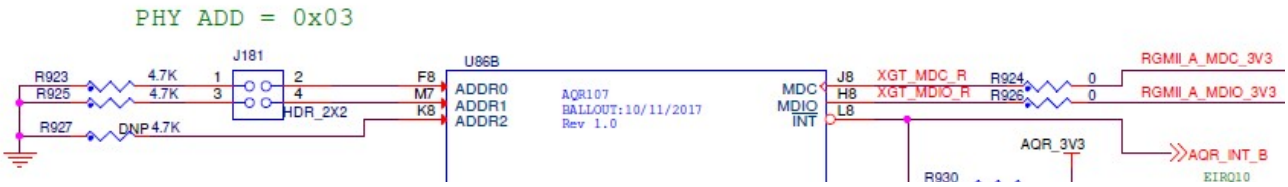
GMAC0 的 SGMII 接口及 MDIO:



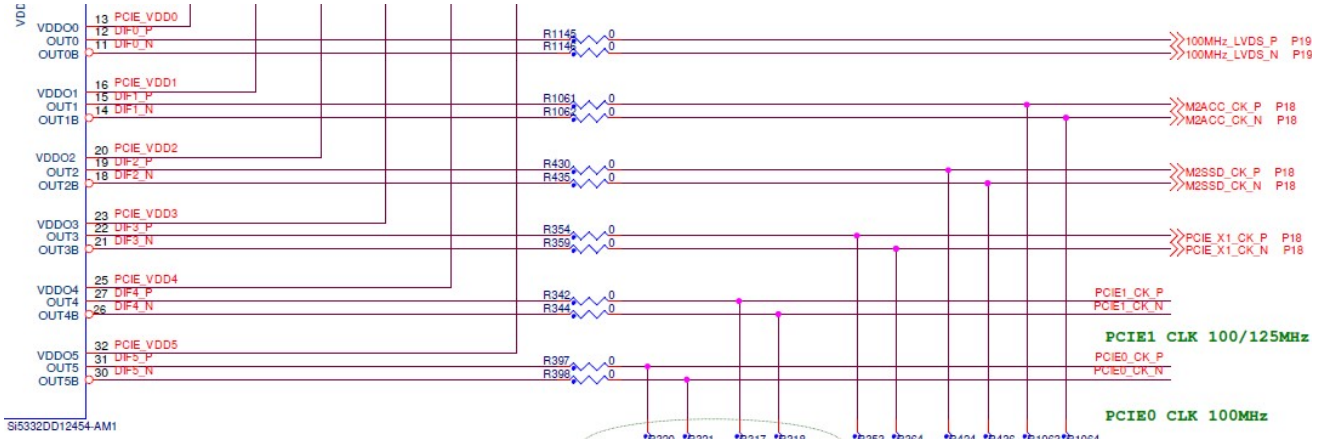


连接到 AQR107 上:

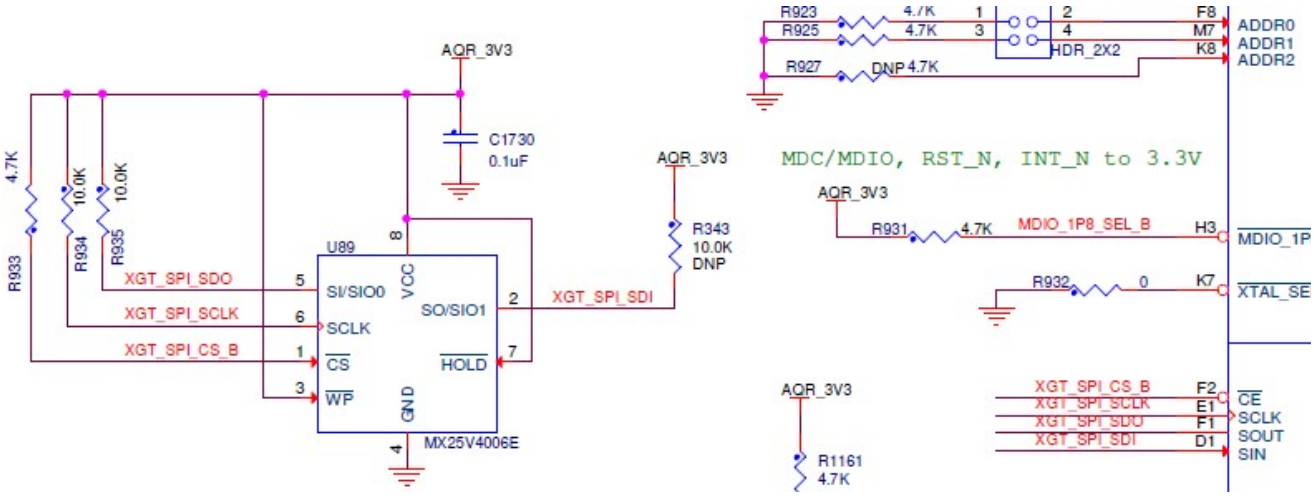
MDIO 及 PHY 地址设置如下：



其 clock 由一颗外部专用 SerDas 时钟芯片提供：



其 firmware 使用一颗 SPI NOR 储存，不需要 HOST 下载：



以下驱动分析：

内核配置：

\configs\s32g274ardb2_defconfig

CONFIG_PHY_ATHEROS=y

CONFIG_PHY_MICREL=y

CONFIG_PHY_MICREL_KSZ90X1=y //支持 KSZ90X1 驱动

CONFIG_PHY_FIXED=y //支持外接 switcher 芯片，目前 uboot 中没有使用

CONFIG_DM_ETH=y //支持 ethernet DM

CONFIG_FSL_PFENG=y //支持 PFE engine，目前 uboot 中没有使用

CONFIG_FSL_PFENG_EMAC_1_RGMII=y

CONFIG_DWC_ETH_QOS_DEVICES=y //以下与 GMAC0 RGMII 接口相关

CONFIG_DWC_ETH_QOS_S32CC=y

CONFIG_RGMII=y

CONFIG_PCI=y //以下与 GMAC0 的 SGMII 接口相关，和 PCIe 复用 Serdas

CONFIG_DM_PCI=y

CONFIG_PCIE_S32GEN1=y

Makefile：

\drivers\net\Makefile

```
ccflags-$(CONFIG_DWC_ETH_QOS_S32CC) += -Iarch/$(ARCH)/include/asm/arch-s32/s32-gen1
ccflags-$(CONFIG_DWC_ETH_QOS_S32CC) += -Idrivers/clk/s32/include
ccflags-$(CONFIG_DWC_ETH_QOS_S32CC) += -Iboard/freescale/s32-gen1
obj-$(CONFIG_DWC_ETH_QOS_S32CC) += dwc_eth_qos_core.o dwc_eth_qos_s32cc.o
obj-$(CONFIG_FSL_PFE) += pfe_eth/
obj-$(CONFIG_FSL_PFENG) += pfeng/
```

\drivers\net\phy\ Makefile

```
obj-$(CONFIG_PHY_MICREL_KSZ90X1) += micrel_ksz90x1.o
obj-$(CONFIG_PHY_FIXED) += fixed.o
obj-$(CONFIG_PHY_AQUANTIA) += aquantia.o
```

DTS:

\arch\arm\dts\fsl-s32g274ardb2.dts 及其包含：

```
//gmac0 的芯片层代码，一般不需要修改
gmac0: ethernet@4033c000 {
                compatible = "fsl,s32cc-dwmac";
                reg = <0x0 0x4033c000 0x0 0x2000>, /* gmac IP */
                      <0x0 0x4007C004 0x0 0x4>;    /* S32 CTRL_STS reg */
                tx-fifo-depth = <20480>;
                rx-fifo-depth = <20480>;
                status = "disabled";
                gmac0_mdio: mdio0 {
                        compatible = "snps,dwmac-mdio";
                };
        };
//gmac0 的 clock tree
&gmac0 {
        clocks = <&clks S32GEN1_SCMI_CLK_GMAC0_RX_SGMII>,
```

```
                <&clks S32GEN1_SCMI_CLK_GMAC0_TX_SGMII>,
                <&clks S32GEN1_SCMI_CLK_GMAC0_TS_SGMII>,
                <&clks S32GEN1_SCMI_CLK_GMAC0_RX_RGMII>,
                <&clks S32GEN1_SCMI_CLK_GMAC0_TX_RGMII>,
                <&clks S32GEN1_SCMI_CLK_GMAC0_TS_RGMII>,
                <&clks S32GEN1_SCMI_CLK_GMAC0_AXI>;
        clock-names = "rx_sgmii", "tx_sgmii", "ts_sgmii",
                      "rx_rgmii", "tx_rgmii", "ts_rgmii",
                      "axi";
    }

    &gmac0 {
      status = "okay";
      phy-mode = "rgmii";
      /* Connected to KSZ9031 MDIO_A */
      phy-handle = <&mdio_a_phy1>; //这个表示 gmac 通过 mdio_a 接口通过 rgmii 协议连接到了 KSZ9031 上。
    };
    &gmac0_mdio {
      #address-cells = <1>;
      #size-cells = <0>;
      /* KSZ9031 GMAC */
      mdio_a_phy1: ethernet-phy@1 {
        max-speed = <1000>;
        reg = <1>; // KSZ9031 PHY 地址设置为 0x1，与硬件设计一致
      };
      /* ARQ107 */
      mdio_a_phy3: ethernet-phy@3 {// mdio_a_phy3 没有设置为 gmac0 的 phy-handle，所以不能工作，默认没有使能。
        compatible = "ethernet-phy-ieee802.3-c45";
        reg = <3>; //ARQ107 设置为 0x3，与硬件设计一致
      };
    };
```

所以默认 uboot 中仅使用 KSZ9031 的 PHY:

需要注意的地方如上 PHY 的地址设置要和硬件设计一致，使用的 MDIO 接口要一致，另外就是如下的 RGMII 与 MDIO 的 IOMUX 配置要与硬件设计一致：

\drivers\net\dwc_eth_qos_s32cc.c

```
static const struct udevice_id eqos_ids[] = {//dwc_eth_qos_dm.h
                .compatible = "fsl,s32cc-dwmac",
                .data = (ulong)&eqos_s32cc_config
        },
struct eqos_config eqos_s32cc_config = {
…
        .ops = &eqos_s32cc_ops
};
static struct eqos_ops eqos_s32cc_ops = {
…
        .eqos_start_clks = eqos_start_clks_s32cc,
…
};
eqos_start_clks_s32cc
|-> setup_iomux_enet_gmac
```

\board\freescale\s32-gen1\eth.c

```
void setup_iomux_enet_gmac(struct udevice *dev, int intf)
{…
case PHY_INTERFACE_MODE_RGMII:
                pinctrl_select_state(dev, "gmac_rgmii");
                break;
…}
```

从 BSP30 开始，支持 DM 的驱动的 IOMUX 配置已经移到 DTS 中配置：

Arch\arm\dts\fsl-s32g.dtsi

```
&gmac0 {…
        pinctrl-0 = <&pinctrl0_gmac0 &pinctrl0_gmac0_mdio>;
        pinctrl-1 = <&pinctrl0_gmac0_mdio>;
        pinctrl-names = "gmac_rgmii", "gmac_sgmii";
};
                pinctrl0_gmac0: pinctrl0_gmac0 {
                        fsl,pins = <PE02_MSCR_S32G2XX PE02_GMAC0_TX_CLK_CFG
```

```
                                PE03_MSCR_S32G2XX PE03_GMAC0_TX_EN_CFG
                                PE04_MSCR_S32G2XX PE04_GMAC0_TX_D0_CFG
/*
/* PE04 */
#define PE04_MSCR_S32G2XX       (68) //MSCR 寄存器序号
#define PE04_GMAC0_TX_D0_CFG  (SIUL2_MSCR_S32_G1_MUX_ID_1 | \
      ENET_OUT_PIN_CFG)
#define SIUL2_MSCR_S32_G1_MUX_ID_1    (1) //IOMUX 值
#define ENET_OUT_PIN_CFG        (SIUL2_MSCR_S32_G1_OBE) //IOPAD 设置， 仅设置为输出
*/
…
                                PE07_MSCR_S32G2XX PE07_GMAC0_TX_D3_CFG
                                PE08_MSCR_S32G2XX PE08_GMAC0_RX_CLK_CFG
                                PE09_MSCR_S32G2XX PE09_GMAC0_RX_DV_CFG
                                PE10_MSCR_S32G2XX PE10_GMAC0_RX_D0_CFG
…
                                PE13_MSCR_S32G2XX PE13_GMAC0_RX_D3_CFG
                                GMAC0_TX_CLK_IMCR PE02_GMAC0_TX_CLK_IN
                                GMAC0_RX_CLK_IMCR PE08_GMAC0_RX_CLK_IN
                                GMAC0_RX_DV_IMCR PE09_GMAC0_RX_DV_IN
                                GMAC0_RX_D0_IMCR PE10_GMAC0_RX_D0_IN
/*
#define GMAC0_RX_D0_IMCR        (531) //对输入管脚，除了 MSCR 外，还需要设置 IMCR 寄存器
#define PE10_GMAC0_RX_D0_IN     (SIUL2_MSCR_S32_G1_MUX_ID_2)
#define SIUL2_MSCR_S32_G1_MUX_ID_2    (2) //IMCR 仅需要配置输入到那个模块
*/
…
                                GMAC0_RX_D3_IMCR PE13_GMAC0_RX_D3_IN
                                >;
                }
                pinctrl0_gmac0_mdio: pinctrl0_gmac0_mdio {
                        fsl,pins = <PD12_MSCR_S32G2XX PD12_GMAC0_MDC_CFG
                                PD13_MSCR_S32G2XX PD13_GMAC0_MDIO_CFG
                                GMAC0_MDIO_IMCR PD13_GMAC0_MDIO_IN
```

```
                                        >;
                };
```

如下分析一下网口驱动加载过程，并注明具体要在 debug 中检查的地方：

Drivers\net\dwc_eth_qos_core.c

```
.of_match = of_match_ptr(eqos_ids),
.ofdata_to_platdata = of_match_ptr(eqos_ofdata_to_platdata),
.probe = eqos_probe,
.ops = &eqos_ops,
…
1: static const struct udevice_id eqos_ids[] =      { // Include\dm\platform_data\dwc_eth_q
                .compatible = "fsl,s32cc-dwmac",
                .data = (ulong)&eqos_s32cc_config
        }
2: eqos_ofdata_to_platdata
|->
/* DT: parse phy-mode */ // pdata->eth.phy_interface=rgmii
If /* DT: check for fixed-link subnode */
Else /* DT: parse phy-handle */ and      /* DT: parse max-speed */
// eqos->phy_addr=reg
// pdata->eth.max_speed= max-speed
/* DT: allow rewrite platform specific t/rx-fifo-depth */
// pdata->config->tx_fifo_size
// pdata->config->rx_fifo_size
3:eqos_probe
|-> eqos_probe_resources_core
|->     eqos->mii->read = eqos_mdio_read;
        eqos->mii->write = eqos_mdio_write;
|-> mdio_register
4: static const struct eth_ops eqos_ops = {
        .start = eqos_start,…
eqos_start
|-> phy_connect
|    |-> phy_find_by_mask
```

| | |->get_phy_device_by_mask

| | | |->create_phy_by_mask

| | | | |->get_phy_id //drivers/net/phy/phy.c: 此函数是非常重要的函数，MAC通过MDIO接口，根据PHY ADD的设置第一次读取 PHY 的 ID 寄存器：

```
/*
        * Grab the bits from PHYIR1, and put them
        * in the upper half
        */
       phy_reg = bus->read(bus, addr, devad, MII_PHYSID1);
       if (phy_reg < 0)
                return -EIO;
       *phy_id = (phy_reg & 0xffff) << 16;
       /* Grab the bits from PHYIR2, and put them in the lower half */
       phy_reg = bus->read(bus, addr, devad, MII_PHYSID2);
       if (phy_reg < 0)
                return -EIO;
       *phy_id |= (phy_reg & 0xffff);
```

在 get_phy_id 函数中增加打印，确认一下读出的 PHY ID 是否正确，如果与 datasheet 对比正确，则表示 PHY 已经正常工作了。

参考 KSZ9031 数据手册可能如下：

Address	Name	Description	Mode Note 4-1	Default
2.15:0	PHY ID Number	Assigned to Bits [3:18] of the organizationally unique identifier (OUI). KENDIN Communication's OUI is 0010A1h.	RO	0022h
Register 3h - PHY Identifier 2				
3.15:10	PHY ID Number	Assigned to Bits [19:24] of the organizationally unique identifier (OUI). KENDIN Communication's OUI is 0010A1h.	RO	0001_01

通常如果得到了 PHY ID，驱动会根据这个 ID 号去找 PHY 的驱动函数，执行其 probe 来初始化 PHY 驱动，所以对于 C22 的 PHY，就可以如上所说在对应厂商的驱动文件中确认或加入新的 PHY 支持。

因为 PHY 的驱动已经事先注册成功了，如下：

Common\board_r.c\

init_sequence_r

|-> initr_net

```
|   |-> eth_initialize
|   |   |-> eth_common_init
|   |   |   |->phy_init
|   |   |   |   |->phy_micrel_ksz90x1_init();
phy_register(&ksz9031_driver);
static struct phy_driver ksz9031_driver = {
        .name = "Micrel ksz9031",
        .uid = 0x221620, //此处与从 get_phy_id 打印出来的数据对比，如果正确说明 PHY 已经正确工
```

作了，PHY 的工作正常需要保证

- 电源正常
- 时钟正常
- Reset 正常
- PHY ADDRESS 设置正确，MDIO 的 IOMUX 设置正确，这样才能保证读到 PHY 的 ID 寄存器。

  如果 PHY 的访问正确了，基本说明 PHY 已经工作了，以下就要完善 PHY 的访问函数，如果有现在的最好，如果没有就需要自己开发。

```
        .mask = 0xfffff0,
        .features = PHY_GBIT_FEATURES,
        .config   = &ksz9031_config,
        .startup  = &ksz90xx_startup,
        .shutdown = &genphy_shutdown,
        .writeext = &ksz9031_phy_extwrite,
        .readext = &ksz9031_phy_extread,
};
```

所以总结一下在 Uboot 中支持一个以太网 PHY 芯片驱动的步骤是：

1. 硬件检查 PHY 芯片的电源，时钟， reset(reset 有可能是硬件拉动的，也可能是软件 gpio 拉动的，如果是用软件拉注意一下电平要求)，使用万用表或示波器。
2. 检查 MDIO 的 IOMUX 设置和 PHY 的地址设置，代码和 DTS 中的设置要与之匹配。
3. 确认 get_phy_id 是否已经拿到了 phy id，如果拿到了，说明 phy 已经开始工作了。
4. 在 drivers\net\phy\下检查是否已经有此 phy 的驱动源代码，然后在 phy.c 中需要在初始化时调用 phy 的初始化：

   int phy_init(void)

   { phy_micrel_ksz90x1_init();…}

5. 在 phy 驱动代码的 phy_driver 数据结构中检查其 uid 是否与 get_phy_id 的值相同，如果相同，则此 phy 驱动会被自动加载。
6. phy_driver 数据结构包含了 phy 的配置，读写，与电源管理函数等，这样函数的会通过 MDIO 操作 phy，要与 phy 的数据手册对应看看相应的寄存器操作是否正确。
7. 所以如上所说，如果是支持一块新的 phy 芯片，则可以从相近 phy 驱动源文件中拷贝一份，按照以下说所配置其 uid，实现所有的 phy_driver 访问函数。

   本节是说明 phy 驱动的开发情况，switcher 芯片的 fixed link 说明以后增加。

   KSZ9031 的 uboot 命令情况如下：

3.6.7 S32G254/234 的支持

根据文档《S32G_LinuxBSP30.0.0_Release_Notes.pdf》说明：

SoC	Component and Version	Boards supported & tested
S32G274A	Linux v5.10.41 U-Boot v2020.04 Yocto 3.2 ARM Trusted Firmware v2.3 Xen v4.14.0 OP-TEE v3.11	S32G274A EVB S32G274A RDB2
S32G254A	Linux v5.10.41 U-boot v2020.04 Yocto 3.2 ARM Trusted Firmware v2.3 Xen v4.14.0 OP-TEE v3.11	S32G254A EVB
S32G233A	Linux v5.10.41 U-boot v2020.04 Yocto 3.2 ARM Trusted Firmware v2.3 Xen v4.14.0 OP-TEE v3.11	S32G233A EVB

This Linux BSP has been built and tested using GCC 10.2 toolchain.

由于 EVB 板有 socket 板，所以很容易更换芯片来支持 S32G254/234，从 BSP30 开始有正式代码支持，并且为自动支持的方式：

如下步骤说明了如何在 S32G274 RDB2 板上如何测试 254：

1: burn the *.sdcard in the tfcard.

2: hack the uboot codes. As follows:

arch\arm\include\asm\arch-s32\siul.h:

```
static inline enum s32g2_derivative get_s32g2_derivative(void)
{…

    switch (deriv) {
    case 0x1D12U:
        return S32G254A_DERIV ; //johnli test S32G274A_DERIV;
    case 0x1CFEU:
        return S32G254A_DERIV;
```

3: burn the uboot the tfcard.

```
sudo dd if=u-boot.s32 of=/dev/sdc bs=256 count=1 seek=0
sync
sudo dd if=u-boot.s32 of=/dev/sdc bs=512 seek=1 skip=1
sync
```

4: boot

Uboot log:

```
CPU: NXP S32G254A rev. 2.1.0
Reset cause: Power-On Reset
Model: NXP S32G2XX
Board:  NXP S32G254A-RDB
DRAM:  3.5 GiB
CA53 core 2 running.
All (2) cores are up.
…
```

Kernel log:

```
Auto Linux BSP 1.0 s32g274ardb2 ttyLF0
s32g274ardb2 login: root
```

```
root@s32g274ardb2:~# cat /proc/cpuinfo
processor       : 0
BogoMIPS        : 10.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

processor       : 1
BogoMIPS        : 10.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4
```

233 如下：

arch\arm\include\asm\arch-s32\siul.h:

```
static inline enum s32g2_derivative get_s32g2_derivative(void)
{…

    switch (deriv) {
    case 0x1D12U:
            return S32G233A_DERIV ; //johnli test S32G274A_DERIV;
```

boot:

```
U-Boot 2020.04-dirty (Mar 23 2021 - 13:14:40 +0800)
CPU: NXP S32G233A rev. 2.1.0
Reset cause: Power-On Reset
Model: NXP S32G2XX
Board:  NXP S32G233A-RDB
DRAM:  3.5 GiB
CA53 core 2 running.
```

All (2) cores are up.

内核仍是两核。

所以如果使用 RDB2 的 BSP，则芯片会自动读取内部寄存器定义来加载不同数目的 A 核，不需要特别修改代码，只是内核中的提示符可以修改一下。

3.7 Uboot debug 信息

3.7.1 Print env

Boot 2020.04+g7eba18e1c0 (Aug 19 2021 - 15:37:50 +0000)

CPU: NXP S32G274A rev. 2.1.0

Reset cause: Power-On Reset //此外表明此次启动为上电，或掉电重启

Model: NXP S32G2XX

Board: NXP S32G274A-RDB

DRAM: 3.5 GiB //内存大小 enable ECC 下的情况

CA53 core 1 running.

CA53 core 2 running.

CA53 core 3 running.

All (4) cores are up.//4 核启动

MMC: FSL_SDHC: 0

Loading Environment from MMC... OK

Using external clock for PCIe0, CRNS

Configuring PCIe0 as RootComplex(x2)

Using external clock for PCIe1, CRNS

Frequency 125Mhz configured for PCIe1

Configuring PCIe1 as SGMII(x2) [XPCS0 2.5G, XPCS1 OFF]

PCIe0: Failed to get link up

Pcie0: LINK_DBG_1: 0x00000000, LINK_DBG_2: 0x00000800 (expected 0x000000d1)

DEBUG_R0: 0x000a2900, DEBUG_R1: 0x08200000

PCI: Failed autoconfig bar 20

PCI: Failed autoconfig bar 24

PCIe1: Not configuring PCIe, PHY not configured

In: serial

Out: serial

Err: serial

Board revision: RDB2/GLDBOX Revision C

Net: EQOS phy: rgmii @ 1

Warning: eth_eqos (eth0) using random MAC address - 3e:ff:c2:de:c2:94

eth0: eth_eqos PFE: emac0: sgmii emac1: none emac2: rgmii

Structure length mismatch: found 0 required 176

Warning: eth_pfeng using MAC address from ROM

, eth1: eth_pfeng

Hit any key to stop autoboot: 0

=> pri

arch=arm

baudrate=115200//串口波特率

board=s32-gen1//板

board_name=s32-gen1//板名

board_rev=C

boot_fdt=try

boot_mtd=booti

bootargs=console=ttyLF0,115200 root=/dev/ram rw earlycon

bootcmd=pfeng stop; mmc dev ${mmcdev}; if mmc rescan; then if run loadimage; then run mmcboot; else run netboot; fi; else run netboot; fi

bootdelay=3

bootscript=echo Running bootscript from mmc ...; source

console=ttyLF0

cpu=armv8

ddr_limit0=0xE0000000;

eth1addr=00:01:be:be:ef:11

ethact=eth_pfeng

fdt_addr=0x83E00000 //fdt 加载的地址

fdt_file=fsl-s32g274a-rdb2.dtb //此处为内核需要加载使用的 DTB 文件。

fdt_high=0xa0000000

fdtcontroladdr=fcd2fa18

flashboot=echo Booting from flash...; run flashbootargs;sf probe 6:0;sf read ${loadaddr} ${kernel_flashaddr} ${kernel_maxsize};sf read ${fdt_addr} ${fdt_flashaddr} ${fdt_maxsize};sf read ${ramdisk_addr} ${ramdisk_flashaddr} ${ramdisk_maxsize};${boot_mtd} ${loadaddr} ${ramdisk_addr} ${fdt_addr};

flashbootargs=setenv bootargs console=${console} root=/dev/ram rw earlycon nohz=off coherent pool=64M ;setexpr uboot_flashaddr (0x0 + 0x0);setexpr kernel_flashaddr (0x0 + 0x1f0000);setenv kernel_maxsize 0xa00000;setexpr fdt flashaddr (0x0 + 0xbf0000);setenv fdt_maxsize 0x100000;setexpr ramdisk_flashaddr (0x0 + 0xcf0000);setenv ramdisk_maxsize 0x2000000;

hwconfig=pcie0:mode=rc,clock=ext;pcie1:mode=sgmii,clock=ext,fmhz=125,xpcs_mode=2G5

image=Image

initrd_high=0xffffffff

ipaddr=10.0.0.100

jtagboot=echo Booting using jtag...; ${boot_mtd} ${loadaddr} ${ramdisk_addr} ${fdt_addr}

jtagsdboot=echo Booting loading Linux with ramdisk from SD...; run loadimage; run loadramdisk; run loadfdt;${boot_mtd} ${loadaddr} ${ramdisk_addr} ${fdt_addr}

loadaddr=0x80080000

loadbootscript=fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${script};

loadfdt=fatload mmc ${mmcdev}:${mmcpart} ${fdt_addr} ${fdt_file};

loadimage=fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${image}

loadramdisk=fatload mmc ${mmcdev}:${mmcpart} ${ramdisk_addr} ${ramdisk}

loadtftpfdt=tftp ${fdt_addr} ${fdt_file};

loadtftpimage=tftp ${loadaddr} ${image};

loadtftpramdisk=tftp ${ramdisk_addr} ${ramdisk};

mem=4169728k

mmcargs=setenv bootargs console=${console},${baudrate} root=${mmcroot} earlycon nohz=off coherent_pool=64M

mmcboot=echo Booting from mmc ...; run mmcargs; run loadimage; if run loadfdt; then ${boot_mtd} ${loadaddr} - ${fdt_addr}; else echo WARN: Cannot load the DT; fi;

mmcdev=0 //默认使用的 emmc 接口，S32G 只有一个接口

mmcpart=1 //默认使用的 boot partition. Mmcblk0p1 是放的 kernel 与 dtb

mmcroot=/dev/mmcblk0p2 rootwait rw

netargs=setenv bootargs console=${console},${baudrate} root=/dev/nfs ip=dhcp nfsroot=${serverip}:${nfsroot},v3,tcp earlycon nohz=off coherent_pool=64M

netboot=echo Booting from net ...; run netargs; if test ${ip_dyn} = yes; then setenv get_cmd dhcp; else setenv get_cmd tftp; fi; ${get_cmd} ${image}; if test ${boot_fdt} = yes || test ${boot_fdt} = try; then if ${get_cmd} ${fdt_addr} ${fdt_file}; then ${boot_mtd} ${loadaddr} - ${fdt_addr}; else if test ${boot_fdt} = try; then ${boot_mtd}; else echo WARN: Cannot load the DT; fi; fi; else ${boot_mtd}; fi;

netmask=255.255.255.0

nfsboot=echo Booting from net using tftp and nfs...; run nfsbootargs;run loadtftpimage; run loadtftpfdt;${boot_mtd} ${loadaddr} - ${fdt_addr};

nfsbootargs=setenv bootargs console=${console},${baudrate} root=/dev/nfs rw ip=${ipaddr}:${serverip}::${netmask}::eth0:off nfsroot=${serverip}:/tftpboot/rfs,nolock,v3,tcp earlycon nohz=off coherent_pool=64M

pfe1_phy_addr=3

pfe1addr=00:01:be:be:ef:22

pfe2addr=00:01:be:be:ef:33

pfeaddr=00:01:be:be:ef:11

pfeng_mode=enable,sgmii,none,rgmii

pfengemac=2

ramdisk=rootfs.uimg

ramdisk_addr=0x84000000

s32cc_gmac_mode=enable

script=boot.scr

serverip=10.0.0.1

soc=s32

stderr=serial

stdin=serial

stdout=serial

update_sd_firmware=if test ${ip_dyn} = yes; then setenv get_cmd dhcp; else setenv get_cmd tftp; fi; if mmc dev ${mmcdev}; then if ${get_cmd} ${update_sd_firmware_filename}; then setexpr fw_sz ${filesize} / 0x200; setexpr fw_sz ${fw_sz} + 1; mmc write ${loadaddr} 0x8 ${fw_sz}; fi; fi

update_sd_firmware_filename=u-boot.s32

vendor=freescale

Environment size: 3922/8188 bytes

3.7.2 dm - Driver model low level access

```
=> dm tree
 Class     Index  Probed  Driver                Name
-----------------------------------------------------------
 root          0  [ + ]   root_driver           root_driver
 clk           0  [ + ]   fixed_rate_clock      |-- fxosc@40050000
 clk           1  [ + ]   fixed_rate_clock      |-- firc@0
 clk           2  [ + ]   fixed_rate_clock      |-- sirc@0
 clk           3  [   ]   fixed_rate_clock      |-- ftm0_ext@0
 clk           4  [   ]   fixed_rate_clock      |-- ftm1_ext@0
 clk           5  [ + ]   fixed_rate_clock      |-- gmac0_ext_rx@0
 clk           6  [   ]   fixed_rate_clock      |-- gmac0_ext_tx@0
```

```
clk          7 [   ]   fixed_rate_clock      |-- gmac0_ext_ref@0
clk          8 [   ]   fixed_rate_clock      |-- gmac0_ext_ts@0
clk          9 [ + ]   fixed_rate_clock      |-- serdes0_lane0_ext_cdr@0
clk         10 [ + ]   fixed_rate_clock      |-- serdes0_lane0_ext_tx@0
clk         11 [ + ]   fixed_rate_clock      |-- serdes0_lane1_ext_cdr@0
clk         12 [ + ]   fixed_rate_clock      |-- serdes0_lane1_ext_tx@0
clk         13 [ + ]   fixed_rate_clock      |-- serdes1_lane0_ext_cdr@0
clk         14 [ + ]   fixed_rate_clock      |-- serdes1_lane0_ext_tx@0
clk         15 [ + ]   fixed_rate_clock      |-- serdes1_lane1_ext_cdr@0
clk         16 [ + ]   fixed_rate_clock      |-- serdes1_lane1_ext_tx@0
clk         17 [ + ]   s32gen1_clk           |-- clks
clk         27 [   ]   s32gen1_clk           |   |-- fxosc@40050000
clk         28 [   ]   s32gen1_clk           |   |-- mc_cgm0@40030000
clk         29 [   ]   s32gen1_clk           |   |-- mc_me@40088000
clk         30 [   ]   s32gen1_clk           |   |-- rdc@440080000
clk         31 [   ]   s32gen1_clk           |   |-- rgm@40078000
clk         32 [   ]   s32gen1_clk           |   |-- mc_cgm1@40034000
clk         33 [   ]   s32gen1_clk           |   |-- mc_cgm2@44018000
clk         34 [   ]   s32gen1_clk           |   |-- mc_cgm5@40068000
clk         35 [   ]   s32gen1_clk           |   |-- armpll@4003800
clk         36 [   ]   s32gen1_clk           |   |-- periphpll@4003C000
clk         37 [   ]   s32gen1_clk           |   |-- accelpll@40040000
clk         38 [   ]   s32gen1_clk           |   |-- ddrpll@40044000
clk         39 [   ]   s32gen1_clk           |   |-- armdfs@40054000
clk         40 [   ]   s32gen1_clk           |   `-- armdfs@40058000
mmc          0 [ + ]   fsl-esdhc-mmc         |-- usdhc@402F0000
blk          0 [ + ]   mmc_blk               |   `-- usdhc@402F0000.blk
pci_generi   0 [ + ]   serdes_s32gen1        |-- serdes@40480000
pci          0 [ + ]   pci_s32gen1           |-- pcie@40400000
pci_generi   2 [   ]   pci_generic_drv       |   `-- pci_0:0.0
spi          0 [   ]   fsl_dspi              |-- dspi1@401D8000
spi          1 [   ]   fsl_dspi              |-- dspi5@402D0000
spi          2 [   ]   fsl_qspi              |-- quadspi@40134000
spi_flash    0 [   ]   spi_flash_std         |   `-- mx25uw51245g@0
```

```
eth          0  [ + ]   eth_eqos             |-- eth_eqos
i2c          0  [   ]   i2c_mxc             |-- i2c@401E4000
i2c          1  [   ]   i2c_mxc             |-- i2c@401E8000
i2c          2  [   ]   i2c_mxc             |-- i2c@401EC000
i2c          3  [ + ]   i2c_mxc             |-- i2c@402DC000
pmic         0  [   ]   vr5510_pmic         |   |-- vr5510
pmic         1  [ + ]   vr5510_pmic         |   |-- vr5510_fsu
pmic         2  [   ]   pf5020_pmic         |   |-- pf5020_a
pmic         3  [   ]   pf5020_pmic         |   |-- pf5020_b
pmic         4  [   ]   fs5600_pmic         |   `-- fs5600
misc         0  [ + ]   s32gen1-ocotp       |-- ocotp@400A4000
pci_generi   1  [ + ]   serdes_s32gen1      |-- serdes@44180000
pci          1  [ + ]   pci_s32gen1         |-- pcie@44100000
clk         18  [   ]   fixed_rate_clock    |-- pfe_mac0_ext_rx@0
clk         19  [   ]   fixed_rate_clock    |-- pfe_mac0_ext_tx@0
clk         20  [ + ]   fixed_rate_clock    |-- pfe_mac0_ext_ref@0
clk         21  [   ]   fixed_rate_clock    |-- pfe_mac1_ext_rx@0
clk         22  [   ]   fixed_rate_clock    |-- pfe_mac1_ext_tx@0
clk         23  [ + ]   fixed_rate_clock    |-- pfe_mac1_ext_ref@0
clk         24  [   ]   fixed_rate_clock    |-- pfe_mac2_ext_rx@0
clk         25  [   ]   fixed_rate_clock    |-- pfe_mac2_ext_tx@0
clk         26  [ + ]   fixed_rate_clock    |-- pfe_mac2_ext_ref@0
eth          1  [ + ]   eth_pfeng           |-- eth_pfeng
simple_bus   0  [ + ]   generic_simple_bus  |-- siul2_0@4009C000
pinctrl      0  [ + ]   s32_pinctrl         |   |-- siul2-pinctrl0
pinconfig    0  [ + ]   pinconfig           |   |   |-- board_generic_pinctrl0
pinconfig    1  [ + ]   pinconfig           |   |   |   |-- pinctrl0_gmac0_mdio
pinconfig    2  [ + ]   pinconfig           |   |   |   |-- pinctrl0_gmac0
pinconfig    3  [ + ]   pinconfig           |   |   |   |-- pinctrl0_pfe0_mdio
pinconfig    4  [   ]   pinconfig           |   |   |   |-- pinctrl0_pfe0
pinconfig    5  [   ]   pinconfig           |   |   |   |-- pinctrl0_pfe1_mdio
pinconfig    6  [   ]   pinconfig           |   |   |   |-- pinctrl0_pfe1
pinconfig    7  [ + ]   pinconfig           |   |   |   |-- pinctrl0_pfe2_mdio
pinconfig    8  [ + ]   pinconfig           |   |   |   `-- pinctrl0_pfe2
```

```
pinconfig     9  [ + ]   pinconfig            |  |  `-- board_pinctrl0
pinconfig    10  [   ]   pinconfig            |  |     |-- pinctrl0_i2c0
pinconfig    11  [   ]   pinconfig            |  |     |-- pinctrl0_i2c0_gpio
pinconfig    12  [   ]   pinconfig            |  |     |-- pinctrl0_i2c2
pinconfig    13  [   ]   pinconfig            |  |     |-- pinctrl0_i2c2_gpio
pinconfig    14  [ + ]   pinconfig            |  |     |-- pinctrl0_i2c4
pinconfig    15  [   ]   pinconfig            |  |     |-- pinctrl0_i2c4_gpio
pinconfig    16  [   ]   pinconfig            |  |     |-- pinctrl0_qspi
pinconfig    17  [ + ]   pinconfig            |  |     |-- pinctrl0_sd0
pinconfig    18  [   ]   pinconfig            |  |     |-- pinctrl0_dspi1
pinconfig    19  [   ]   pinconfig            |  |     `-- pinctrl0_dspi5
gpio         0   [ + ]   s32_gpio             |  `-- siul2-gpio0
simple_bus   1   [ + ]   generic_simple_bus   `-- siul2_1@44010000
pinctrl      1   [ + ]   s32_pinctrl          |-- siul2-pinctrl1
pinconfig    20  [ + ]   pinconfig            |  |-- board_generic_pinctrl1
pinconfig    21  [ + ]   pinconfig            |  |  |-- pinctrl1_pfe0_mdio
pinconfig    22  [   ]   pinconfig            |  |  |-- pinctrl1_pfe0
pinconfig    23  [   ]   pinconfig            |  |  |-- pinctrl1_pfe1_mdio
pinconfig    24  [   ]   pinconfig            |  |  |-- pinctrl1_pfe1
pinconfig    25  [ + ]   pinconfig            |  |  |-- pinctrl1_pfe2_mdio
pinconfig    26  [ + ]   pinconfig            |  |  `-- pinctrl1_pfe2
pinconfig    27  [ + ]   pinconfig            |  `-- board_pinctrl1
pinconfig    28  [   ]   pinconfig            |     |-- pinctrl1_i2c1
pinconfig    29  [   ]   pinconfig            |     |-- pinctrl1_i2c1_gpio
pinconfig    30  [   ]   pinconfig            |     |-- pinctrl1_i2c2
pinconfig    31  [   ]   pinconfig            |     |-- pinctrl1_i2c2_gpio
pinconfig    32  [ + ]   pinconfig            |     |-- pinctrl1_i2c4
pinconfig    33  [   ]   pinconfig            |     |-- pinctrl1_i2c4_gpio
pinconfig    34  [   ]   pinconfig            |     |-- pinctrl1_dspi1
pinconfig    35  [   ]   pinconfig            |     `-- pinctrl1_dspi5
gpio         1   [   ]   s32_gpio             `-- siul2-gpio1
```

3.7.3 => fdt

```
=> pri
…
fdt_addr=0x83E00000
…
=>run loadfdt
=>fdt addr ${fdt_addr}
=>fdt print
```

会打印出内核使用的 DTB 文件内容。用于确认是否加载了正确的 DTB 内容。

3.7.4 I2C 测试

I2C 口可以用于访问 PMIC，当然，直接使用 PMIC 命令也可以，如下为一个测试范列：

1: power on the board first time, uboot printed.

```
Reset cause: Power-On Reset…
```

2: put the sw2 key to hard button reset again:

```
Reset cause: External Reset…
```

3: put the sw2 key to hard button reset once again:

```
Reset cause: Power-On Reset…
```

Which will definitely to trigger POR.

And then, we use uboot i2c command to read out the follow 5510 registers by the command.

```
=> i2c dev 4
Setting bus to 4
=> i2c md 0x20 2b.1 3
002b: 21 61 af    !a.
=> i2c md 0x20 0.1 3
0000: 5c 0a 54    \.T
=> i2c md 0x20 c.1 3
000c: 00 00 a5    ...
=> i2c md 0x20 d.1 3
000d: 32 7d ab    2}.
=> i2c md 0x20 e.1 3
000e: ff 84 b5    ...
=> i2c md 0x21 0.1 3
```

0000: 11 00 70 ..p

=> i2c md 0x21 13.1 3

0013: 55 54 89 UT.

=> i2c md 0x21 11.1 3

0011: b5 65 8b .e.

=> i2c md 0x21 15.1 3

0015: 7e 00 dd ~..

=> i2c md 0x21 16.1 3

0016: 06 0f 9b ...

=> i2c md 0x21 b.1 3

000b: 50 83 57 P.W

Registers	Description	address	value
M_DEVICEID	这个从数据手册 看应该是 0x61	0x2b	21 61
PMIC_VR55XX_M_FLAG(0x20)	总标志寄存器，反映 VR5510 的总体异动，详见手册 pg91	0x0	5c 0a
PMIC_VR55XX_M_FLAG1(0x20)	标记 BUCK/LDO 是否有 OC/OT, 详见手册 pg108	0xc	00 00
PMIC_VR55XX_M_FLAG2(0x20)	标记是否有电压异常发生，详见手册 pg110	0xd	32 7d
PMIC_VR55XX_M_FLAG3(0x20)	标记 BUCK/LDO 的状态，详见手册 pg113	0xe	ff 84
PMIC_VR55XX_FS_GRL_FLAGS (0x21)	标记和 fail-safe 状态机相关的状态信息，详见手手册 pg120	0x0	11 00
PMIC_VR55XX_FS_OVUVREG_STATUS (0x21)	标记各路电压是否有 OV/UV 发生，详见手册 pg133	0x13	55 54
PMIC_VR55XX_FS_WD_SEED (0x21)	这个寄存器喂狗值，无需监控	0x11	b5 65
PMIC_VR55XX_FS_SAFE_IOS (0x21)	VR5510 安全管脚 (FS0B/RSTB/PGOOD) 是否有事件发生，详见手册 pg136	0x15	7e 00

PMIC_VR55XX_FS_DIAG_SAFETY (0x21)	和安全相关的状态信息显示，比如 FCCU/WATCHDOG 等，详见手册 pg138	0x16	06 0
PMIC_VR55XX_FS_I_FSSM (0x21)	VR5510 内部是否有错误发生，详见手册 pg128	0xb	50 83

And the follows are the

Reset cause: Functional Reset

Value:

```
=> i2c dev 4
Setting bus to 4
=> i2c md 0x20 2b.1 3
002b: 21 61 af    !a.
=> i2c md 0x20 0.1 3
0000: 5c 0a 54    \.T
=> i2c md 0x20 c.1 3
000c: 00 00 a5    ...
=> i2c md 0x20 d.1 3
000d: 32 7d ab    2}.
=> i2c md 0x20 e.1 3
000e: ff 84 b5    ...
=> i2c md 0x21 0.1 3
0000: 11 00 70    ..p
=> i2c md 0x21 13.1 3
0013: 55 54 89    UT.
=> i2c md 0x21 11.1 3
0011: b5 65 8b    .e.
=> i2c md 0x21 15.1 3
0015: 7e 00 dd    ~..
=> i2c md 0x21 b.1 3
000b: 50 84 04    P..
```

3.7.1 芯片寄存器访问

```
=>  md 4008c000 1
4008c000: 00000003
```

```
=> md 4008c010 1
4008c010: ffffffff
```

3.7.2 clk 验证

```
=> clk dump
…
SDHC                        : 400000000 Hz
DDR_PLL_MUX                 : 40000000 Hz
DDR_PLL_VCO                 : 1600000000 Hz
DDR_PLL_PHI0                : 800000000 Hz
MC_CGM5_MUX0                : 800000000 Hz
DDR                         : 800000000 Hz
..
verifclk  - Verifies clocks frequencies using CMU module
verifclk
   CMU    |     Monitored     | Reference | Expected |  Verified interval
  Address |       clock       |   clock   |  (MHz)   |       (MHz)
-----------|-------------------|-----------|----------|--------------------
0x4005c000 |       FXOSC_CLK |  FIRC_CLK |   40.000 |   38.867 -   39.257
0x4005c020 |        FIRC_CLK | FXOSC_CLK |   48.000 |             48.217
0x4005c040 |        SIRC_CLK | FXOSC_CLK |    0.032 |             0.031
0x4005c060 |    FTM_0_REF_CLK | FXOSC_CLK |   40.000 |             40.000
0x4005c080 |    FTM_1_REF_CLK | FXOSC_CLK |   40.000 |             40.000
0x4005c000 |       FXOSC_CLK |  FIRC_CLK |   40.000 |   38.867 -   39.257
0x4005c0a0 |    XBAR_DIV3_CLK |  FIRC_CLK |  133.330 |  124.375 -  125.624
0x4005c0c0 |    XBAR_CLK_M7_0 |  FIRC_CLK |  400.000 |  373.125 -  376.874
0x4005c0e0 |    XBAR_DIV3_CLK | FXOSC_CLK |  133.330 |  132.148 -  133.476
0x4005c100 |    XBAR_CLK_M7_1 |  FIRC_CLK |  400.000 |  373.125 -  376.874
0x4005c120 |    XBAR_CLK_M7_2 |  FIRC_CLK |  400.000 |  373.125 -  376.874
0x4005c140 |         PER_CLK |  FIRC_CLK |   80.000 |   77.734 -   78.515
0x4005c160 |   SERDES_REF_CLK | FXOSC_CLK |  125.000 |   99.111 -  100.107
0x4005c1a0 |      CAN_PE_CLK | FXOSC_CLK |   80.000 |   79.677 -   80.478
0x4005c1c0 |    GMAC_0_TX_CLK | FXOSC_CLK |  125.000 |  124.375 -  125.624
```

0x4005c1e0 | GMAC_TS_CLK | FXOSC_CLK | 200.000 | 198.222 - 200.214
0x4005c200 | LIN_CLK | FXOSC_CLK | 125.000 | 62.187 - 62.812
0x4005c220 | QSPI_1X_CLK | FXOSC_CLK | 200.000 | 198.222 - 200.214
0x4005c240 | SDHC_CLK | FXOSC_CLK | 400.000 | 396.445 - 400.429
0x4005c280 | DDR_CLK | FIRC_CLK | 666.659 | 746.250 - 753.749
0x4005c2a0 | GMAC_0_RX_CLK | FXOSC_CLK | 125.000 | 24.777 - 25.026
0x4005c2c0 | SPI_CLK | FXOSC_CLK | 100.000 | 99.111 - 100.107
0x4005c360 | A53_CORE_CLK | FXOSC_CLK | 1000.000 | 995.000 - 1004.999
0x4005c380 | A53_CORE_CLK | FIRC_CLK | 1000.000 | 995.000 - 1004.999
0x4005c4e0 | PFE_SYS_CLK | FXOSC_CLK | 300.000 | 299.277 - 302.285
0x4005c5c0 | PFE_MAC_0_TX_CLK | FXOSC_CLK | 312.500 | 310.937 - 314.062
0x4005c5e0 | PFE_MAC_0_RX_CLK | FXOSC_CLK | 312.500 | 310.937 - 314.062
0x4005c600 | PFE_MAC_1_TX_CLK | FXOSC_CLK | 125.000 | 0.000 - 0.015
0x4005c620 | PFE_MAC_1_RX_CLK | FXOSC_CLK | 125.000 | 47.612 - 48.090
0x4005c640 | PFE_MAC_2_TX_CLK | FXOSC_CLK | 125.000 | 124.375 - 125.624
0x4005c660 | PFE_MAC_2_RX_CLK | FXOSC_CLK | 125.000 | 24.777 - 25.026