U-Boot Hacker Manual

The U-Boot development community

Nov 23, 2020

CONTENTS

This is the top level of the U-Boot's documentation tree. U-Boot documentation, like the U-Boot itself, is very much a work in progress; that is especially true as we work to integrate our many scattered documents into a coherent whole. Please note that improvements to the documentation are welcome; join the U-Boot list at http://lists.denx.de if you want to help out.

CHAPTER

ONE

USER-ORIENTED DOCUMENTATION

The following manuals are written for *users* of the U-Boot - those who are trying to get it to work optimally on a given system.

1.1 Build U-Boot

1.1.1 Obtaining the source

The source of the U-Boot project is maintained in a Git repository.

You can download the source via

```
git clone https://gitlab.denx.de/u-boot/u-boot.git
```

A mirror of the source is maintained on Github

```
git clone https://github.com/u-boot/u-boot
```

The released versions are available as tags which use the naming scheme:

```
v<year>.<month>
```

Release candidates are named:

```
v<year>.<month>-rc<number>
```

To checkout the October 2020 release you would use:

```
git checkout v2020.10
```

1.1.2 Building with GCC

Dependencies

For building U-Boot you need a GCC compiler for your host platform. If you are not building on the target platform you further need a GCC cross compiler.

Debian based

On Debian based systems the cross compiler packages are named gcc-<architecture>-linux-gnu.

You could install GCC and the GCC cross compiler for the ARMv8 architecture with

```
sudo apt-get install gcc gcc-aarch64-linux-gnu
```

Depending on the build targets further packages maybe needed

```
sudo apt-get install bc bison build-essential coccinelle \
  device-tree-compiler dfu-util efitools flex gdisk liblz4-tool \
  libguestfs-tools libncurses-dev libpython3-dev libsdl2-dev libssl-dev \
  lzma-alone openssl python3 python3-coverage python3-pyelftools \
  python3-pytest python3-sphinxcontrib.apidoc python3-sphinx-rtd-theme swig
```

Prerequisites

For some boards you have to build prerequisite files before you can build U-Boot, e.g. for the some boards you will need to build the ARM Trusted Firmware beforehand. Please, refer to the board specific documentation *Board-specific doc* .

Configuration

Directory configs/ contains the template configuration files for the maintained boards following the naming scheme:

```
<board name>_defconfig
```

These files have been stripped of default settings. So you cannot use them directly. Instead their name serves as a make target to generate the actual configuration file .config. For instance the configuration template for the Odroid C2 board is called odroid-c2_defconfig. The corresponding .config file is generated by

```
make odroid-c2_defconfig
```

You can adjust the configuration using

```
make menuconfig
```

Building

When cross compiling you will have to specify the prefix of the cross-compiler. You can either specify the value of the CROSS_COMPILE variable on the make command line or export it beforehand.

```
CROSS_COMPILE=<compiler-prefix> make
```

Assuming cross compiling on Debian for ARMv8 this would be

```
CROSS_COMPILE=aarch64-linux-gnu- make
```

Build parameters

A list of available parameters for the make command can be obtained via

```
make help
```

You can speed up compilation by parallelization using the -j parameter, e.g.

```
CROSS_COMPILE=aarch64-linux-gnu- make -j$(nproc)
```

Further important build parameters are

- O=<dir> - generate all output files in directory <dir>, including .config
- V=1 - verbose build

Other build targets

A list of all make targets can be obtained via

```
make help
```

Important ones are

- clean - remove most generated files but keep the configuration
- mrproper - remove all generated files + config + various backup files

Installation

The process for installing U-Boot on the target device is device specific. Please, refer to the board specific documentation *Board-specific doc* .

1.1.3 Building with Clang

The biggest problem when trying to compile U-Boot with Clang is that almost all archs rely on storing gd in a global register and the Clang 3.5 user manual states: “Clang does not support global register variables; this is unlikely to be implemented soon because it requires additional LLVM backend support.”

The ARM backend can be instructed not to use the r9 and x18 registers using -ffixed-r9 or -ffixed-x18 respectively. As global registers themselves are not supported inline assembly is needed to get and set the r9 or x18 value. This leads to larger code then strictly necessary, but at least works.

NOTE: target compilation only work for _some_ ARM boards at the moment. Also AArch64 is not supported currently due to a lack of private libgcc support. Boards which reassign gd in c will also fail to compile, but there is in no strict reason to do so in the ARM world, since crt0.S takes care of this. These assignments can be avoided by changing the init calls but this is not in mainline yet.

Debian based

Required packages can be installed via apt, e.g.

```
sudo apt-get install clang
```

Note that we still use binutils for some tools so we must continue to set CROSS_COMPILE. To compile U-Boot with Clang on Linux without IAS use e.g.

```
make HOSTCC=clang rpi_2_defconfig
make HOSTCC=clang CROSS_COMPILE=arm-linux-gnueabi- \
     CC="clang -target arm-linux-gnueabi" -j8
```

It can also be used to compile sandbox:

```
make HOSTCC=clang sandbox_defconfig
make HOSTCC=clang CC=clang -j8
```

FreeBSD 11

Since llvm 3.4 is currently in the base system, the integrated assembler as is incapable of building U-Boot. Therefore gas from devel/arm-gnueabi-binutils is used instead. It needs a symlink to be picked up correctly though:

```
ln -s /usr/local/bin/arm-gnueabi-freebsd-as /usr/bin/arm-freebsd-eabi-as
```

The following commands compile U-Boot using the Clang xdev toolchain.

NOTE: CROSS_COMPILE and target differ on purpose!

```
export CROSS_COMPILE=arm-gnueabi-freebsd-
gmake rpi_2_defconfig
gmake CC="clang -target arm-freebsd-eabi --sysroot /usr/arm-freebsd" -j8
```

Given that U-Boot will default to gcc, above commands can be simplified with a simple wrapper script - saved as /usr/local/bin/arm-gnueabi-freebsd-gcc - listed below:

```
#!/bin/sh
exec clang -target arm-freebsd-eabi --sysroot /usr/arm-freebsd "$@"
```

1.1.4 Host tools

Building tools for Linux

To allow distributions to distribute all possible tools in a generic way, avoiding the need of specific tools building for each machine, a tools only defconfig file is provided.

Using this, we can build the tools by doing:

```
$ make tools-only_defconfig
$ make tools-only
```

Building tools for Windows

If you wish to generate Windows versions of the utilities in the tools directory you can use MSYS2, a software distro and building platform for Windows.

Download the MSYS2 installer from https://www.msys2.org. Make sure you have installed all required packages below in order to build these host tools:

```
* gcc (9.1.0)
* make (4.2.1)
* bison (3.4.2)
* diffutils (3.7)
* openssl-devel (1.1.1.d)
```

Note the version numbers in these parentheses above are the package versions at the time being when writing this document. The MSYS2 installer tested is http://repo.msys2.org/distrib/x86_64/msys2-x86_64-20190524.exe.

There are 3 MSYS subsystems installed: MSYS2, MinGW32 and MinGW64. Each subsystem provides an environment to build Windows applications. The MSYS2 environment is for building POSIX compliant software on Windows using an emulation layer. The MinGW32/64 subsystems are for building native Windows applications using a linux toolchain (gcc, bash, etc), targeting respectively 32 and 64 bit Windows.

Launch the MSYS2 shell of the MSYS2 environment, and do the following:

```
$ make tools-only_defconfig
$ make tools-only NO_SDL=1
```

CHAPTER

TWO

DEVELOPER-ORIENTED DOCUMENTATION

The following manuals are written for *developers* of the U-Boot - those who want to contribute to U-Boot.

2.1 Develop U-Boot

2.1.1 Coccinelle

Coccinelle is a tool for pattern matching and text transformation that has many uses in kernel development, including the application of complex, tree-wide patches and detection of problematic programming patterns.

Getting Coccinelle

The semantic patches included in the kernel use features and options which are provided by Coccinelle version 1.0.0-rc11 and above. Using earlier versions will fail as the option names used by the Coccinelle files and coccicheck have been updated.

Coccinelle is available through the package manager of many distributions, e.g. :

- Debian
- Fedora
- Ubuntu
- OpenSUSE
- Arch Linux
- NetBSD
- FreeBSD

Some distribution packages are obsolete and it is recommended to use the latest version released from the Coccinelle homepage at http://coccinelle.lip6.fr/

Or from Github at:

https://github.com/coccinelle/coccinelle

Once you have it, run the following commands:

```
./autogen
./configure
make
```

as a regular user, and install it with:

```
sudo make install
```

More detailed installation instructions to build from source can be found at:

https://github.com/coccinelle/coccinelle/blob/master/install.txt

Supplemental documentation

For supplemental documentation refer to the wiki:

https://bottest.wiki.kernel.org/coccicheck

The wiki documentation always refers to the linux-next version of the script.

For Semantic Patch Language(SmPL) grammar documentation refer to:

http://coccinelle.lip6.fr/documentation.php

Using Coccinelle on the Linux kernel

A Coccinelle-specific target is defined in the top level Makefile. This target is named `coccicheck` and calls the `coccicheck` front-end in the `scripts` directory.

Four basic modes are defined: `patch`, `report`, `context`, and `org`. The mode to use is specified by setting the MODE variable with `MODE=<mode>`.

- `patch` proposes a fix, when possible.
- `report` generates a list in the following format: file:line:column-column: message
- `context` highlights lines of interest and their context in a diff-like style.Lines of interest are indicated with `-`.
- `org` generates a report in the Org mode format of Emacs.

Note that not all semantic patches implement all modes. For easy use of Coccinelle, the default mode is "report".

Two other modes provide some common combinations of these modes.

- `chain` tries the previous modes in the order above until one succeeds.
- `rep+ctxt` runs successively the report mode and the context mode. It should be used with the C option (described later) which checks the code on a file basis.

Examples

To make a report for every semantic patch, run the following command:

```
make coccicheck MODE=report
```

To produce patches, run:

```
make coccicheck MODE=patch
```

The coccicheck target applies every semantic patch available in the sub-directories of `scripts/coccinelle` to the entire Linux kernel.

For each semantic patch, a commit message is proposed. It gives a description of the problem being checked by the semantic patch, and includes a reference to Coccinelle.

As any static code analyzer, Coccinelle produces false positives. Thus, reports must be carefully checked, and patches reviewed.

To enable verbose messages set the V= variable, for example:

```
make coccicheck MODE=report V=1
```

Coccinelle parallelization

By default, coccicheck tries to run as parallel as possible. To change the parallelism, set the J= variable. For example, to run across 4 CPUs:

```
make coccicheck MODE=report J=4
```

As of Coccinelle 1.0.2 Coccinelle uses Ocaml parmap for parallelization, if support for this is detected you will benefit from parmap parallelization.

When parmap is enabled coccicheck will enable dynamic load balancing by using `--chunksize 1` argument, this ensures we keep feeding threads with work one by one, so that we avoid the situation where most work gets done by only a few threads. With dynamic load balancing, if a thread finishes early we keep feeding it more work.

When parmap is enabled, if an error occurs in Coccinelle, this error value is propagated back, the return value of the `make coccicheck` captures this return value.

Using Coccinelle with a single semantic patch

The optional make variable COCCI can be used to check a single semantic patch. In that case, the variable must be initialized with the name of the semantic patch to apply.

For instance:

```
make coccicheck COCCI=<my_SP.cocci> MODE=patch
```

or:

```
make coccicheck COCCI=<my_SP.cocci> MODE=report
```

Controlling Which Files are Processed by Coccinelle

By default the entire kernel source tree is checked.

To apply Coccinelle to a specific directory, `M=` can be used. For example, to check drivers/net/wireless/ one may write:

```
make coccicheck M=drivers/net/wireless/
```

To apply Coccinelle on a file basis, instead of a directory basis, the following command may be used:

```
make C=1 CHECK="scripts/coccicheck"
```

To check only newly edited code, use the value 2 for the C flag, i.e.:

```
make C=2 CHECK="scripts/coccicheck"
```

In these modes, which works on a file basis, there is no information about semantic patches displayed, and no commit message proposed.

This runs every semantic patch in scripts/coccinelle by default. The COCCI variable may additionally be used to only apply a single semantic patch as shown in the previous section.

The “report” mode is the default. You can select another one with the MODE variable explained above.

Debugging Coccinelle SmPL patches

Using coccicheck is best as it provides in the spatch command line include options matching the options used when we compile the kernel. You can learn what these options are by using V=1, you could then manually run Coccinelle with debug options added.

Alternatively you can debug running Coccinelle against SmPL patches by asking for stderr to be redirected to stderr, by default stderr is redirected to /dev/null, if you'd like to capture stderr you can specify the `DEBUG_FILE="file.txt"` option to coccicheck. For instance:

```
rm -f cocci.err
make coccicheck COCCI=scripts/coccinelle/free/kfree.cocci MODE=report DEBUG_FILE=cocci.
↪err
cat cocci.err
```

You can use SPFLAGS to add debugging flags, for instance you may want to add both –profile –show-trying to SPFLAGS when debugging. For instance you may want to use:

```
rm -f err.log
export COCCI=scripts/coccinelle/misc/irqf_oneshot.cocci
make coccicheck DEBUG_FILE="err.log" MODE=report SPFLAGS="--profile --show-trying" M=./
↪drivers/mfd/arizona-irq.c
```

err.log will now have the profiling information, while stdout will provide some progress information as Coccinelle moves forward with work.

DEBUG_FILE support is only supported when using coccinelle >= 1.0.2.

.cocciconfig support

Coccinelle supports reading .cocciconfig for default Coccinelle options that should be used every time spatch is spawned, the order of precedence for variables for .cocciconfig is as follows:

- Your current user's home directory is processed first
- Your directory from which spatch is called is processed next
- The directory provided with the –dir option is processed last, if used

Since coccicheck runs through make, it naturally runs from the kernel proper dir, as such the second rule above would be implied for picking up a .cocciconfig when using `make coccicheck`.

`make coccicheck` also supports using M= targets. If you do not supply any M= target, it is assumed you want to target the entire kernel. The kernel coccicheck script has:

```
if [ "$KBUILD_EXTMOD" = "" ] ; then
    OPTIONS="--dir $srctree $COCCIINCLUDE"
else
    OPTIONS="--dir $KBUILD_EXTMOD $COCCIINCLUDE"
fi
```

KBUILD_EXTMOD is set when an explicit target with M= is used. For both cases the spatch –dir argument is used, as such third rule applies when whether M= is used or not, and when M= is used the target directory can have its own .cocciconfig file. When M= is not passed as an argument to coccicheck the target directory is the same as the directory from where spatch was called.

If not using the kernel's coccicheck target, keep the above precedence order logic of .cocciconfig reading. If using the kernel's coccicheck target, override any of the kernel's .coccicheck's settings using SPFLAGS.

We help Coccinelle when used against Linux with a set of sensible defaults options for Linux with our own Linux .cocciconfig. This hints to coccinelle git can be used for `git grep` queries over coccigrep. A timeout of 200 seconds should suffice for now.

The options picked up by coccinelle when reading a .cocciconfig do not appear as arguments to spatch processes running on your system, to confirm what options will be used by Coccinelle run:

```
spatch --print-options-only
```

You can override with your own preferred index option by using SPFLAGS. Take note that when there are conflicting options Coccinelle takes precedence for the last options passed. Using .cocciconfig is possible to use idutils, however given the order of precedence followed by Coccinelle, since the kernel now carries its own .cocciconfig, you will need to use SPFLAGS to use idutils if desired. See below section "Additional flags" for more details on how to use idutils.

Additional flags

Additional flags can be passed to spatch through the SPFLAGS variable. This works as Coccinelle respects the last flags given to it when options are in conflict.

```
make SPFLAGS=--use-glimpse coccicheck
```

Coccinelle supports idutils as well but requires coccinelle >= 1.0.6. When no ID file is specified coccinelle assumes your ID database file is in the file .id-utils.index on the top level of the kernel, coccinelle carries a script scripts/idutils_index.sh which creates the database with:

```
mkid -i C --output .id-utils.index
```

If you have another database filename you can also just symlink with this name.

```
make SPFLAGS=--use-idutils coccicheck
```

Alternatively you can specify the database filename explicitly, for instance:

```
make SPFLAGS="--use-idutils /full-path/to/ID" coccicheck
```

See `spatch --help` to learn more about spatch options.

Note that the `--use-glimpse` and `--use-idutils` options require external tools for indexing the code. None of them is thus active by default. However, by indexing the code with one of these tools, and according to the cocci file used, spatch could proceed the entire code base more quickly.

SmPL patch specific options

SmPL patches can have their own requirements for options passed to Coccinelle. SmPL patch specific options can be provided by providing them at the top of the SmPL patch, for instance:

```
// Options: --no-includes --include-headers
```

SmPL patch Coccinelle requirements

As Coccinelle features get added some more advanced SmPL patches may require newer versions of Coccinelle. If an SmPL patch requires at least a version of Coccinelle, this can be specified as follows, as an example if requiring at least Coccinelle >= 1.0.5:

```
// Requires: 1.0.5
```

Proposing new semantic patches

New semantic patches can be proposed and submitted by kernel developers. For sake of clarity, they should be organized in the sub-directories of `scripts/coccinelle/`.

Detailed description of the `report` mode

`report` generates a list in the following format:

```
file:line:column-column: message
```

Example

Running:

```
make coccicheck MODE=report COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@r depends on !context && !patch && (org || report)@
expression x;
position p;
@@

  ERR_PTR@p(PTR_ERR(x))

@script:python depends on report@
p << r.p;
x << r.x;
@@

msg="ERR_CAST can be used with %s" % (x)
coccilib.report.print_report(p[0], msg)
</smpl>
```

This SmPL excerpt generates entries on the standard output, as illustrated below:

```
/home/user/linux/crypto/ctr.c:188:9-16: ERR_CAST can be used with alg
/home/user/linux/crypto/authenc.c:619:9-16: ERR_CAST can be used with auth
/home/user/linux/crypto/xts.c:227:9-16: ERR_CAST can be used with alg
```

Detailed description of the `patch` mode

When the `patch` mode is available, it proposes a fix for each problem identified.

Example

Running:

```
make coccicheck MODE=patch COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@ depends on !context && patch && !org && !report @
expression x;
@@

- ERR_PTR(PTR_ERR(x))
```

(continues on next page)

(continued from previous page)

```
+ ERR_CAST(x)
</smpl>
```

This SmPL excerpt generates patch hunks on the standard output, as illustrated below:

```
diff -u -p a/crypto/ctr.c b/crypto/ctr.c
--- a/crypto/ctr.c 2010-05-26 10:49:38.000000000 +0200
+++ b/crypto/ctr.c 2010-06-03 23:44:49.000000000 +0200
@@ -185,7 +185,7 @@ static struct crypto_instance *crypto_ct
    alg = crypto_attr_alg(tb[1], CRYPTO_ALG_TYPE_CIPHER,
                              CRYPTO_ALG_TYPE_MASK);
    if (IS_ERR(alg))
-           return ERR_PTR(PTR_ERR(alg));
+           return ERR_CAST(alg);

    /* Block size must be >= 4 bytes. */
    err = -EINVAL;
```

Detailed description of the `context` mode

`context` highlights lines of interest and their context in a diff-like style.

> **NOTE**: The diff-like output generated is NOT an applicable patch. The intent of the `context` mode is to highlight the important lines (annotated with minus, `-`) and gives some surrounding context lines around. This output can be used with the diff mode of Emacs to review the code.

Example

Running:

```
make coccicheck MODE=context COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@ depends on context && !patch && !org && !report@
expression x;
@@

* ERR_PTR(PTR_ERR(x))
</smpl>
```

This SmPL excerpt generates diff hunks on the standard output, as illustrated below:

```
diff -u -p /home/user/linux/crypto/ctr.c /tmp/nothing
--- /home/user/linux/crypto/ctr.c   2010-05-26 10:49:38.000000000 +0200
+++ /tmp/nothing
@@ -185,7 +185,6 @@ static struct crypto_instance *crypto_ct
    alg = crypto_attr_alg(tb[1], CRYPTO_ALG_TYPE_CIPHER,
                              CRYPTO_ALG_TYPE_MASK);
    if (IS_ERR(alg))
-           return ERR_PTR(PTR_ERR(alg));

    /* Block size must be >= 4 bytes. */
    err = -EINVAL;
```

Detailed description of the org mode

org generates a report in the Org mode format of Emacs.

Example

Running:

```
make coccicheck MODE=org COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@r depends on !context && !patch && (org || report)@
expression x;
position p;
@@

  ERR_PTR@p(PTR_ERR(x))

@script:python depends on org@
p << r.p;
x << r.x;
@@

msg="ERR_CAST can be used with %s" % (x)
msg_safe=msg.replace("[","@(").replace("]",")")
coccilib.org.print_todo(p[0], msg_safe)
</smpl>
```

This SmPL excerpt generates Org entries on the standard output, as illustrated below:

```
* TODO [[view:/home/user/linux/crypto/ctr.c::face=ovl-
↪face1::linb=188::colb=9::cole=16][ERR_CAST can be used with alg]]
* TODO [[view:/home/user/linux/crypto/authenc.c::face=ovl-
↪face1::linb=619::colb=9::cole=16][ERR_CAST can be used with auth]]
* TODO [[view:/home/user/linux/crypto/xts.c::face=ovl-
↪face1::linb=227::colb=9::cole=16][ERR_CAST can be used with alg]]
```

2.1.2 Analyzing crash dumps

When the CPU detects an instruction that it cannot execute it raises an interrupt. U-Boot then writes a crash dump. This chapter describes how such dump can be analyzed.

Creating a crash dump voluntarily

For describing the analysis of a crash dump we need an example. U-Boot comes with a command 'exception' that comes in handy here. The command is enabled by:

```
CONFIG_CMD_EXCEPTION=y
```

The example output below was recorded when running qemu_arm64_defconfig on QEMU:

```
=> exception undefined
"Synchronous Abort" handler, esr 0x02000000
elr: 00000000000101fc lr : 00000000000214ec (reloc)
elr: 000000007ff291fc lr : 000000007ff3a4ec
x0 : 000000007ffbd7f8 x1 : 0000000000000000
x2 : 0000000000000001 x3 : 000000007eedce18
x4 : 000000007ff291fc x5 : 000000007eedce50
x6 : 0000000000000064 x7 : 000000007eedce10
x8 : 0000000000000000 x9 : 0000000000000004
x10: 6db6db6db6db6db7 x11: 000000000000000d
x12: 0000000000000006 x13: 000000000001869f
x14: 000000007edd7dc0 x15: 0000000000000002
x16: 000000007ff291fc x17: 0000000000000000
x18: 000000007eed8dc0 x19: 0000000000000000
x20: 000000007ffbd7f8 x21: 0000000000000000
x22: 000000007eedce10 x23: 0000000000000002
x24: 000000007ffd4c80 x25: 0000000000000000
x26: 0000000000000000 x27: 0000000000000000
x28: 000000007eedce70 x29: 000000007edd7b40

Code: b00003c0 912ad000 940029d6 17ffff52 (e7f7defb)
Resetting CPU ...

resetting ...
```

The first line provides us with the type of interrupt that occurred. On ARMv8 a synchronous abort is an exception thrown when hitting an unallocated instruction. The exception syndrome register ESR register contains information describing the reason for the exception. Bit 25 set here indicates that a 32 bit instruction led to the exception.

The second line provides the contents of the elr and the lr register after subtracting the relocation offset. - U-Boot relocates itself after being loaded. - The relocation offset can also be displayed using the bdinfo command.

After the contents of the registers we get a line indicating the machine code of the instructions preceding the crash and in parentheses the instruction leading to the dump.

Analyzing the code location

We can convert the instructions in the line starting with ‘Code:’ into mnemonics using the objdump command. To make things easier scripts/decodecode is supplied:

```
$echo 'Code: b00003c0 912ad000 940029d6 17ffff52 (e7f7defb)' | \
  CROSS_COMPILE=aarch64-linux-gnu- ARCH=arm64 scripts/decodecode
Code: b00003c0 912ad000 940029d6 17ffff52 (e7f7defb)
All code
========
   0:   b00003c0     adrp   x0, 0x79000
   4:   912ad000     add    x0, x0, #0xab4
   8:   940029d6     bl     0xa760
   c:   17ffff52     b      0xfffffffffffffd54
  10:*  e7f7defb     .inst  0xe7f7defb ; undefined <-- trapping instruction

Code starting with the faulting instruction
===========================================
   0:   e7f7defb     .inst  0xe7f7defb ; undefined
```

Now lets use the locations provided by the elr and lr registers after subtracting the relocation offset to find out where in the code the crash occurred and from where it was invoked.

File u-boot.map contains the memory layout of the U-Boot binary. Here we find these lines:

```
.text.do_undefined
               0x00000000000101fc        0xc cmd/built-in.o
.text.exception_complete
               0x0000000000010208       0x90 cmd/built-in.o
...
.text.cmd_process
               0x00000000000213b8      0x164 common/built-in.o
               0x00000000000213b8                cmd_process
.text.cmd_process_error
               0x000000000002151c       0x40 common/built-in.o
               0x000000000002151c                cmd_process_error
```

So the error occurred at the start of function do_undefined() and this function was invoked from somewhere inside function cmd_process().

If we want to dive deeper, we can disassemble the U-Boot binary:

```
$ aarch64-linux-gnu-objdump -S -D u-boot | less

00000000000101fc <do_undefined>:
{
        /*
         * 0xe7f...f.   is undefined in ARM mode
         * 0xde..       is undefined in Thumb mode
        */
        asm volatile (".word 0xe7f7defb\n");
   101fc:       e7f7defb        .inst   0xe7f7defb ; undefined
        return CMD_RET_FAILURE;
}
10200:       52800020        mov     w0, #0x1        // #1
10204:       d65f03c0        ret
```

This example is based on the ARMv8 architecture but the same procedures can be used on other architectures as well.

2.1.3 Global data

Globally required fields are held in the global data structure. A pointer to the structure is available as symbol gd. The symbol is made available by the macro %DECLARE_GLOBAL_DATA_PTR.

Register pointing to global data

On most architectures the global data pointer is stored in a register.

ARC	r25
ARM 32bit	r9
ARM 64bit	x18
M68000	d7
MicroBlaze	r31
NDS32	r10
Nios II	gp
PowerPC	r2
RISC-V	gp (x3)
SuperH	r13

The sandbox, x86, and Xtensa are notable exceptions.

Clang for ARM does not support assigning a global register. When using Clang gd is defined as an inline function using assembly code. This adds a few bytes to the code size.

Binaries called by U-Boot are not aware of the register usage and will not conserve gd. UEFI binaries call the API provided by U-Boot and may return to U-Boot. The value of gd has to be saved every time U-Boot is left and restored whenever U-Boot is reentered. This is also relevant for the implementation of function tracing. For setting the value of gd function set_gd() can be used.

Global data structure

struct **global_data**
global data structure

Definition

```
struct global_data {
  struct bd_info *bd;
  unsigned long flags;
  unsigned int baudrate;
  unsigned long cpu_clk;
  unsigned long bus_clk;
  unsigned long pci_clk;
  unsigned long mem_clk;
#if defined(CONFIG_LCD) || defined(CONFIG_VIDEO) || defined(CONFIG_DM_VIDEO);
  unsigned long fb_base;
#endif;
#if defined(CONFIG_POST);
  unsigned long post_log_word;
  unsigned long post_log_res;
  unsigned long post_init_f_time;
#endif;
#ifdef CONFIG_BOARD_TYPES;
  unsigned long board_type;
#endif;
  unsigned long have_console;
#if CONFIG_IS_ENABLED(PRE_CONSOLE_BUFFER);
  unsigned long precon_buf_idx;
#endif;
  unsigned long env_addr;
  unsigned long env_valid;
  unsigned long env_has_init;
  int env_load_prio;
  unsigned long ram_base;
  unsigned long ram_top;
```

(continues on next page)

(continued from previous page)

```
  unsigned long relocaddr;
  phys_size_t ram_size;
  unsigned long mon_len;
  unsigned long irq_sp;
  unsigned long start_addr_sp;
  unsigned long reloc_off;
  struct global_data *new_gd;
#ifdef CONFIG_DM;
  struct udevice *dm_root;
  struct udevice *dm_root_f;
  struct list_head uclass_root;
# if CONFIG_IS_ENABLED(OF_PLATDATA);
  struct driver_rt *dm_driver_rt;
# endif;
#endif;
#ifdef CONFIG_TIMER;
  struct udevice *timer;
#endif;
  const void *fdt_blob;
  void *new_fdt;
  unsigned long fdt_size;
#if CONFIG_IS_ENABLED(OF_LIVE);
  struct device_node *of_root;
#endif;
#if CONFIG_IS_ENABLED(MULTI_DTB_FIT);
  const void *multi_dtb_fit;
#endif;
  struct jt_funcs *jt;
  char env_buf[32];
#ifdef CONFIG_TRACE;
  void *trace_buff;
#endif;
#if defined(CONFIG_SYS_I2C);
  int cur_i2c_bus;
#endif;
  unsigned int timebase_h;
  unsigned int timebase_l;
#if CONFIG_VAL(SYS_MALLOC_F_LEN);
  unsigned long malloc_base;
  unsigned long malloc_limit;
  unsigned long malloc_ptr;
#endif;
#ifdef CONFIG_PCI;
  struct pci_controller *hose;
  phys_addr_t pci_ram_top;
#endif;
#ifdef CONFIG_PCI_BOOTDELAY;
  int pcidelay_done;
#endif;
  struct udevice *cur_serial_dev;
  struct arch_global_data arch;
#ifdef CONFIG_CONSOLE_RECORD;
  struct membuff console_out;
  struct membuff console_in;
#endif;
#ifdef CONFIG_DM_VIDEO;
```

(continues on next page)

(continued from previous page)

```
  ulong video_top;
  ulong video_bottom;
#endif;
#ifdef CONFIG_BOOTSTAGE;
  struct bootstage_data *bootstage;
  struct bootstage_data *new_bootstage;
#endif;
#ifdef CONFIG_LOG;
  int log_drop_count;
  int default_log_level;
  struct list_head log_head;
  int log_fmt;
  bool processing_msg;
  int logc_prev;
  int logl_prev;
#endif;
#if CONFIG_IS_ENABLED(BLOBLIST);
  struct bloblist_hdr *bloblist;
  struct bloblist_hdr *new_bloblist;
# ifdef CONFIG_SPL;
  struct spl_handoff *spl_handoff;
# endif;
#endif;
#if defined(CONFIG_TRANSLATION_OFFSET);
  fdt_addr_t translation_offset;
#endif;
#if CONFIG_IS_ENABLED(WDT);
  struct udevice *watchdog_dev;
#endif;
#ifdef CONFIG_GENERATE_ACPI_TABLE;
  struct acpi_ctx *acpi_ctx;
#endif;
};
```

Members

bd board information

flags global data flags

See *enum gd_flags*

baudrate baud rate of the serial interface

cpu_clk CPU clock rate in Hz

bus_clk platform clock rate in Hz

pci_clk PCI clock rate in Hz

mem_clk memory clock rate in Hz

fb_base base address of frame buffer memory

post_log_word active POST tests

post_log_word is a bit mask defining which POST tests are recorded (see constants POST_*).

post_log_res POST results

post_log_res is a bit mask with the POST results. A bit with value 1 indicates successful execution.

post_init_f_time time in ms when `post_init_f()` started

board_type board type

If a U-Boot configuration supports multiple board types, the actual board type may be stored in this field.

have_console console is available

A value of 1 indicates that *serial_init()* was called and a console is available. A value of 0 indicates that console input and output drivers shall not be called.

precon_buf_idx pre-console buffer index

precon_buf_idx indicates the current position of the buffer used to collect output before the console becomes available

env_addr address of environment structure

env_addr contains the address of the structure holding the environment variables.

env_valid environment is valid

See enum env_valid

env_has_init bit mask indicating environment locations

enum env_location defines which bit relates to which location

env_load_prio priority of the loaded environment

ram_base base address of RAM used by U-Boot

ram_top top address of RAM used by U-Boot

relocaddr start address of U-Boot in RAM

After relocation this field indicates the address to which U-Boot has been relocated. It can be displayed using the bdinfo command. Its value is needed to display the source code when debugging with GDB using the 'add-symbol-file u-boot <relocaddr>' command.

ram_size RAM size in bytes

mon_len monitor length in bytes

irq_sp IRQ stack pointer

start_addr_sp initial stack pointer address

reloc_off relocation offset

new_gd pointer to relocated global data

dm_root root instance for Driver Model

dm_root_f pre-relocation root instance

uclass_root head of core tree

timer timer instance for Driver Model

fdt_blob U-Boot's own device tree, NULL if none

new_fdt relocated device tree

fdt_size space reserved for relocated device space

of_root root node of the live tree

multi_dtb_fit pointer to uncompressed multi-dtb FIT image

jt jump table

The jump table contains pointers to exported functions. A pointer to the jump table is passed to standalone applications.

env_buf buffer for env_get() before reloc

trace_buff trace buffer

When tracing function in U-Boot this field points to the buffer recording the function calls.

cur_i2c_bus currently used I2C bus

timebase_h high 32 bits of timer

timebase_l low 32 bits of timer

malloc_base base address of early `malloc()`

malloc_limit limit address of early `malloc()`

malloc_ptr current address of early `malloc()`

hose PCI hose for early use

pci_ram_top top of region accessible to PCI

pcidelay_done delay time before scanning of PIC hose expired

If CONFIG_PCI_BOOTDELAY=y, `pci_hose_scan()` waits for the number of milliseconds defined by environment variable pcidelay before scanning. Once this delay has expired the flag **pcidelay_done** is set to 1.

cur_serial_dev current serial device

arch architecture-specific data

console_out output buffer for console recording

This buffer is used to collect output during console recording.

console_in input buffer for console recording

If console recording is activated, this buffer can be used to emulate input.

video_top top of video frame buffer area

video_bottom bottom of video frame buffer area

bootstage boot stage information

new_bootstage relocated boot stage information

log_drop_count number of dropped log messages

This counter is incremented for each log message which can not be processed because logging is not yet available as signaled by flag `GD_FLG_LOG_READY` in **flags**.

default_log_level default logging level

For logging devices without filters **default_log_level** defines the logging level, cf. *enum log_level_t*.

log_head list of logging devices

log_fmt bit mask for logging format

The **log_fmt** bit mask selects the fields to be shown in log messages. `enum log_fmt` defines the bits of the bit mask.

processing_msg a log message is being processed

This flag is used to suppress the creation of additional messages while another message is being processed.

logc_prev logging category of previous message

This value is used as logging category for continuation messages.

logl_prev logging level of the previous message

This value is used as logging level for continuation messages.

bloblist blob list information

new_bloblist relocated blob list information

spl_handoff SPL hand-off information

translation_offset optional translation offset

See CONFIG_TRANSLATION_OFFSET.

watchdog_dev watchdog device

acpi_ctx ACPI context pointer

gd_board_type()
retrieve board type

Parameters

Return

global board type

enum **gd_flags**
global data flags

Constants

GD_FLG_RELOC code was relocated to RAM

GD_FLG_DEVINIT devices have been initialized

GD_FLG_SILENT silent mode

GD_FLG_POSTFAIL critical POST test failed

GD_FLG_POSTSTOP POST sequence aborted

GD_FLG_LOGINIT log Buffer has been initialized

GD_FLG_DISABLE_CONSOLE disable console (in & out)

GD_FLG_ENV_READY environment imported into hash table

GD_FLG_SERIAL_READY pre-relocation serial console ready

GD_FLG_FULL_MALLOC_INIT full `malloc()` is ready

GD_FLG_SPL_INIT `spl_init()` has been called

GD_FLG_SKIP_RELOC don't relocate

GD_FLG_RECORD record console

GD_FLG_ENV_DEFAULT default variable flag

GD_FLG_SPL_EARLY_INIT early SPL initialization is done

GD_FLG_LOG_READY log system is ready for use

GD_FLG_WDT_READY watchdog is ready for use

GD_FLG_SKIP_LL_INIT don't perform low-level initialization

GD_FLG_SMP_READY SMP initialization is complete

Description

See field flags of *struct global_data*.

2.1.4 Logging in U-Boot

Introduction

U-Boot's internal operation involves many different steps and actions. From setting up the board to displaying a start-up screen to loading an Operating System, there are many component parts each with many actions.

Most of the time this internal detail is not useful. Displaying it on the console would delay booting (U-Boot's primary purpose) and confuse users.

But for digging into what is happening in a particular area, or for debugging a problem it is often useful to see what U-Boot is doing in more detail than is visible from the basic console output.

U-Boot's logging feature aims to satisfy this goal for both users and developers.

Logging levels

There are a number logging levels available.

enum **log_level_t**
Log levels supported, ranging from most to least important

Constants

LOGL_EMERG U-Boot is unstable

LOGL_ALERT Action must be taken immediately

LOGL_CRIT Critical conditions

LOGL_ERR Error that prevents something from working

LOGL_WARNING Warning may prevent optimial operation

LOGL_NOTICE Normal but significant condition, `printf()`

LOGL_INFO General information message

LOGL_DEBUG Basic debug-level message

LOGL_DEBUG_CONTENT Debug message showing full message content

LOGL_DEBUG_IO Debug message showing hardware I/O access

LOGL_COUNT Total number of valid log levels

LOGL_NONE Used to indicate that there is no valid log level

LOGL_LEVEL_MASK Mask for valid log levels

LOGL_FORCE_DEBUG Mask to force output due to LOG_DEBUG

LOGL_FIRST The first, most-important log level

LOGL_MAX The last, least-important log level

LOGL_CONT Use same log level as in previous call

Logging category

Logging can come from a wide variety of places within U-Boot. Each log message has a category which is intended to allow messages to be filtered according to their source.

enum **log_category_t**
Log categories supported.

Constants

LOGC_FIRST First log category

LOGC_NONE Default log category

LOGC_ARCH Related to arch-specific code

LOGC_BOARD Related to board-specific code

LOGC_CORE Related to core features (non-driver-model)

LOGC_DM Core driver-model

LOGC_DT Device-tree

LOGC_EFI EFI implementation

LOGC_ALLOC Memory allocation

LOGC_SANDBOX Related to the sandbox board

LOGC_BLOBLIST Bloblist

LOGC_DEVRES Device resources (`devres_...` functions)

LOGC_ACPI Advanced Configuration and Power Interface (ACPI)

LOGC_BOOT *undescribed*

LOGC_COUNT Number of log categories

LOGC_END Sentinel value for lists of log categories

LOGC_CONT Use same category as in previous call

Description

Log categories between `LOGC_FIRST` and `LOGC_NONE` correspond to uclasses (i.e. `enum uclass_id`), but there are also some more generic categories.

Remember to update log_cat_name[] after adding a new category.

Enabling logging

The following options are used to enable logging at compile time:

- CONFIG_LOG - Enables the logging system
- CONFIG_LOG_MAX_LEVEL - Max log level to build (anything higher is compiled out)
- CONFIG_LOG_CONSOLE - Enable writing log records to the console

If CONFIG_LOG is not set, then no logging will be available.

The above have SPL and TPL versions also, e.g. CONFIG_SPL_LOG_MAX_LEVEL and CONFIG_TPL_LOG_MAX_LEVEL.

Temporary logging within a single file

Sometimes it is useful to turn on logging just in one file. You can use this

```
#define LOG_DEBUG
```

to enable building in of all logging statements in a single file. Put it at the top of the file, before any #includes.

To actually get U-Boot to output this you need to also set the default logging level - e.g. set CONFIG_LOG_DEFAULT_LEVEL to 7 (`LOGL_DEBUG`) or more. Otherwise debug output is suppressed and will not be generated.

Using DEBUG

U-Boot has traditionally used a #define called DEBUG to enable debugging on a file-by-file basis. The debug() macro compiles to a printf() statement if DEBUG is enabled, and an empty statement if not.

With logging enabled, debug() statements are interpreted as logging output with a level of LOGL_DEBUG and a category of LOGC_NONE.

The logging facilities are intended to replace DEBUG, but if DEBUG is defined at the top of a file, then it takes precedence. This means that debug() statements will result in output to the console and this output will not be logged.

Logging statements

The main logging function is:

```
log(category, level, format_string, ...)
```

Also debug() and error() will generate log records - these use LOG_CATEGORY as the category, so you should #define this right at the top of the source file to ensure the category is correct.

You can also define CONFIG_LOG_ERROR_RETURN to enable the log_ret() macro. This can be used whenever your function returns an error value:

```
return log_ret(uclass_first_device(UCLASS_MMC, &dev));
```

This will write a log record when an error code is detected (a value < 0). This can make it easier to trace errors that are generated deep in the call stack.

Convenience functions

A number of convenience functions are available to shorten the code needed for logging:

- log_err(_fmt…)
- log_warning(_fmt…)
- log_notice(_fmt…)
- log_info(_fmt…)
- log_debug(_fmt…)
- log_content(_fmt…)
- log_io(_fmt…)

With these the log level is implicit in the name. The category is set by LOG_CATEGORY, which you can only define once per file, above all #includes, e.g.

```
#define LOG_CATEGORY LOGC_ALLOC
```

or

```
#define LOG_CATEGORY UCLASS_SPI
```

Remember that all uclasses IDs are log categories too.

Logging destinations

If logging information goes nowhere then it serves no purpose. U-Boot provides several possible determinations for logging information, all of which can be enabled or disabled independently:

- console - goes to stdout
- syslog - broadcast RFC 3164 messages to syslog servers on UDP port 514

The syslog driver sends the value of environmental variable 'log_hostname' as HOSTNAME if available.

Filters

Filters are attached to log drivers to control what those drivers emit. FIlters can either allow or deny a log message when they match it. Only records which are allowed by a filter make it to the driver.

Filters can be based on several criteria:

- minimum or maximum log level
- in a set of categories
- in a set of files

If no filters are attached to a driver then a default filter is used, which limits output to records with a level less than CONFIG_MAX_LOG_LEVEL.

Log command

The 'log' command provides access to several features:

- level - list log levels or set the default log level
- categories - list log categories
- drivers - list log drivers
- filter-list - list filters
- filter-add - add a new filter
- filter-remove - remove filters
- format - access the console log format
- rec - output a log record

Type 'help log' for details.

Log format

You can control the log format using the 'log format' command. The basic format is:

```
LEVEL.category,file.c:123-func() message
```

In the above, file.c:123 is the filename where the log record was generated and func() is the function name. By default ('log format default') only the message is displayed on the console. You can control which fields are present, but not the field order.

Adding Filters

To add new filters at runtime, use the ‘log filter-add’ command. For example, to suppress messages from the SPI and MMC subsystems, run:

```
log filter-add -D -c spi -c mmc
```

You will also need to add another filter to allow other messages (because the default filter no longer applies):

```
log filter-add -A -l info
```

Log levels may be either symbolic names (like above) or numbers. For example, to disable all debug and above (log level 7) messages from `drivers/core/lists.c` and `drivers/core/ofnode.c`, run:

```
log filter-add -D -f drivers/core/lists.c,drivers/core/ofnode.c -L 7
```

To view active filters, use the ‘log filter-list’ command. Some example output is:

```
=> log filter-list
num policy level            categories files
  2   deny >= DEBUG                    drivers/core/lists.c,drivers/core/ofnode.c
  0   deny <= IO                   spi
                                   mmc
  1  allow <= INFO
```

Note that filters are processed in-order from top to bottom, not in the order of their filter number. Filters are added to the top of the list if they deny when they match, and to the bottom if they allow when they match. For more information, consult the usage of the ‘log’ command, by running ‘help log’.

Code size

Code size impact depends largely on what is enabled. The following numbers are generated by ‘buildman -S’ for snow, which is a Thumb-2 board (all units in bytes):

```
This series: adds bss +20.0 data +4.0 rodata +4.0 text +44.0
CONFIG_LOG: bss -52.0 data +92.0 rodata -635.0 text +1048.0
CONFIG_LOG_MAX_LEVEL=7: bss +188.0 data +4.0 rodata +49183.0 text +98124.0
```

The last option turns every debug() statement into a logging call, which bloats the code hugely. The advantage is that it is then possible to enable all logging within U-Boot.

To Do

There are lots of useful additions that could be made. None of the below is implemented! If you do one, please add a test in test/log/log_test.c log filter-add -D -f drivers/core/lists.c,drivers/core/ofnode.c -l 6 Convenience functions to support setting the category:

- log_arch(level, format_string, …) - category LOGC_ARCH
- log_board(level, format_string, …) - category LOGC_BOARD
- log_core(level, format_string, …) - category LOGC_CORE
- log_dt(level, format_string, …) - category LOGC_DT

More logging destinations:

- device - goes to a device (e.g. serial)
- buffer - recorded in a memory buffer

Convert debug() statements in the code to log() statements

Support making printf() emit log statements at L_INFO level

Convert error() statements in the code to log() statements

Figure out what to do with BUG(), BUG_ON() and warn_non_spl()

Add a way to browse log records

Add a way to record log records for browsing using an external tool

Add commands to add and remove log devices

Allow sharing of printf format strings in log records to reduce storage size for large numbers of log records

Consider making log() calls emit an automatic newline, perhaps with a logn() function to avoid that

Passing log records through to linux (e.g. via device tree /chosen)

Provide a command to access the number of log records generated, and the number dropped due to them being generated before the log system was ready.

Add a printf() format string pragma so that log statements are checked properly

Add a command to delete existing log records.

Logging API

enum **log_level_t**
Log levels supported, ranging from most to least important

Constants

LOGL_EMERG U-Boot is unstable

LOGL_ALERT Action must be taken immediately

LOGL_CRIT Critical conditions

LOGL_ERR Error that prevents something from working

LOGL_WARNING Warning may prevent optimial operation

LOGL_NOTICE Normal but significant condition, `printf()`

LOGL_INFO General information message

LOGL_DEBUG Basic debug-level message

LOGL_DEBUG_CONTENT Debug message showing full message content

LOGL_DEBUG_IO Debug message showing hardware I/O access

LOGL_COUNT Total number of valid log levels

LOGL_NONE Used to indicate that there is no valid log level

LOGL_LEVEL_MASK Mask for valid log levels

LOGL_FORCE_DEBUG Mask to force output due to LOG_DEBUG

LOGL_FIRST The first, most-important log level

LOGL_MAX The last, least-important log level

LOGL_CONT Use same log level as in previous call

enum **log_category_t**
Log categories supported.

Constants

LOGC_FIRST First log category

LOGC_NONE Default log category

LOGC_ARCH Related to arch-specific code

LOGC_BOARD Related to board-specific code

LOGC_CORE Related to core features (non-driver-model)

LOGC_DM Core driver-model

LOGC_DT Device-tree

LOGC_EFI EFI implementation

LOGC_ALLOC Memory allocation

LOGC_SANDBOX Related to the sandbox board

LOGC_BLOBLIST Bloblist

LOGC_DEVRES Device resources (`devres_...` functions)

LOGC_ACPI Advanced Configuration and Power Interface (ACPI)

LOGC_BOOT *undescribed*

LOGC_COUNT Number of log categories

LOGC_END Sentinel value for lists of log categories

LOGC_CONT Use same category as in previous call

Description

Log categories between `LOGC_FIRST` and `LOGC_NONE` correspond to uclasses (i.e. `enum uclass_id`), but there are also some more generic categories.

Remember to update log_cat_name[] after adding a new category.

int **_log**(enum *log_category_t cat*, enum *log_level_t level*, const char * *file*, int *line*, const char * *func*, const char * *fmt*, ...)

Internal function to emit a new log record

Parameters

enum log_category_t cat Category of log record (indicating which subsystem generated it)

enum log_level_t level Level of log record (indicating its severity)

const char * file File name of file where log record was generated

int line Line number in file where log record was generated

const char * func Function where log record was generated

const char * fmt `printf()` format string for log record

... Optional parameters, according to the format string **fmt**

Return

0 if log record was emitted, -ve on error

assert(*x*)

assert expression is true

Parameters

x expression to test

Description

If the expression x evaluates to false and _DEBUG evaluates to true, a panic message is written and the system stalls. The value of _DEBUG is set to true if DEBUG is defined before including common.h.

The expression x is always executed irrespective of the value of _DEBUG.

struct **log_rec**
a single log record

Definition

```
struct log_rec {
  enum log_category_t cat;
  enum log_level_t level;
  bool force_debug;
  const char *file;
  int line;
  const char *func;
  const char *msg;
};
```

Members

cat Category, representing a uclass or part of U-Boot

level Severity level, less severe is higher

force_debug Force output of debug

file Name of file where the log record was generated (not allocated)

line Line number where the log record was generated

func Function where the log record was generated (not allocated)

msg Log message (allocated)

Description

Holds information about a single record in the log

Members marked as 'not allocated' are stored as pointers and the caller is responsible for making sure that the data pointed to is not overwritten. Memebers marked as 'allocated' are allocated (e.g. via `strdup()`) by the log system.

TODO(sjg**chromium.org**): Compress this struct down a bit to reduce space, e.g. a single u32 for cat, level, line and force_debug

struct **log_driver**
a driver which accepts and processes log records

Definition

```
struct log_driver {
  const char *name;
  int (*emit)(struct log_device *ldev, struct log_rec *rec);
  unsigned short flags;
};
```

Members

name Name of driver

emit emit a log record

Called by the log system to pass a log record to a particular driver for processing. The filter is checked before calling this function.

flags Initial value for flags (use LOGDF_ENABLE to enable on start-up)

struct **log_device**
an instance of a log driver

Definition

```
struct log_device {
  unsigned short next_filter_num;
  unsigned short flags;
  struct log_driver *drv;
  struct list_head filter_head;
  struct list_head sibling_node;
};
```

Members

next_filter_num Seqence number of next filter filter added (0=no filters yet). This increments with each new filter on the device, but never decrements

flags Flags for this filter (enum log_device_flags)

drv Pointer to driver for this device

filter_head List of filters for this device

sibling_node Next device in the list of all devices

Description

Since drivers are set up at build-time we need to have a separate device for the run-time aspects of drivers (currently just a list of filters to apply to records send to this device).

enum **log_filter_flags**
Flags which modify a filter

Constants

LOGFF_HAS_CAT Filter has a category list

LOGFF_DENY Filter denies matching messages

LOGFF_LEVEL_MIN Filter's level is a minimum, not a maximum

struct **log_filter**
criterial to filter out log messages

Definition

```
struct log_filter {
  int filter_num;
  int flags;
  enum log_category_t cat_list[LOGF_MAX_CATEGORIES];
  enum log_level_t level;
  const char *file_list;
  struct list_head sibling_node;
};
```

Members

filter_num Sequence number of this filter. This is returned when adding a new filter, and must be provided when removing a previously added filter.

flags Flags for this filter (`LOGFF_...`)

cat_list List of categories to allow (terminated by `LOGC_END`). If empty then all categories are permitted. Up to `LOGF_MAX_CATEGORIES` entries can be provided

level Maximum (or minimum, if `LOGFF_MIN_LEVEL`) log level to allow

file_list List of files to allow, separated by comma. If NULL then all files are permitted

sibling_node Next filter in the list of filters for this log device

Description

If a message matches all criteria, then it is allowed. If LOGFF_DENY is set, then it is denied instead.

const char * **log_get_cat_name**(enum *log_category_t cat*)
    Get the name of a category

Parameters

enum log_category_t cat Category to look up

Return

category name (which may be a uclass driver name) if found, or "<invalid>" if invalid, or "<missing>" if not found. All error responses begin with '<'.

enum *log_category_t* **log_get_cat_by_name**(const char * *name*)
    Look up a category by name

Parameters

const char * name Name to look up

Return

Category, or `LOGC_NONE` if not found

const char * **log_get_level_name**(enum *log_level_t level*)
    Get the name of a log level

Parameters

enum log_level_t level Log level to look up

Return

Log level name (in ALL CAPS)

enum *log_level_t* **log_get_level_by_name**(const char * *name*)
    Look up a log level by name

Parameters

const char * name Name to look up

Return

Log level, or `LOGL_NONE` if not found

struct *log_device* * **log_device_find_by_name**(const char * *drv_name*)
    Look up a log device by its driver's name

Parameters

const char * drv_name Name of the driver

Return

the log device, or `NULL` if not found

bool **log_has_cat**(enum *log_category_t cat_list*, enum *log_category_t cat*)
    check if a log category exists within a list

Parameters

enum log_category_t cat_list List of categories to check, at most `LOGF_MAX_CATEGORIES` entries long, terminated by `LC_END` if fewer

enum log_category_t cat Category to search for

Return

`true` if **cat** is in **cat_list**, else `false`

bool **log_has_file**(const char * *file_list*, const char * *file*)
check if a file is with a list

Parameters

const char * file_list List of files to check, separated by comma

const char * file File to check for. This string is matched against the end of each file in the list, i.e. ignoring any preceding path. The list is intended to consist of relative pathnames, e.g. common/main.c,cmd/log.c

Return

`true` if **file** is in **file_list**, else `false`

int **log_add_filter_flags**(const char * *drv_name*, enum *log_category_t cat_list*, enum *log_level_t level*, const char * *file_list*, int *flags*)
Add a new filter to a log device, specifying flags

Parameters

const char * drv_name Driver name to add the filter to (since each driver only has a single device)

enum log_category_t cat_list List of categories to allow (terminated by `LOGC_END`). If empty then all categories are permitted. Up to `LOGF_MAX_CATEGORIES` entries can be provided

enum log_level_t level Maximum (or minimum, if `LOGFF_LEVEL_MIN`) log level to allow

const char * file_list List of files to allow, separated by comma. If NULL then all files are permitted

int flags Flags for this filter (`LOGFF_...`)

Return

the sequence number of the new filter (>=0) if the filter was added, or a -ve value on error

int **log_add_filter**(const char * *drv_name*, enum *log_category_t cat_list*, enum *log_level_t max_level*, const char * *file_list*)
Add a new filter to a log device

Parameters

const char * drv_name Driver name to add the filter to (since each driver only has a single device)

enum log_category_t cat_list List of categories to allow (terminated by `LOGC_END`). If empty then all categories are permitted. Up to `LOGF_MAX_CATEGORIES` entries can be provided

enum log_level_t max_level Maximum log level to allow

const char * file_list List of files to allow, separated by comma. If `NULL` then all files are permitted

Return

the sequence number of the new filter (>=0) if the filter was added, or a -ve value on error

int **log_remove_filter**(const char * *drv_name*, int *filter_num*)
Remove a filter from a log device

Parameters

const char * drv_name Driver name to remove the filter from (since each driver only has a single device)

int filter_num Filter number to remove (as returned by *log_add_filter()*)

Return

0 if the filter was removed, -`ENOENT` if either the driver or the filter number was not found

int **log_device_set_enable**(struct *log_driver* * *drv*, bool *enable*)
Enable or disable a log device

Parameters

struct log_driver * drv Driver of device to enable

bool enable true to enable, false to disable **return** 0 if OK, -ENOENT if the driver was not found

Description

Devices are referenced by their driver, so use LOG_GET_DRIVER(name) to pass the driver to this function. For example if the driver is declared with LOG_DRIVER(wibble) then pass LOG_GET_DRIVER(wibble) here.

int **log_init**(void)
Set up the log system ready for use

Parameters

void no arguments

Return

0 if OK, -ENOMEM if out of memory

int **log_get_default_format**(void)
get default log format

Parameters

void no arguments

Description

The default log format is configurable via CONFIG_LOGF_FILE, CONFIG_LOGF_LINE, and CONFIG_LOGF_FUNC.

Return

default log format

CHAPTER

THREE

UNIFIED EXTENSIBLE FIRMWARE (UEFI)

U-Boot provides an implementation of the UEFI API allowing to run UEFI compliant software like Linux, GRUB, and iPXE. Furthermore U-Boot itself can be run an UEFI payload.

3.1 Unified Extensible Firmware (UEFI)

3.1.1 UEFI on U-Boot

The Unified Extensible Firmware Interface Specification (UEFI) [1] has become the default for booting on AArch64 and x86 systems. It provides a stable API for the interaction of drivers and applications with the firmware. The API comprises access to block storage, network, and console to name a few. The Linux kernel and boot loaders like GRUB or the FreeBSD loader can be executed.

Development target

The implementation of UEFI in U-Boot strives to reach the requirements described in the “Embedded Base Boot Requirements (EBBR) Specification - Release v1.0” [2]. The “Server Base Boot Requirements System Software on ARM Platforms” [3] describes a superset of the EBBR specification and may be used as further reference.

A full blown UEFI implementation would contradict the U-Boot design principle “keep it small”.

Building U-Boot for UEFI

The UEFI standard supports only little-endian systems. The UEFI support can be activated for ARM and x86 by specifying:

```
CONFIG_CMD_BOOTEFI=y
CONFIG_EFI_LOADER=y
```

in the .config file.

Support for attaching virtual block devices, e.g. iSCSI drives connected by the loaded UEFI application [4], requires:

```
CONFIG_BLK=y
CONFIG_PARTITIONS=y
```

Executing a UEFI binary

The bootefi command is used to start UEFI applications or to install UEFI drivers. It takes two parameters:

```
bootefi <image address> [fdt address]
```

- image address - the memory address of the UEFI binary
- fdt address - the memory address of the flattened device tree

Below you find the output of an example session starting GRUB:

```
=> load mmc 0:2 ${fdt_addr_r} boot/dtb
29830 bytes read in 14 ms (2 MiB/s)
=> load mmc 0:1 ${kernel_addr_r} efi/debian/grubaa64.efi
reading efi/debian/grubaa64.efi
120832 bytes read in 7 ms (16.5 MiB/s)
=> bootefi ${kernel_addr_r} ${fdt_addr_r}
```

The bootefi command uses the device, the file name, and the file size (environment variable 'filesize') of the most recently loaded file when setting up the binary for execution. So the UEFI binary should be loaded last.

The environment variable 'bootargs' is passed as load options in the UEFI system table. The Linux kernel EFI stub uses the load options as command line arguments.

Launching a UEFI binary from a FIT image

A signed FIT image can be used to securely boot a UEFI image via the bootm command. This feature is available if U-Boot is configured with:

```
CONFIG_BOOTM_EFI=y
```

A sample configuration is provided as file doc/uImage.FIT/uefi.its.

Below you find the output of an example session starting GRUB:

```
=> load mmc 0:1 ${kernel_addr_r} image.fit
4620426 bytes read in 83 ms (53.1 MiB/s)
=> bootm ${kernel_addr_r}#config-grub-nofdt
## Loading kernel from FIT Image at 40400000 ...
   Using 'config-grub-nofdt' configuration
   Verifying Hash Integrity ... sha256,rsa2048:dev+ OK
   Trying 'efi-grub' kernel subimage
     Description:  GRUB EFI Firmware
     Created:      2019-11-20   8:18:16 UTC
     Type:         Kernel Image (no loading done)
     Compression:  uncompressed
     Data Start:   0x404000d0
     Data Size:    450560 Bytes = 440 KiB
     Hash algo:    sha256
     Hash value:   4dbee00021112df618f58b3f7cf5e1595533d543094064b9ce991e8b054a9eec
   Verifying Hash Integrity ... sha256+ OK
   XIP Kernel Image (no loading done)
## Transferring control to EFI (at address 404000d0) ...
Welcome to GRUB!
```

See doc/uImage.FIT/howto.txt for an introduction to FIT images.

Configuring UEFI secure boot

The UEFI specification[1] defines a secure way of executing UEFI images by verifying a signature (or message digest) of image with certificates. This feature on U-Boot is enabled with:

```
CONFIG_UEFI_SECURE_BOOT=y
```

To make the boot sequence safe, you need to establish a chain of trust; In UEFI secure boot the chain trust is defined by the following UEFI variables

- PK - Platform Key
- KEK - Key Exchange Keys
- db - white list database
- dbx - black list database

An in depth description of UEFI secure boot is beyond the scope of this document. Please, refer to the UEFI specification and available online documentation. Here is a simple example that you can follow for your initial attempt (Please note that the actual steps will depend on your system and environment.):

Install the required tools on your host

- openssl
- efitools
- sbsigntool

Create signing keys and the key database on your host:

The platform key

```
openssl req -x509 -sha256 -newkey rsa:2048 -subj /CN=TEST_PK/ \
        -keyout PK.key -out PK.crt -nodes -days 365
cert-to-efi-sig-list -g 11111111-2222-3333-4444-123456789abc \
        PK.crt PK.esl;
sign-efi-sig-list -c PK.crt -k PK.key PK PK.esl PK.auth
```

The key exchange keys

```
openssl req -x509 -sha256 -newkey rsa:2048 -subj /CN=TEST_KEK/ \
        -keyout KEK.key -out KEK.crt -nodes -days 365
cert-to-efi-sig-list -g 11111111-2222-3333-4444-123456789abc \
        KEK.crt KEK.esl
sign-efi-sig-list -c PK.crt -k PK.key KEK KEK.esl KEK.auth
```

The whitelist database

```
$ openssl req -x509 -sha256 -newkey rsa:2048 -subj /CN=TEST_db/ \
        -keyout db.key -out db.crt -nodes -days 365
$ cert-to-efi-sig-list -g 11111111-2222-3333-4444-123456789abc \
        db.crt db.esl
$ sign-efi-sig-list -c KEK.crt -k KEK.key db db.esl db.auth
```

Copy the *.auth files to media, say mmc, that is accessible from U-Boot.

Sign an image with one of the keys in “db” on your host

```
sbsign --key db.key --cert db.crt helloworld.efi
```

Now in U-Boot install the keys on your board:

```
fatload mmc 0:1 <tmpaddr> PK.auth
setenv -e -nv -bs -rt -at -i <tmpaddr>:$filesize PK
fatload mmc 0:1 <tmpaddr> KEK.auth
setenv -e -nv -bs -rt -at -i <tmpaddr>:$filesize KEK
fatload mmc 0:1 <tmpaddr> db.auth
setenv -e -nv -bs -rt -at -i <tmpaddr>:$filesize db
```

Set up boot parameters on your board:

```
efidebug boot add 1 HELLO mmc 0:1 /helloworld.efi.signed ""
```

Now your board can run the signed image via the boot manager (see below). You can also try this sequence by running Pytest, test_efi_secboot, on the sandbox

```
cd <U-Boot source directory>
pytest.py test/py/tests/test_efi_secboot/test_signed.py --bd sandbox
```

UEFI binaries may be signed by Microsoft using the following certificates:

- KEK: Microsoft Corporation KEK CA 2011 http://go.microsoft.com/fwlink/?LinkId=321185.
- db: Microsoft Windows Production PCA 2011 http://go.microsoft.com/fwlink/p/?linkid=321192.
- db: Microsoft Corporation UEFI CA 2011 http://go.microsoft.com/fwlink/p/?linkid=321194.

Using OP-TEE for EFI variables

Instead of implementing UEFI variable services inside U-Boot they can also be provided in the secure world by a module for OP-TEE[1]. The interface between U-Boot and OP-TEE for variable services is enabled by CONFIG_EFI_MM_COMM_TEE=y.

Tianocore EDK II's standalone management mode driver for variables can be linked to OP-TEE for this purpose. This module uses the Replay Protected Memory Block (RPMB) of an eMMC device for persisting non-volatile variables. When calling the variable services via the OP-TEE API U-Boot's OP-TEE supplicant relays calls to the RPMB driver which has to be enabled via CONFIG_SUPPORT_EMMC_RPMB=y.

[1] https://optee.readthedocs.io/ - OP-TEE documentation

Executing the boot manager

The UEFI specification foresees to define boot entries and boot sequence via UEFI variables. Booting according to these variables is possible via:

```
bootefi bootmgr [fdt address]
```

As of U-Boot v2020.10 UEFI variables cannot be set at runtime. The U-Boot command 'efidebug' can be used to set the variables.

Executing the built in hello world application

A hello world UEFI application can be built with:

```
CONFIG_CMD_BOOTEFI_HELLO_COMPILE=y
```

It can be embedded into the U-Boot binary with:

```
CONFIG_CMD_BOOTEFI_HELLO=y
```

The bootefi command is used to start the embedded hello world application:

```
bootefi hello [fdt address]
```

Below you find the output of an example session:

```
=> bootefi hello ${fdtcontroladdr}
## Starting EFI application at 01000000 ...
WARNING: using memory device/image path, this may confuse some payloads!
Hello, world!
Running on UEFI 2.7
Have SMBIOS table
Have device tree
Load options: root=/dev/sdb3 init=/sbin/init rootwait ro
## Application terminated, r = 0
```

The environment variable fdtcontroladdr points to U-Boot's internal device tree (if available).

Executing the built-in self-test

An UEFI self-test suite can be embedded in U-Boot by building with:

```
CONFIG_CMD_BOOTEFI_SELFTEST=y
```

For testing the UEFI implementation the bootefi command can be used to start the self-test:

```
bootefi selftest [fdt address]
```

The environment variable 'efi_selftest' can be used to select a single test. If it is not provided all tests are executed except those marked as 'on request'. If the environment variable is set to 'list' a list of all tests is shown.

Below you can find the output of an example session:

```
=> setenv efi_selftest simple network protocol
=> bootefi selftest
Testing EFI API implementation
Selected test: 'simple network protocol'
Setting up 'simple network protocol'
Setting up 'simple network protocol' succeeded
Executing 'simple network protocol'
DHCP Discover
DHCP reply received from 192.168.76.2 (52:55:c0:a8:4c:02)
  as broadcast message.
Executing 'simple network protocol' succeeded
Tearing down 'simple network protocol'
Tearing down 'simple network protocol' succeeded
Boot services terminated
Summary: 0 failures
Preparing for reset. Press any key.
```

The UEFI life cycle

After the U-Boot platform has been initialized the UEFI API provides two kinds of services:

- boot services
- runtime services

The API can be extended by loading UEFI drivers which come in two variants:

- boot drivers
- runtime drivers

UEFI drivers are installed with U-Boot's bootefi command. With the same command UEFI applications can be executed.

Loaded images of UEFI drivers stay in memory after returning to U-Boot while loaded images of applications are removed from memory.

An UEFI application (e.g. an operating system) that wants to take full control of the system calls ExitBootServices. After a UEFI application calls ExitBootServices

- boot services are not available anymore
- timer events are stopped
- the memory used by U-Boot except for runtime services is released
- the memory used by boot time drivers is released

So this is a point of no return. Afterwards the UEFI application can only return to U-Boot by rebooting.

The UEFI object model

UEFI offers a flexible and expandable object model. The objects in the UEFI API are devices, drivers, and loaded images. These objects are referenced by handles.

The interfaces implemented by the objects are referred to as protocols. These are identified by GUIDs. They can be installed and uninstalled by calling the appropriate boot services.

Handles are created by the InstallProtocolInterface or the InstallMultipleProtocolinterfaces service if NULL is passed as handle.

Handles are deleted when the last protocol has been removed with the UninstallProtocolInterface or the UninstallMultipleProtocolInterfaces service.

Devices offer the EFI_DEVICE_PATH_PROTOCOL. A device path is the concatenation of device nodes. By their device paths all devices of a system are arranged in a tree.

Drivers offer the EFI_DRIVER_BINDING_PROTOCOL. This protocol is used to connect a driver to devices (which are referenced as controllers in this context).

Loaded images offer the EFI_LOADED_IMAGE_PROTOCOL. This protocol provides meta information about the image and a pointer to the unload callback function.

The UEFI events

In the UEFI terminology an event is a data object referencing a notification function which is queued for calling when the event is signaled. The following types of events exist:

- periodic and single shot timer events
- exit boot services events, triggered by calling the ExitBootServices() service
- virtual address change events
- memory map change events
- read to boot events
- reset system events
- system table events
- events that are only triggered programmatically

Events can be created with the CreateEvent service and deleted with CloseEvent service.

Events can be assigned to an event group. If any of the events in a group is signaled, all other events in the group are also set to the signaled state.

The UEFI driver model

A driver is specific for a single protocol installed on a device. To install a driver on a device the ConnectController service is called. In this context controller refers to the device for which the driver is installed.

The relevant drivers are identified using the EFI_DRIVER_BINDING_PROTOCOL. This protocol has has three functions:

- supported - determines if the driver is compatible with the device
- start - installs the driver by opening the relevant protocol with attribute EFI_OPEN_PROTOCOL_BY_DRIVER
- stop - uninstalls the driver

The driver may create child controllers (child devices). E.g. a driver for block IO devices will create the device handles for the partitions. The child controllers will open the supported protocol with the attribute EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER.

A driver can be detached from a device using the DisconnectController service.

U-Boot devices mapped as UEFI devices

Some of the U-Boot devices are mapped as UEFI devices

- block IO devices
- console
- graphical output
- network adapter

As of U-Boot 2018.03 the logic for doing this is hard coded.

The development target is to integrate the setup of these UEFI devices with the U-Boot driver model [5]. So when a U-Boot device is discovered a handle should be created and the device path protocol and the relevant IO protocol should be installed. The UEFI driver then would be attached by calling ConnectController. When a U-Boot device is removed DisconnectController should be called.

UEFI devices mapped as U-Boot devices

UEFI drivers binaries and applications may create new (virtual) devices, install a protocol and call the ConnectController service. Now the matching UEFI driver is determined by iterating over the implementations of the EFI_DRIVER_BINDING_PROTOCOL.

It is the task of the UEFI driver to create a corresponding U-Boot device and to proxy calls for this U-Boot device to the controller.

In U-Boot 2018.03 this has only been implemented for block IO devices.

UEFI uclass

An UEFI uclass driver (lib/efi_driver/efi_uclass.c) has been created that takes care of initializing the UEFI drivers and providing the EFI_DRIVER_BINDING_PROTOCOL implementation for the UEFI drivers.

A linker created list is used to keep track of the UEFI drivers. To create an entry in the list the UEFI driver uses the U_BOOT_DRIVER macro specifying UCLASS_EFI as the ID of its uclass, e.g:

```
/* Identify as UEFI driver */
U_BOOT_DRIVER(efi_block) = {
    .name  = "EFI block driver",
    .id    = UCLASS_EFI,
    .ops   = &driver_ops,
};
```

The available operations are defined via the structure struct efi_driver_ops:

```
struct efi_driver_ops {
    const efi_guid_t *protocol;
    const efi_guid_t *child_protocol;
    int (*bind)(efi_handle_t handle, void *interface);
};
```

When the supported() function of the EFI_DRIVER_BINDING_PROTOCOL is called the uclass checks if the protocol GUID matches the protocol GUID of the UEFI driver. In the start() function the bind() function of the UEFI driver is called after checking the GUID. The stop() function of the EFI_DRIVER_BINDING_PROTOCOL disconnects the child controllers created by the UEFI driver and the UEFI driver. (In U-Boot v2013.03 this is not yet completely implemented.)

UEFI block IO driver

The UEFI block IO driver supports devices exposing the EFI_BLOCK_IO_PROTOCOL.

When connected it creates a new U-Boot block IO device with interface type IF_TYPE_EFI, adds child controllers mapping the partitions, and installs the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL on these. This can be used together with the software iPXE to boot from iSCSI network drives [4].

This driver is only available if U-Boot is configured with:

```
CONFIG_BLK=y
CONFIG_PARTITIONS=y
```

Miscellaneous

Load file 2 protocol

The load file 2 protocol can be used by the Linux kernel to load the initial RAM disk. U-Boot can be configured to provide an implementation with:

```
EFI_LOAD_FILE2_INITRD=y
EFI_INITRD_FILESPEC=interface dev:part path_to_initrd
```

Links

- [1] http://uefi.org/specifications - UEFI specifications
- [2] https://github.com/ARM-software/ebbr/releases/download/v1.0/ebbr-v1.0.pdf - Embedded Base Boot Requirements (EBBR) Specification - Release v1.0
- [3] https://developer.arm.com/docs/den0044/latest/server-base-boot-requirements-system-software-on-ar - Server Base Boot Requirements System Software on ARM Platforms - Version 1.1
- [4] *iSCSI booting with U-Boot and iPXE*
- [5] *Driver Model*

3.1.2 U-Boot on EFI

This document provides information about U-Boot running on top of EFI, either as an application or just as a means of getting U-Boot onto a new platform.

Motivation

Running U-Boot on EFI is useful in several situations:

- You have EFI running on a board but U-Boot does not natively support it fully yet. You can boot into U-Boot from EFI and use that until U-Boot is fully ported
- You need to use an EFI implementation (e.g. UEFI) because your vendor requires it in order to provide support
- You plan to use coreboot to boot into U-Boot but coreboot support does not currently exist for your platform. In the meantime you can use U-Boot on EFI and then move to U-Boot on coreboot when ready
- You use EFI but want to experiment with a simpler alternative like U-Boot

Status

Only x86 is supported at present. If you are using EFI on another architecture you may want to reconsider. However, much of the code is generic so could be ported.

U-Boot supports running as an EFI application for 32-bit EFI only. This is not very useful since only a serial port is provided. You can look around at memory and type 'help' but that is about it.

More usefully, U-Boot supports building itself as a payload for either 32-bit or 64-bit EFI. U-Boot is packaged up and loaded in its entirety by EFI. Once started, U-Boot changes to 32-bit mode (currently) and takes over the machine. You can use devices, boot a kernel, etc.

Build Instructions

First choose a board that has EFI support and obtain an EFI implementation for that board. It will be either 32-bit or 64-bit. Alternatively, you can opt for using QEMU [1] and the OVMF [2], as detailed below.

To build U-Boot as an EFI application (32-bit EFI required), enable CONFIG_EFI and CONFIG_EFI_APP. The efi-x86_app config (efi-x86_app_defconfig) is set up for this. Just build U-Boot as normal, e.g.:

```
make efi-x86_app_defconfig
make
```

To build U-Boot as an EFI payload (32-bit or 64-bit EFI can be used), enable CONFIG_EFI, CONFIG_EFI_STUB, and select either CONFIG_EFI_STUB_32BIT or CONFIG_EFI_STUB_64BIT. The efi-x86_payload configs (efi-x86_payload32_defconfig and efi-x86_payload32_defconfig) are set up for this. Then build U-Boot as normal, e.g.:

```
make efi-x86_payload32_defconfig (or efi-x86_payload64_defconfig)
make
```

You will end up with one of these files depending on what you build for:

- u-boot-app.efi - U-Boot EFI application
- u-boot-payload.efi - U-Boot EFI payload application

Trying it out

QEMU is an emulator and it can emulate an x86 machine. Please make sure your QEMU version is 2.3.0 or above to test this. You can run the payload with something like this:

```
mkdir /tmp/efi
cp /path/to/u-boot*.efi /tmp/efi
qemu-system-x86_64 -bios bios.bin -hda fat:/tmp/efi/
```

Add -nographic if you want to use the terminal for output. Once it starts type 'fs0:u-boot-payload.efi' to run the payload or 'fs0:u-boot-app.efi' to run the application. 'bios.bin' is the EFI 'BIOS'. Check [2] to obtain a prebuilt EFI BIOS for QEMU or you can build one from source as well.

To try it on real hardware, put u-boot-app.efi on a suitable boot medium, such as a USB stick. Then you can type something like this to start it:

```
fs0:u-boot-payload.efi
```

(or fs0:u-boot-app.efi for the application)

This will start the payload, copy U-Boot into RAM and start U-Boot. Note that EFI does not support booting a 64-bit application from a 32-bit EFI (or vice versa). Also it will often fail to print an error message if you get this wrong.

Inner workings

Here follow a few implementation notes for those who want to fiddle with this and perhaps contribute patches.

The application and payload approaches sound similar but are in fact implemented completely differently.

EFI Application

For the application the whole of U-Boot is built as a shared library. The efi_main() function is in lib/efi/efi_app.c. It sets up some basic EFI functions with efi_init(), sets up U-Boot global_data, allocates memory for U-Boot's malloc(), etc. and enters the normal init sequence (board_init_f() and board_init_r()).

Since U-Boot limits its memory access to the allocated regions very little special code is needed. The CONFIG_EFI_APP option controls a few things that need to change so 'git grep CONFIG_EFI_APP' may be instructive. The CONFIG_EFI option controls more general EFI adjustments.

The only available driver is the serial driver. This calls back into EFI 'boot services' to send and receive characters. Although it is implemented as a serial driver the console device is not necessarilly serial. If you boot EFI with video output then the 'serial' device will operate on your target devices's display instead and the device's USB keyboard will also work if connected. If you have both serial and video output, then both consoles will be active. Even though U-Boot does the same thing normally, These are features of EFI, not U-Boot.

Very little code is involved in implementing the EFI application feature. U-Boot is highly portable. Most of the difficulty is in modifying the Makefile settings to pass the right build flags. In particular there is very little x86-specific code involved - you can find most of it in arch/x86/cpu. Porting to ARM (which can also use EFI if you are brave enough) should be straightforward.

Use the 'reset' command to get back to EFI.

EFI Payload

The payload approach is a different kettle of fish. It works by building U-Boot exactly as normal for your target board, then adding the entire image (including device tree) into a small EFI stub application responsible for booting it. The stub application is built as a normal EFI application except that it has a lot of data attached to it.

The stub application is implemented in lib/efi/efi_stub.c. The efi_main() function is called by EFI. It is responsible for copying U-Boot from its original location into memory, disabling EFI boot services and starting U-Boot. U-Boot then starts as normal, relocates, starts all drivers, etc.

The stub application is architecture-dependent. At present it has some x86-specific code and a comment at the top of efi_stub.c describes this.

While the stub application does allocate some memory from EFI this is not used by U-Boot (the payload). In fact when U-Boot starts it has all of the memory available to it and can operate as it pleases (but see the next section).

Tables

The payload can pass information to U-Boot in the form of EFI tables. At present this feature is used to pass the EFI memory map, an inordinately large list of memory regions. You can use the 'efi mem all' command to display this list. U-Boot uses the list to work out where to relocate itself.

Although U-Boot can use any memory it likes, EFI marks some memory as used by 'run-time services', code that hangs around while U-Boot is running and is even present when Linux is running. This is common on x86 and provides a way for Linux to call back into the firmware to control things like CPU fan speed. U-Boot uses only 'conventional' memory, in EFI terminology. It will relocate itself to the top of the largest block of memory it can find below 4GB.

Interrupts

U-Boot drivers typically don't use interrupts. Since EFI enables interrupts it is possible that an interrupt will fire that U-Boot cannot handle. This seems to cause problems. For this reason the U-Boot payload runs with interrupts disabled at present.

32/64-bit

While the EFI application can in principle be built as either 32- or 64-bit, only 32-bit is currently supported. This means that the application can only be used with 32-bit EFI.

The payload stub can be build as either 32- or 64-bits. Only a small amount of code is built this way (see the extra- line in lib/efi/Makefile). Everything else is built as a normal U-Boot, so is always 32-bit on x86 at present.

Future work

This work could be extended in a number of ways:

- Add ARM support
- Add 64-bit application support
- Figure out how to solve the interrupt problem
- Add more drivers to the application side (e.g. video, block devices, USB, environment access). This would mostly be an academic exercise as a strong use case is not readily apparent, but it might be fun.
- Avoid turning off boot services in the stub. Instead allow U-Boot to make use of boot services in case it wants to. It is unclear what it might want though.

Where is the code?

lib/efi payload stub, application, support code. Mostly arch-neutral

arch/x86/cpu/efi x86 support code for running as an EFI application and payload

board/efi/efi-x86_app/efi.c x86 board code for running as an EFI application

board/efi/efi-x86_payload generic x86 EFI payload board support code

common/cmd_efi.c the 'efi' command

– Ben Stoltz, Simon Glass Google, Inc July 2015

- [1] http://www.qemu.org
- [2] http://www.tianocore.org/ovmf/

3.1.3 iSCSI booting with U-Boot and iPXE

Motivation

U-Boot has only a reduced set of supported network protocols. The focus for network booting has been on UDP based protocols. A TCP stack and HTTP support are expected to be integrated in 2018 together with a wget command.

For booting a diskless computer this leaves us with BOOTP or DHCP to get the address of a boot script. TFTP or NFS can be used to load the boot script, the operating system kernel and the initial file system (initrd).

These protocols are insecure. The client cannot validate the authenticity of the contacted servers. And the server cannot verify the identity of the client.

Furthermore the services providing the operating system loader or kernel are not the ones that the operating system typically will use. Especially in a SAN environment this makes updating the operating system a hassle. After installing a new kernel version the boot files have to be copied to the TFTP server directory.

The HTTPS protocol provides certificate based validation of servers. Sensitive data like passwords can be securely transmitted.

The iSCSI protocol is used for connecting storage attached networks. It provides mutual authentication using the CHAP protocol. It typically runs on a TCP transport.

Thus a better solution than DHCP/TFTP/NFS boot would be to load a boot script via HTTPS and to download any other files needed for booting via iSCSI from the same target where the operating system is installed.

An alternative to implementing these protocols in U-Boot is to use an existing software that can run on top of U-Boot. iPXE[1] is the "swiss army knife" of network booting. It supports both HTTPS and iSCSI. It has a scripting engine for fine grained control of the boot process and can provide a command shell.

iPXE can be built as an EFI application (named snp.efi) which can be loaded and run by U-Boot.

Boot sequence

U-Boot loads the EFI application iPXE snp.efi using the bootefi command. This application has network access via the simple network protocol offered by U-Boot.

iPXE executes its internal script. This script may optionally chain load a secondary boot script via HTTPS or open a shell.

For the further boot process iPXE connects to the iSCSI server. This includes the mutual authentication using the CHAP protocol. After the authentication iPXE has access to the iSCSI targets.

For a selected iSCSI target iPXE sets up a handle with the block IO protocol. It uses the ConnectController boot service of U-Boot to request U-Boot to connect a file system driver. U-Boot reads from the iSCSI drive via the block IO protocol offered by iPXE. It creates the partition handles and installs the simple file protocol. Now iPXE can call the simple file protocol to load GRUB[2]. U-Boot uses the block IO protocol offered by iPXE to fulfill the request.

Once GRUB is started it uses the same block IO protocol to load Linux. Via the EFI stub Linux is called as an EFI application:

```
              +--------+         +--------+
              |        | Runs    |        |
              | U-Boot |========>| iPXE   |
              | EFI    |         | snp.efi|
+--------+    |        | DHCP    |        |
|        |<===|********|<========|        |
| DHCP   |    |        | Get IP  |        |
| Server |    |        | Address |        |
|        |===>|********|========>|        |
+--------+    |        | Response|        |
              |        |         |        |
              |        |         |        |
+--------+    |        | HTTPS   |        |
|        |<===|********|<========|        |
| HTTPS  |    |        | Load    |        |
| Server |    |        | Script  |        |
|        |===>|********|========>|        |
+--------+    |        |         |        |
              |        |         |        |
              |        |         |        |
+--------+    |        | iSCSI   |        |
|        |<===|********|<========|        |
| iSCSI  |    |        | Auth    |        |
| Server |===>|********|========>|        |
|        |    |        |         |        |
|        |    |        | Loads   |        |
|        |<===|********|<========|        |       +--------+
|        |    |        | GRUB    |        | Runs  |        |
|        |===>|********|========>|        |======>| GRUB   |
|        |    |        |         |        |       |        |
|        |    |        |         |        |       |        |
|        |    |        |         |        | Loads |        |
|        |<===|********|<========|********|<======|        |      +--------+
|        |    |        |         |        | Linux |        | Runs |        |
|        |===>|********|========>|********|======>|        |=====>| Linux  |
|        |    |        |         |        |       |        |      |        |
+--------+    +--------+         +--------+       +--------+      |        |
                                                                  |        |
                                                                  |        |
                                                                  | ~ ~ ~ ~|
```

Security

The iSCSI protocol is not encrypted. The traffic could be secured using IPsec but neither U-Boot nor iPXE does support this. So we should at least separate the iSCSI traffic from all other network traffic. This can be achieved using a virtual local area network (VLAN).

Configuration

iPXE

For running iPXE on arm64 the bin-arm64-efi/snp.efi build target is needed:

```
git clone http://git.ipxe.org/ipxe.git
cd ipxe/src
make bin-arm64-efi/snp.efi -j6 EMBED=myscript.ipxe
```

The available commands for the boot script are documented at:

http://ipxe.org/cmd

Credentials are managed as environment variables. These are described here:

http://ipxe.org/cfg

iPXE by default will put the CPU to rest when waiting for input. U-Boot does not wake it up due to missing interrupt support. To avoid this behavior create file src/config/local/nap.h:

```
/* nap.h */
#undef NAP_EFIX86
#undef NAP_EFIARM
#define NAP_NULL
```

The supported commands in iPXE are controlled by an include, too. Putting the following into src/config/local/general.h is sufficient for most use cases:

```
/* general.h */
#define NSLOOKUP_CMD            /* Name resolution command */
#define PING_CMD                /* Ping command */
#define NTP_CMD                 /* NTP commands */
#define VLAN_CMD                /* VLAN commands */
#define IMAGE_EFI               /* EFI image support */
#define DOWNLOAD_PROTO_HTTPS    /* Secure Hypertext Transfer Protocol */
#define DOWNLOAD_PROTO_FTP      /* File Transfer Protocol */
#define DOWNLOAD_PROTO_NFS      /* Network File System Protocol */
#define DOWNLOAD_PROTO_FILE     /* Local file system access */
```

Open-iSCSI

When the root file system is on an iSCSI drive you should disable pings and set the replacement timer to a high value in the configuration file [3]:

```
node.conn[0].timeo.noop_out_interval = 0
node.conn[0].timeo.noop_out_timeout = 0
node.session.timeo.replacement_timeout = 86400
```

Links

- [1] https://ipxe.org - iPXE open source boot firmware
- [2] https://www.gnu.org/software/grub/ - GNU GRUB (Grand Unified Bootloader)
- [3] https://github.com/open-iscsi/open-iscsi/blob/master/README - Open-iSCSI README

CHAPTER

FOUR

DRIVER-MODEL DOCUMENTATION

The following holds information on the U-Boot device driver framework: driver-model, including the design details of itself and several driver subsystems.

4.1 Driver Model

4.1.1 Binding/unbinding a driver

This document aims to describe the bind and unbind commands.

For debugging purpose, it should be useful to bind or unbind a driver from the U-boot command line.

The unbind command calls the remove device driver callback and unbind the device from its driver.

The bind command binds a device to its driver.

In some cases it can be useful to be able to bind a device to a driver from the command line. The obvious example is for versatile devices such as USB gadget. Another use case is when the devices are not yet ready at startup and require some setup before the drivers are bound (ex: FPGA which bitsream is fetched from a mass storage or ethernet)

usage:

bind <node path> <driver> bind <class> <index> <driver>

unbind <node path> unbind <class> <index> unbind <class> <index> <driver>

Where:

- <node path> is the node's device tree path
- <class> is one of the class available in the list given by the "dm uclass" command or first column of "dm tree" command.
- <index> is the index of the parent's node (second column of "dm tree" output).
- <driver> is the driver name to bind given by the "dm drivers" command or the by the fourth column of "dm tree" output.

example:

bind usb_dev_generic 0 usb_ether unbind usb_dev_generic 0 usb_ether or unbind eth 1

bind /ocp/omap_dwc3@48380000/usb@48390000 usb_ether unbind /ocp/omap_dwc3@48380000/usb@4839000

4.1.2 Debugging driver model

This document aims to provide help when you cannot work out why driver model is not doing what you expect.

Useful techniques in general

Here are some useful debugging features generally.

- If you are writing a new feature, consider doing it in sandbox instead of on your board. Sandbox has no limits, allows easy debugging (e.g. gdb) and you can write emulators for most common devices.
- Put '#define DEBUG' at the top of a file, to activate all the debug() and log_debug() statements in that file.
- Where logging is used, change the logging level, e.g. in SPL with CONFIG_SPL_LOG_MAX_LEVEL=7 (which is LOGL_DEBUG) and CONFIG_LOG_DEFAULT_LEVEL=7
- Where logging of return values is implemented with log_msg_ret(), set CONFIG_LOG_ERROR_RETURN=y to see exactly where the error is happening
- Make sure you have a debug UART enabled - see CONFIG_DEBUG_UART. With this you can get serial output (printf(), etc.) before the serial driver is running.
- Use a JTAG emulator to set breakpoints and single-step through code

Not that most of these increase code/data size somewhat when enabled.

Failure to locate a device

Let's say you have uclass_first_device_err() and it is not finding anything.

If it is returning an error, then that gives you a clue. Look up linux/errno.h to see errors. Common ones are:

- -ENOMEM which indicates that memory is short. If it happens in SPL or before relocation in U-Boot, check CONFIG_SPL_SYS_MALLOC_F_LEN and CONFIG_SYS_MALLOC_F_LEN as they may need to be larger. Add '#define DEBUG' at the very top of malloc_simple.c to get an idea of where your memory is going.
- -EINVAL which typically indicates that something was missing or wrong in the device tree node. Check that everything is correct and look at the ofdata_to_platdata() method in the driver.

If there is no error, you should check if the device is actually bound. Call dm_dump_all() just before you locate the device to make sure it exists.

If it does not exist, check your device tree compatible strings match up with what the driver expects (in the struct udevice_id array).

If you are using of-platdata (e.g. CONFIG_SPL_OF_PLATDATA), check that the driver name is the same as the first compatible string in the device tree (with invalid-variable characters converted to underscore).

If you are really stuck, putting '#define LOG_DEBUG' at the top of drivers/core/lists.c should show you what is going on.

4.1.3 Design Details

This README contains high-level information about driver model, a unified way of declaring and accessing drivers in U-Boot. The original work was done by:

- Marek Vasut <marex@denx.de>
- Pavel Herrmann <morpheus.ibis@gmail.com>
- Viktor Křivák <viktor.krivak@gmail.com>
- Tomas Hlavacek <tmshlvck@gmail.com>

This has been both simplified and extended into the current implementation by:

- Simon Glass <sjg@chromium.org>

Terminology

Uclass a group of devices which operate in the same way. A uclass provides a way of accessing individual devices within the group, but always using the same interface. For example a GPIO uclass provides operations for get/set value. An I2C uclass may have 10 I2C ports, 4 with one driver, and 6 with another.

Driver some code which talks to a peripheral and presents a higher-level interface to it.

Device an instance of a driver, tied to a particular port or peripheral.

How to try it

Build U-Boot sandbox and run it:

```
make sandbox_defconfig
make
./u-boot -d u-boot.dtb

(type 'reset' to exit U-Boot)
```

There is a uclass called 'demo'. This uclass handles saying hello, and reporting its status. There are two drivers in this uclass:

- simple: Just prints a message for hello, doesn't implement status
- shape: Prints shapes and reports number of characters printed as status

The demo class is pretty simple, but not trivial. The intention is that it can be used for testing, so it will implement all driver model features and provide good code coverage of them. It does have multiple drivers, it handles parameter data and platdata (data which tells the driver how to operate on a particular platform) and it uses private driver data.

To try it, see the example session below:

```
=>demo hello 1
Hello '@' from 07981110: red 4
=>demo status 2
Status: 0
=>demo hello 2
g
r@
e@@
e@@@
n@@@@
g@@@@@
=>demo status 2
Status: 21
=>demo hello 4 ^
  y^^^
 e^^^^^
l^^^^^^^
l^^^^^^^
 o^^^^^
  w^^^
=>demo status 4
Status: 36
=>
```

Running the tests

The intent with driver model is that the core portion has 100% test coverage in sandbox, and every uclass has its own test. As a move towards this, tests are provided in test/dm. To run them, try:

```
./test/py/test.py --bd sandbox --build -k ut_dm -v
```

You should see something like this:

```
(venv)$ ./test/py/test.py --bd sandbox --build -k ut_dm -v
+make O=/root/u-boot/build-sandbox -s sandbox_defconfig
+make O=/root/u-boot/build-sandbox -s -j8
============================= test session starts ==============================
platform linux2 -- Python 2.7.5, pytest-2.9.0, py-1.4.31, pluggy-0.3.1 -- /root/u-boot/
↪venv/bin/python
cachedir: .cache
rootdir: /root/u-boot, inifile:
collected 199 items

test/py/tests/test_ut.py::test_ut_dm_init PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_adc_bind] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_adc_multi_channel_conversion] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_adc_multi_channel_shot] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_adc_single_channel_conversion] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_adc_single_channel_shot] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_adc_supply] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_adc_wrong_channel_selection] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_autobind] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_autobind_uclass_pdata_alloc] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_autobind_uclass_pdata_valid] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_autoprobe] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_child_post_bind] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_child_post_bind_uclass] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_child_pre_probe_uclass] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_children] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_children_funcs] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_children_iterators] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_parent_data] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_parent_data_uclass] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_parent_ops] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_parent_platdata] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_bus_parent_platdata_uclass] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_children] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_clk_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_clk_periph] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_device_get_uclass_id] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_eth] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_eth_act] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_eth_alias] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_eth_prime] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_eth_rotate] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_fdt] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_fdt_offset] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_fdt_pre_reloc] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_fdt_uclass_seq] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_gpio] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_gpio_anon] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_gpio_copy] PASSED
```

(continues on next page)

(continued from previous page)

```
test/py/tests/test_ut.py::test_ut[ut_dm_gpio_leak] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_gpio_phandles] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_gpio_requestf] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_i2c_bytewise] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_i2c_find] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_i2c_offset] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_i2c_offset_len] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_i2c_probe_empty] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_i2c_read_write] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_i2c_speed] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_leak] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_led_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_led_gpio] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_led_label] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_lifecycle] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_mmc_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_net_retry] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_operations] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_ordering] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_pci_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_pci_busnum] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_pci_swapcase] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_platdata] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_pmic_get] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_pmic_io] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_regulator_autoset] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_regulator_autoset_list] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_regulator_get] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_regulator_set_get_current] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_regulator_set_get_enable] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_regulator_set_get_mode] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_power_regulator_set_get_voltage] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_pre_reloc] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_ram_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_regmap_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_regmap_syscon] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_remoteproc_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_remove] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_reset_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_reset_walk] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_rtc_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_rtc_dual] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_rtc_reset] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_rtc_set_get] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_spi_find] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_spi_flash] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_spi_xfer] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_syscon_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_syscon_by_driver_data] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_timer_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_uclass] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_uclass_before_ready] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_uclass_devices_find] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_uclass_devices_find_by_name] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_uclass_devices_get] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_uclass_devices_get_by_name] PASSED
```

(continues on next page)

(continued from previous page)

```
test/py/tests/test_ut.py::test_ut[ut_dm_usb_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_usb_flash] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_usb_keyb] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_usb_multi] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_usb_remove] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_usb_tree] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_usb_tree_remove] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_usb_tree_reorder] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_base] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_bmp] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_bmp_comp] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_chars] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_context] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_rotation1] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_rotation2] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_rotation3] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_text] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_truetype] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_truetype_bs] PASSED
test/py/tests/test_ut.py::test_ut[ut_dm_video_truetype_scroll] PASSED

======================= 84 tests deselected by '-kut_dm' =======================
================== 115 passed, 84 deselected in 3.77 seconds ===================
```

What is going on?

Let's start at the top. The demo command is in cmd/demo.c. It does the usual command processing and then:

```
struct udevice *demo_dev;

ret = uclass_get_device(UCLASS_DEMO, devnum, &demo_dev);
```

UCLASS_DEMO means the class of devices which implement 'demo'. Other classes might be MMC, or GPIO, hashing or serial. The idea is that the devices in the class all share a particular way of working. The class presents a unified view of all these devices to U-Boot.

This function looks up a device for the demo uclass. Given a device number we can find the device because all devices have registered with the UCLASS_DEMO uclass.

The device is automatically activated ready for use by uclass_get_device().

Now that we have the device we can do things like:

```
return demo_hello(demo_dev, ch);
```

This function is in the demo uclass. It takes care of calling the 'hello' method of the relevant driver. Bearing in mind that there are two drivers, this particular device may use one or other of them.

The code for demo_hello() is in drivers/demo/demo-uclass.c:

```
int demo_hello(struct udevice *dev, int ch)
{
        const struct demo_ops *ops = device_get_ops(dev);

        if (!ops->hello)
                return -ENOSYS;
```

(continues on next page)

(continued from previous page)

```
        return ops->hello(dev, ch);
}
```

As you can see it just calls the relevant driver method. One of these is in drivers/demo/demo-simple.c:

```
static int simple_hello(struct udevice *dev, int ch)
{
        const struct dm_demo_pdata *pdata = dev_get_platdata(dev);

        printf("Hello from %08x: %s %d\n", map_to_sysmem(dev),
               pdata->colour, pdata->sides);

        return 0;
}
```

So that is a trip from top (command execution) to bottom (driver action) but it leaves a lot of topics to address.

Declaring Drivers

A driver declaration looks something like this (see drivers/demo/demo-shape.c):

```
static const struct demo_ops shape_ops = {
        .hello = shape_hello,
        .status = shape_status,
};

U_BOOT_DRIVER(demo_shape_drv) = {
        .name   = "demo_shape_drv",
        .id     = UCLASS_DEMO,
        .ops    = &shape_ops,
        .priv_data_size = sizeof(struct shape_data),
};
```

This driver has two methods (hello and status) and requires a bit of private data (accessible through dev_get_priv(dev) once the driver has been probed). It is a member of UCLASS_DEMO so will register itself there.

In U_BOOT_DRIVER it is also possible to specify special methods for bind and unbind, and these are called at appropriate times. For many drivers it is hoped that only 'probe' and 'remove' will be needed.

The U_BOOT_DRIVER macro creates a data structure accessible from C, so driver model can find the drivers that are available.

The methods a device can provide are documented in the device.h header. Briefly, they are:

- bind - make the driver model aware of a device (bind it to its driver)
- unbind - make the driver model forget the device
- ofdata_to_platdata - convert device tree data to platdata - see later
- probe - make a device ready for use
- remove - remove a device so it cannot be used until probed again

The sequence to get a device to work is bind, ofdata_to_platdata (if using device tree) and probe.

Platform Data

Note: platform data is the old way of doing things. It is basically a C structure which is passed to drivers to tell them about platform-specific settings like the address of its registers, bus speed, etc. Device tree is now the preferred way of handling this. Unless you have a good reason not to use device tree (the main one being you need serial support in SPL and don't have enough SRAM for the cut-down device tree and libfdt libraries) you should stay away from platform data.

Platform data is like Linux platform data, if you are familiar with that. It provides the board-specific information to start up a device.

Why is this information not just stored in the device driver itself? The idea is that the device driver is generic, and can in principle operate on any board that has that type of device. For example, with modern highly-complex SoCs it is common for the IP to come from an IP vendor, and therefore (for example) the MMC controller may be the same on chips from different vendors. It makes no sense to write independent drivers for the MMC controller on each vendor's SoC, when they are all almost the same. Similarly, we may have 6 UARTs in an SoC, all of which are mostly the same, but lie at different addresses in the address space.

Using the UART example, we have a single driver and it is instantiated 6 times by supplying 6 lots of platform data. Each lot of platform data gives the driver name and a pointer to a structure containing information about this instance - e.g. the address of the register space. It may be that one of the UARTS supports RS-485 operation - this can be added as a flag in the platform data, which is set for this one port and clear for the rest.

Think of your driver as a generic piece of code which knows how to talk to a device, but needs to know where it is, any variant/option information and so on. Platform data provides this link between the generic piece of code and the specific way it is bound on a particular board.

Examples of platform data include:

- The base address of the IP block's register space
- **Configuration options, like:**
  - the SPI polarity and maximum speed for a SPI controller
  - the I2C speed to use for an I2C device
  - the number of GPIOs available in a GPIO device

Where does the platform data come from? It is either held in a structure which is compiled into U-Boot, or it can be parsed from the Device Tree (see 'Device Tree' below).

For an example of how it can be compiled in, see demo-pdata.c which sets up a table of driver names and their associated platform data. The data can be interpreted by the drivers however they like - it is basically a communication scheme between the board-specific code and the generic drivers, which are intended to work on any board.

Drivers can access their data via dev->info->platdata. Here is the declaration for the platform data, which would normally appear in the board file.

```
static const struct dm_demo_pdata red_square = {
        .colour = "red",
        .sides = 4.
};

static const struct driver_info info[] = {
        {
                .name = "demo_shape_drv",
                .platdata = &red_square,
        },
};
```

(continues on next page)

(continued from previous page)

```
demo1 = driver_bind(root, &info[0]);
```

Device Tree

While platdata is useful, a more flexible way of providing device data is by using device tree. In U-Boot you should use this where possible. Avoid sending patches which make use of the U_BOOT_DEVICE() macro unless strictly necessary.

With device tree we replace the above code with the following device tree fragment:

```
red-square {
        compatible = "demo-shape";
        colour = "red";
        sides = <4>;
};
```

This means that instead of having lots of U_BOOT_DEVICE() declarations in the board file, we put these in the device tree. This approach allows a lot more generality, since the same board file can support many types of boards (e,g. with the same SoC) just by using different device trees. An added benefit is that the Linux device tree can be used, thus further simplifying the task of board-bring up either for U-Boot or Linux devs (whoever gets to the board first!).

The easiest way to make this work it to add a few members to the driver:

```
.platdata_auto_alloc_size = sizeof(struct dm_test_pdata),
.ofdata_to_platdata = testfdt_ofdata_to_platdata,
```

The 'auto_alloc' feature allowed space for the platdata to be allocated and zeroed before the driver's ofdata_to_platdata() method is called. The ofdata_to_platdata() method, which the driver write supplies, should parse the device tree node for this device and place it in dev->platdata. Thus when the probe method is called later (to set up the device ready for use) the platform data will be present.

Note that both methods are optional. If you provide an ofdata_to_platdata method then it will be called first (during activation). If you provide a probe method it will be called next. See Driver Lifecycle below for more details.

If you don't want to have the platdata automatically allocated then you can leave out platdata_auto_alloc_size. In this case you can use malloc in your ofdata_to_platdata (or probe) method to allocate the required memory, and you should free it in the remove method.

The driver model tree is intended to mirror that of the device tree. The root driver is at device tree offset 0 (the root node, '/'), and its children are the children of the root node.

In order for a device tree to be valid, the content must be correct with respect to either device tree specification (https://www.devicetree.org/specifications/) or the device tree bindings that are found in the doc/device-tree-bindings directory. When not U-Boot specific the bindings in this directory tend to come from the Linux Kernel. As such certain design decisions may have been made already for us in terms of how specific devices are described and bound. In most circumstances we wish to retain compatibility without additional changes being made to the device tree source files.

Declaring Uclasses

The demo uclass is declared like this:

```
UCLASS_DRIVER(demo) = {
        .id             = UCLASS_DEMO,
};
```

It is also possible to specify special methods for probe, etc. The uclass numbering comes from include/dm/uclass-id.h. To add a new uclass, add to the end of the enum there, then declare your uclass as above.

Device Sequence Numbers

U-Boot numbers devices from 0 in many situations, such as in the command line for I2C and SPI buses, and the device names for serial ports (serial0, serial1, …). Driver model supports this numbering and permits devices to be locating by their 'sequence'. This numbering uniquely identifies a device in its uclass, so no two devices within a particular uclass can have the same sequence number.

Sequence numbers start from 0 but gaps are permitted. For example, a board may have I2C buses 1, 4, 5 but no 0, 2 or 3. The choice of how devices are numbered is up to a particular board, and may be set by the SoC in some cases. While it might be tempting to automatically renumber the devices where there are gaps in the sequence, this can lead to confusion and is not the way that U-Boot works.

Each device can request a sequence number. If none is required then the device will be automatically allocated the next available sequence number.

To specify the sequence number in the device tree an alias is typically used. Make sure that the uclass has the DM_UC_FLAG_SEQ_ALIAS flag set.

```
aliases {
        serial2 = "/serial@22230000";
};
```

This indicates that in the uclass called "serial", the named node ("/serial@22230000") will be given sequence number 2. Any command or driver which requests serial device 2 will obtain this device.

More commonly you can use node references, which expand to the full path:

```
aliases {
        serial2 = &serial_2;
};
...
serial_2: serial@22230000 {
...
};
```

The alias resolves to the same string in this case, but this version is easier to read.

Device sequence numbers are resolved when a device is probed. Before then the sequence number is only a request which may or may not be honoured, depending on what other devices have been probed. However the numbering is entirely under the control of the board author so a conflict is generally an error.

Bus Drivers

A common use of driver model is to implement a bus, a device which provides access to other devices. Example of buses include SPI and I2C. Typically the bus provides some sort of transport or translation that makes it possible to talk to the devices on the bus.

Driver model provides some useful features to help with implementing buses. Firstly, a bus can request that its children store some 'parent data' which can be used to keep track of child state. Secondly, the bus can define methods which are called when a child is probed or removed. This is similar to the methods the uclass driver provides. Thirdly, per-child platform data can be provided to specify things like the child's address on the bus. This persists across child probe()/remove() cycles.

For consistency and ease of implementation, the bus uclass can specify the per-child platform data, so that it can be the same for all children of buses in that uclass. There are also uclass methods which can be called when children are bound and probed.

Here an explanation of how a bus fits with a uclass may be useful. Consider a USB bus with several devices attached to it, each from a different (made up) uclass:

```
xhci_usb (UCLASS_USB)
   eth (UCLASS_ETH)
   camera (UCLASS_CAMERA)
   flash (UCLASS_FLASH_STORAGE)
```

Each of the devices is connected to a different address on the USB bus. The bus device wants to store this address and some other information such as the bus speed for each device.

To achieve this, the bus device can use dev->parent_platdata in each of its three children. This can be auto-allocated if the bus driver (or bus uclass) has a non-zero value for per_child_platdata_auto_alloc_size. If not, then the bus device or uclass can allocate the space itself before the child device is probed.

Also the bus driver can define the child_pre_probe() and child_post_remove() methods to allow it to do some processing before the child is activated or after it is deactivated.

Similarly the bus uclass can define the child_post_bind() method to obtain the per-child platform data from the device tree and set it up for the child. The bus uclass can also provide a child_pre_probe() method. Very often it is the bus uclass that controls these features, since it avoids each driver having to do the same processing. Of course the driver can still tweak and override these activities.

Note that the information that controls this behaviour is in the bus's driver, not the child's. In fact it is possible that child has no knowledge that it is connected to a bus. The same child device may even be used on two different bus types. As an example. the 'flash' device shown above may also be connected on a SATA bus or standalone with no bus:

```
xhci_usb (UCLASS_USB)
   flash (UCLASS_FLASH_STORAGE)  - parent data/methods defined by USB bus

sata (UCLASS_AHCI)
   flash (UCLASS_FLASH_STORAGE)  - parent data/methods defined by SATA bus

flash (UCLASS_FLASH_STORAGE)  - no parent data/methods (not on a bus)
```

Above you can see that the driver for xhci_usb/sata controls the child's bus methods. In the third example the device is not on a bus, and therefore will not have these methods at all. Consider the case where the flash device defines child methods. These would be used for *its* children, and would be quite separate from the methods defined by the driver for the bus that the flash device is connetced to. The act of attaching a device to a parent device which is a bus, causes the device to start behaving like a bus device, regardless of its own views on the matter.

The uclass for the device can also contain data private to that uclass. But note that each device on the bus may be a member of a different uclass, and this data has nothing to do with the child data for each child on the bus. It is the bus' uclass that controls the child with respect to the bus.

Driver Lifecycle

Here are the stages that a device goes through in driver model. Note that all methods mentioned here are optional - e.g. if there is no probe() method for a device then it will not be called. A simple device may have very few methods actually defined.

Bind stage

U-Boot discovers devices using one of these two methods:

- Scan the U_BOOT_DEVICE() definitions. U-Boot looks up the name specified by each, to find the appropriate U_BOOT_DRIVER() definition. In this case, there is no path by which driver_data may be provided, but the U_BOOT_DEVICE() may provide platdata.

- Scan through the device tree definitions. U-Boot looks at top-level nodes in the the device tree. It looks at the compatible string in each node and uses the of_match table of the U_BOOT_DRIVER() structure to find the right driver for each node. In this case, the of_match table may provide a driver_data value, but platdata cannot be provided until later.

For each device that is discovered, U-Boot then calls device_bind() to create a new device, initializes various core fields of the device object such as name, uclass & driver, initializes any optional fields of the device object that are applicable such as of_offset, driver_data & platdata, and finally calls the driver's bind() method if one is defined.

At this point all the devices are known, and bound to their drivers. There is a 'struct udevice' allocated for all devices. However, nothing has been activated (except for the root device). Each bound device that was created from a U_BOOT_DEVICE() declaration will hold the platdata pointer specified in that declaration. For a bound device created from the device tree, platdata will be NULL, but of_offset will be the offset of the device tree node that caused the device to be created. The uclass is set correctly for the device.

The device's bind() method is permitted to perform simple actions, but should not scan the device tree node, not initialise hardware, nor set up structures or allocate memory. All of these tasks should be left for the probe() method.

Note that compared to Linux, U-Boot's driver model has a separate step of probe/remove which is independent of bind/unbind. This is partly because in U-Boot it may be expensive to probe devices and we don't want to do it until they are needed, or perhaps until after relocation.

Reading ofdata

Most devices have data in the device tree which they can read to find out the base address of hardware registers and parameters relating to driver operation. This is called 'ofdata' (Open-Firmware data).

The device's_ofdata_to_platdata() implemnents allocation and reading of platdata. A parent's ofdata is always read before a child.

The steps are:

1. If priv_auto_alloc_size is non-zero, then the device-private space is allocated for the device and zeroed. It will be accessible as dev->priv. The driver can put anything it likes in there, but should use it for run-time information, not platform data (which should be static and known before the device is probed).

2. If platdata_auto_alloc_size is non-zero, then the platform data space is allocated. This is only useful for device tree operation, since otherwise you would have to specific the platform data in the U_BOOT_DEVICE() declaration. The space is allocated for the device and zeroed. It will be accessible as dev->platdata.

3. If the device's uclass specifies a non-zero per_device_auto_alloc_size, then this space is allocated and zeroed also. It is allocated for and stored in the device, but it is uclass data. owned by the uclass driver. It is possible for the device to access it.

4. If the device's immediate parent specifies a per_child_auto_alloc_size then this space is allocated. This is intended for use by the parent device to keep track of things related to the child. For example a USB flash stick attached to a USB host controller would likely use this space. The controller can hold information about the USB state of each of its children.

5. If the driver provides an ofdata_to_platdata() method, then this is called to convert the device tree data into platform data. This should do various calls like dev_read_u32(dev, …) to access the node and store the resulting information into dev->platdata. After this point, the device works the same way whether it was bound using a device tree node or U_BOOT_DEVICE() structure. In either case, the platform data is now stored in the platdata structure. Typically you will use the platdata_auto_alloc_size feature to specify the size of the platform data structure, and U-Boot will automatically allocate and zero it for you before entry to ofdata_to_platdata(). But if not, you can allocate it yourself in ofdata_to_platdata(). Note that it is preferable to do all the device tree decoding in ofdata_to_platdata() rather than in probe(). (Apart from the ugliness of

mixing configuration and run-time data, one day it is possible that U-Boot will cache platform data for devices which are regularly de/activated).

5. The device is marked 'platdata valid'.

Note that ofdata reading is always done (for a child and all its parents) before probing starts. Thus devices go through two distinct states when probing: reading platform data and actually touching the hardware to bring the device up.

Having probing separate from ofdata-reading helps deal with of-platdata, where the probe() method is common to both DT/of-platdata operation, but the ofdata_to_platdata() method is implemented differently.

Another case has come up where this separate is useful. Generation of ACPI tables uses the of-platdata but does not want to probe the device. Probing would cause U-Boot to violate one of its design principles, viz that it should only probe devices that are used. For ACPI we want to generate a table for each device, even if U-Boot does not use it. In fact it may not even be possible to probe the device - e.g. an SD card which is not present will cause an error on probe, yet we still must tell Linux about the SD card connector in case it is used while Linux is running.

It is important that the ofdata_to_platdata() method does not actually probe the device itself. However there are cases where other devices must be probed in the ofdata_to_platdata() method. An example is where a device requires a GPIO for it to operate. To select a GPIO obviously requires that the GPIO device is probed. This is OK when used by common, core devices such as GPIO, clock, interrupts, reset and the like.

If your device relies on its parent setting up a suitable address space, so that dev_read_addr() works correctly, then make sure that the parent device has its setup code in ofdata_to_platdata(). If it has it in the probe method, then you cannot call dev_read_addr() from the child device's ofdata_to_platdata() method. Move it to probe() instead. Buses like PCI can fall afoul of this rule.

Activation/probe

When a device needs to be used, U-Boot activates it, by first reading ofdata as above and then following these steps (see device_probe()):

1. All parent devices are probed. It is not possible to activate a device unless its predecessors (all the way up to the root device) are activated. This means (for example) that an I2C driver will require that its bus be activated.

2. The device's sequence number is assigned, either the requested one (assuming no conflicts) or the next available one if there is a conflict or nothing particular is requested.

4. The device's probe() method is called. This should do anything that is required by the device to get it going. This could include checking that the hardware is actually present, setting up clocks for the hardware and setting up hardware registers to initial values. The code in probe() can access:

   - platform data in dev->platdata (for configuration)
   - private data in dev->priv (for run-time state)
   - uclass data in dev->uclass_priv (for things the uclass stores about this device)

   Note: If you don't use priv_auto_alloc_size then you will need to allocate the priv space here yourself. The same applies also to platdata_auto_alloc_size. Remember to free them in the remove() method.

5. The device is marked 'activated'

10. The uclass's post_probe() method is called, if one exists. This may cause the uclass to do some housekeeping to record the device as activated and 'known' by the uclass.

Running stage

The device is now activated and can be used. From now until it is removed all of the above structures are accessible. The device appears in the uclass's list of devices (so if the device is in UCLASS_GPIO it will appear as a device in the GPIO uclass). This is the 'running' state of the device.

Removal stage

When the device is no-longer required, you can call device_remove() to remove it. This performs the probe steps in reverse:

1. The uclass's pre_remove() method is called, if one exists. This may cause the uclass to do some housekeeping to record the device as deactivated and no-longer 'known' by the uclass.

2. All the device's children are removed. It is not permitted to have an active child device with a non-active parent. This means that device_remove() is called for all the children recursively at this point.

3. The device's remove() method is called. At this stage nothing has been deallocated so platform data, private data and the uclass data will all still be present. This is where the hardware can be shut down. It is intended that the device be completely inactive at this point, For U-Boot to be sure that no hardware is running, it should be enough to remove all devices.

4. The device memory is freed (platform data, private data, uclass data, parent data).

   Note: Because the platform data for a U_BOOT_DEVICE() is defined with a static pointer, it is not de-allocated during the remove() method. For a device instantiated using the device tree data, the platform data will be dynamically allocated, and thus needs to be deallocated during the remove() method, either:

   - if the platdata_auto_alloc_size is non-zero, the deallocation happens automatically within the driver model core; or
   - when platdata_auto_alloc_size is 0, both the allocation (in probe() or preferably of-data_to_platdata()) and the deallocation in remove() are the responsibility of the driver author.

5. The device sequence number is set to -1, meaning that it no longer has an allocated sequence. If the device is later reactivated and that sequence number is still free, it may well receive the name sequence number again. But from this point, the sequence number previously used by this device will no longer exist (think of SPI bus 2 being removed and bus 2 is no longer available for use).

6. The device is marked inactive. Note that it is still bound, so the device structure itself is not freed at this point. Should the device be activated again, then the cycle starts again at step 2 above.

Unbind stage

The device is unbound. This is the step that actually destroys the device. If a parent has children these will be destroyed first. After this point the device does not exist and its memory has be deallocated.

Data Structures

Driver model uses a doubly-linked list as the basic data structure. Some nodes have several lists running through them. Creating a more efficient data structure might be worthwhile in some rare cases, once we understand what the bottlenecks are.

Changes since v1

For the record, this implementation uses a very similar approach to the original patches, but makes at least the following changes:

- Tried to aggressively remove boilerplate, so that for most drivers there is little or no 'driver model' code to write.
- Moved some data from code into data structure - e.g. store a pointer to the driver operations structure in the driver, rather than passing it to the driver bind function.
- Rename some structures to make them more similar to Linux (struct udevice instead of struct instance, struct platdata, etc.)
- Change the name 'core' to 'uclass', meaning U-Boot class. It seems that this concept relates to a class of drivers (or a subsystem). We shouldn't use 'class' since it is a C++ reserved word, so U-Boot class (uclass) seems better than 'core'.
- Remove 'struct driver_instance' and just use a single 'struct udevice'. This removes a level of indirection that doesn't seem necessary.
- Built in device tree support, to avoid the need for platdata
- Removed the concept of driver relocation, and just make it possible for the new driver (created after relocation) to access the old driver data. I feel that relocation is a very special case and will only apply to a few drivers, many of which can/will just re-init anyway. So the overhead of dealing with this might not be worth it.
- Implemented a GPIO system, trying to keep it simple

Pre-Relocation Support

For pre-relocation we simply call the driver model init function. Only drivers marked with DM_FLAG_PRE_RELOC or the device tree 'u-boot,dm-pre-reloc' property are initialised prior to relocation. This helps to reduce the driver model overhead. This flag applies to SPL and TPL as well, if device tree is enabled (CONFIG_OF_CONTROL) there.

Note when device tree is enabled, the device tree 'u-boot,dm-pre-reloc' property can provide better control granularity on which device is bound before relocation. While with DM_FLAG_PRE_RELOC flag of the driver all devices with the same driver are bound, which requires allocation a large amount of memory. When device tree is not used, DM_FLAG_PRE_RELOC is the only way for statically declared devices via U_BOOT_DEVICE() to be bound prior to relocation.

It is possible to limit this to specific relocation steps, by using the more specialized 'u-boot,dm-spl' and 'u-boot,dm-tpl' flags in the device tree node. For U-Boot proper you can use 'u-boot,dm-pre-proper' which means that it will be processed (and a driver bound) in U-Boot proper prior to relocation, but will not be available in SPL or TPL.

To reduce the size of SPL and TPL, only the nodes with pre-relocation properties ('u-boot,dm-pre-reloc', 'u-boot,dm-spl' or 'u-boot,dm-tpl') are keept in their device trees (see README.SPL for details); the remaining nodes are always bound.

Then post relocation we throw that away and re-init driver model again. For drivers which require some sort of continuity between pre- and post-relocation devices, we can provide access to the pre-relocation device pointers, but this is not currently implemented (the root device pointer is saved but not made available through the driver model API).

SPL Support

Driver model can operate in SPL. Its efficient implementation and small code size provide for a small overhead which is acceptable for all but the most constrained systems.

To enable driver model in SPL, define CONFIG_SPL_DM. You might want to consider the following option also. See the main README for more details.

- CONFIG_SYS_MALLOC_SIMPLE
- CONFIG_DM_WARN
- CONFIG_DM_DEVICE_REMOVE
- CONFIG_DM_STDIO

Enabling Driver Model

Driver model is being brought into U-Boot gradually. As each subsystems gets support, a uclass is created and a CONFIG to enable use of driver model for that subsystem.

For example CONFIG_DM_SERIAL enables driver model for serial. With that defined, the old serial support is not enabled, and your serial driver must conform to driver model. With that undefined, the old serial support is enabled and driver model is not available for serial. This means that when you convert a driver, you must either convert all its boards, or provide for the driver to be compiled both with and without driver model (generally this is not very hard).

See the main README for full details of the available driver model CONFIG options.

Things to punt for later

Uclasses are statically numbered at compile time. It would be possible to change this to dynamic numbering, but then we would require some sort of lookup service, perhaps searching by name. This is slightly less efficient so has been left out for now. One small advantage of dynamic numbering might be fewer merge conflicts in uclass-id.h.

4.1.4 Ethernet Driver Guide

The networking stack in Das U-Boot is designed for multiple network devices to be easily added and controlled at runtime. This guide is meant for people who wish to review the net driver stack with an eye towards implementing your own ethernet device driver. Here we will describe a new pseudo 'APE' driver.

Most existing drivers do already - and new network driver MUST - use the U-Boot core driver model. Generic information about this can be found in doc/driver-model/design.rst, this document will thus focus on the network specific code parts. Some drivers are still using the old Ethernet interface, differences between the two and hints about porting will be handled at the end.

Driver framework

A network driver following the driver model must declare itself using the UCLASS_ETH .id field in the U-Boot driver struct:

```
U_BOOT_DRIVER(eth_ape) = {
        .name                   = "eth_ape",
        .id                     = UCLASS_ETH,
        .of_match               = eth_ape_ids,
        .ofdata_to_platdata     = eth_ape_ofdata_to_platdata,
        .probe                  = eth_ape_probe,
        .ops                    = &eth_ape_ops,
        .priv_auto_alloc_size   = sizeof(struct eth_ape_priv),
        .platdata_auto_alloc_size = sizeof(struct eth_ape_pdata),
        .flags                  = DM_FLAG_ALLOC_PRIV_DMA,
};
```

struct eth_ape_priv contains runtime per-instance data, like buffers, pointers to current descriptors, current speed settings, pointers to PHY related data (like struct mii_dev) and so on. Declaring its size in .priv_auto_alloc_size will let the driver framework allocate it at the right time. It can be retrieved using a dev_get_priv(dev) call.

struct eth_ape_pdata contains static platform data, like the MMIO base address, a hardware variant, the MAC address. `struct eth_pdata eth_pdata` as the first member of this struct helps to avoid duplicated code. If you don't need any more platform data beside the standard member, just use sizeof(struct eth_pdata) for the platdata_auto_alloc_size.

PCI devices add a line pointing to supported vendor/device ID pairs:

```
static struct pci_device_id supported[] = {
        { PCI_DEVICE(PCI_VENDOR_ID_APE, 0x4223) },
        {}
};

U_BOOT_PCI_DEVICE(eth_ape, supported);
```

It is also possible to declare support for a whole class of PCI devices:

```
{ PCI_DEVICE_CLASS(PCI_CLASS_SYSTEM_SDHCI << 8, 0xffff00) },
```

Device probing and instantiation will be handled by the driver model framework, so follow the guidelines there. The probe() function would initialise the platform specific parts of the hardware, like clocks, resets, GPIOs, the MDIO bus. Also it would take care of any special PHY setup (power rails, enable bits for internal PHYs, etc.).

Driver methods

The real work will be done in the driver method functions the driver provides by defining the members of struct eth_ops:

```
struct eth_ops {
        int (*start)(struct udevice *dev);
        int (*send)(struct udevice *dev, void *packet, int length);
        int (*recv)(struct udevice *dev, int flags, uchar **packetp);
        int (*free_pkt)(struct udevice *dev, uchar *packet, int length);
        void (*stop)(struct udevice *dev);
        int (*mcast)(struct udevice *dev, const u8 *enetaddr, int join);
        int (*write_hwaddr)(struct udevice *dev);
        int (*read_rom_hwaddr)(struct udevice *dev);
};
```

An up-to-date version of this struct together with more information can be found in include/net.h.

Only start, stop, send and recv are required, the rest are optional and are handled by generic code or ignored if not provided.

The **start** function initialises the hardware and gets it ready for send/recv operations. You often do things here such as resetting the MAC and/or PHY, and waiting for the link to autonegotiate. You should also take the opportunity to program the device's MAC address with the enetaddr member of the generic struct eth_pdata (which would be the first member of your own platdata struct). This allows the rest of U-Boot to dynamically change the MAC address and have the new settings be respected.

The **send** function does what you think – transmit the specified packet whose size is specified by length (in bytes). The packet buffer can (and will!) be reused for subsequent calls to send(), so it must be no longer used when the send() function returns. The easiest way to achieve this is to wait until the transmission is complete. Alternatively, if supported by the hardware, just waiting for the buffer to be consumed (by some DMA engine) might be an option as well. Another way of consuming the buffer could be to copy the data to be send, then just queue the copied packet (for instance handing it over to a DMA engine), and return immediately afterwards. In any case you should leave the state such that the send function can be called multiple times in a row.

The **recv** function polls for availability of a new packet. If none is available, it must return with -EAGAIN. If a packet has been received, make sure it is accessible to the CPU (invalidate caches if needed), then write its address to the packetp pointer, and return the length. If there is an error (receive error, too short or too long packet), return 0 if you require the packet to be cleaned up normally, or a negative error code otherwise (cleanup not necessary or already done). The U-Boot network stack will then process the packet.

If **free_pkt** is defined, U-Boot will call it after a received packet has been processed, so the packet buffer can be freed or recycled. Typically you would hand it back to the hardware to acquire another packet. free_pkt() will be called after recv(), for the same packet, so you don't necessarily need to infer the buffer to free from the packet pointer, but can rely on that being the last packet that recv() handled. The common code sets up packet buffers for you already in the .bss (net_rx_packets), so there should be no need to allocate your own. This doesn't mean you must use the net_rx_packets array however; you're free to use any buffer you wish.

The **stop** function should turn off / disable the hardware and place it back in its reset state. It can be called at any time (before any call to the related start() function), so make sure it can handle this sort of thing.

The (optional) **write_hwaddr** function should program the MAC address stored in pdata->enetaddr into the Ethernet controller.

So the call graph at this stage would look something like:

```
(some net operation (ping / tftp / whatever...))
eth_init()
        ops->start()
eth_send()
        ops->send()
eth_rx()
        ops->recv()
        (process packet)
        if (ops->free_pkt)
                ops->free_pkt()
eth_halt()
        ops->stop()
```

CONFIG_PHYLIB / CONFIG_CMD_MII

If your device supports banging arbitrary values on the MII bus (pretty much every device does), you should add support for the mii command. Doing so is fairly trivial and makes debugging mii issues a lot easier at runtime.

In your driver's `probe()` function, add a call to mdio_alloc() and mdio_register() like so:

```
bus = mdio_alloc();
if (!bus) {
        ...
        return -ENOMEM;
}

bus->read = ape_mii_read;
bus->write = ape_mii_write;
mdio_register(bus);
```

And then define the mii_read and mii_write functions if you haven't already. Their syntax is straightforward:

```
int mii_read(struct mii_dev *bus, int addr, int devad, int reg);
int mii_write(struct mii_dev *bus, int addr, int devad, int reg,
              u16 val);
```

The read function should read the register 'reg' from the phy at address 'addr' and return the result to its caller. The implementation for the write function should logically follow.

Legacy network drivers

!!! WARNING !!!

This section below describes the old way of doing things. No new Ethernet drivers should be implemented this way. All new drivers should be written against the U-Boot core driver model, as described above.

The actual callback functions are fairly similar, the differences are:

- `start()` is called `init()`
- `stop()` is called `halt()`
- The `recv()` function must loop until all packets have been received, for each packet it must call the net_process_received_packet() function, handing it over the pointer and the length. Afterwards it should free the packet, before checking for new data.

For porting an old driver to the new driver model, split the existing recv() function into the actual new recv() function, just fetching **one** packet, remove the call to net_process_received_packet(), then move the packet cleanup into the `free_pkt()` function.

Registering the driver and probing a device is handled very differently, follow the recommendations in the driver model design documentation for instructions on how to port this over. For the records, the old way of initialising a network driver is as follows:

Old network driver registration

When U-Boot initializes, it will call the common function eth_initialize(). This will in turn call the board-specific board_eth_init() (or if that fails, the cpu-specific cpu_eth_init()). These board-specific functions can do random system handling, but ultimately they will call the driver-specific register function which in turn takes care of initializing that particular instance.

Keep in mind that you should code the driver to avoid storing state in global data as someone might want to hook up two of the same devices to one board. Any such information that is specific to an interface should be stored in a private, driver-defined data structure and pointed to by eth->priv (see below).

So the call graph at this stage would look something like:

```
board_init()
        eth_initialize()
                board_eth_init() / cpu_eth_init()
                        driver_register()
                                initialize eth_device
                                eth_register()
```

At this point in time, the only thing you need to worry about is the driver's register function. The pseudo code would look something like:

```
int ape_register(struct bd_info *bis, int iobase)
{
        struct ape_priv *priv;
```

(continues on next page)

(continued from previous page)

```
        struct eth_device *dev;
        struct mii_dev *bus;

        priv = malloc(sizeof(*priv));
        if (priv == NULL)
                return -ENOMEM;

        dev = malloc(sizeof(*dev));
        if (dev == NULL) {
                free(priv);
                return -ENOMEM;
        }

        /* setup whatever private state you need */

        memset(dev, 0, sizeof(*dev));
        sprintf(dev->name, "APE");

        /*
         * if your device has dedicated hardware storage for the
         * MAC, read it and initialize dev->enetaddr with it
         */
        ape_mac_read(dev->enetaddr);

        dev->iobase = iobase;
        dev->priv = priv;
        dev->init = ape_init;
        dev->halt = ape_halt;
        dev->send = ape_send;
        dev->recv = ape_recv;
        dev->write_hwaddr = ape_write_hwaddr;

        eth_register(dev);

#ifdef CONFIG_PHYLIB
        bus = mdio_alloc();
        if (!bus) {
                free(priv);
                free(dev);
                return -ENOMEM;
        }

        bus->read = ape_mii_read;
        bus->write = ape_mii_write;
        mdio_register(bus);
#endif

        return 1;
}
```

The exact arguments needed to initialize your device are up to you. If you need to pass more/less arguments, that's fine. You should also add the prototype for your new register function to include/netdev.h.

The return value for this function should be as follows: < 0 - failure (hardware failure, not probe failure) >=0 - number of interfaces detected

You might notice that many drivers seem to use xxx_initialize() rather than xxx_register(). This is the old naming convention and should be avoided as it causes confusion with the driver-specific init function.

Other than locating the MAC address in dedicated hardware storage, you should not touch the hardware in anyway. That step is handled in the driver-specific init function. Remember that we are only registering the device here, we are not checking its state or doing random probing.

4.1.5 Pre-relocation device tree manipulation

Purpose

In certain markets, it is beneficial for manufacturers of embedded devices to offer certain ranges of products, where the functionality of the devices within one series either don't differ greatly from another, or can be thought of as "extensions" of each other, where one device only differs from another in the addition of a small number of features (e.g. an additional output connector).

To realize this in hardware, one method is to have a motherboard, and several possible daughter boards that can be attached to this mother board. Different daughter boards then either offer the slightly different functionality, or the addition of the daughter board to the device realizes the "extension" of functionality to the device described previously.

For the software, we obviously want to reuse components for all these variations of the device. This means that the software somehow needs to cope with the situation that certain ICs may or may not be present on any given system, depending on which daughter boards are connected to the motherboard.

In the Linux kernel, one possible solution to this problem is to employ the device tree overlay mechanism: There exists one "base" device tree, which features only the components guaranteed to exist in all varieties of the device. At the start of the kernel, the presence and type of the daughter boards is then detected, and the corresponding device tree overlays are applied to support the components on the daughter boards.

Note that the components present on every variety of the board must, of course, provide a way to find out if and which daughter boards are installed for this mechanism to work.

In the U-Boot boot loader, support for device tree overlays has recently been integrated, and is used on some boards to alter the device tree that is later passed to Linux. But since U-Boot's driver model, which is device tree-based as well, is being used in more and more drivers, the same problem of altering the device tree starts cropping up in U-Boot itself as well.

An additional problem with the device tree in U-Boot is that it is read-only, and the current mechanisms don't allow easy manipulation of the device tree after the driver model has been initialized. While migrating to a live device tree (at least after the relocation) would greatly simplify the solution of this problem, it is a non-negligible task to implement it, an a interim solution is needed to address the problem at least in the medium-term.

Hence, we propose a solution to this problem by offering a board-specific call-back function, which is passed a writeable pointer to the device tree. This function is called before the device tree is relocated, and specifically before the main U-Boot's driver model is instantiated, hence the main U-Boot "sees" all modifications to the device tree made in this function. Furthermore, we have the pre-relocation driver model at our disposal at this stage, which means that we can query the hardware for the existence and variety of the components easily.

Implementation

To take advantage of the pre-relocation device tree manipulation mechanism, boards have to implement the function board_fix_fdt, which has the following signature:

```
int board_fix_fdt (void *rw_fdt_blob)
```

The passed-in void pointer is a writeable pointer to the device tree, which can be used to manipulate the device tree using e.g. functions from include/fdt_support.h. The return value should either be 0 in case of successful execution of the device tree manipulation or something else for a failure. Note that returning a non-null value from the function will unrecoverably halt the boot process, as with any function from init_sequence_f (in common/board_f.c).

Furthermore, the Kconfig option OF_BOARD_FIXUP has to be set for the function to be called:

```
Device Tree Control
-> [*] Board-specific manipulation of Device Tree
```

WARNING: The actual manipulation of the device tree has to be the _last_ set of operations in board_fix_fdt! Since the pre-relocation driver model does not adapt to changes made to the device tree either, its references into the device tree will be invalid after manipulating it, and unpredictable behavior might occur when functions that rely on them are executed!

Hence, the recommended layout of the board_fixup_fdt call-back function is the following:

```
int board_fix_fdt(void *rw_fdt_blob)
{
        /*
         * Collect information about device's hardware and store
         * them in e.g. local variables
         */

        /* Do device tree manipulation using the values previously collected */

        /* Return 0 on successful manipulation and non-zero otherwise */
}
```

If this convention is kept, both an "additive" approach, meaning that nodes for detected components are added to the device tree, as well as a "subtractive" approach, meaning that nodes for absent components are removed from the tree, as well as a combination of both approaches should work.

Example

The controlcenterdc board (board/gdsys/a38x/controlcenterdc.c) features a board_fix_fdt function, in which six GPIO expanders (which might be present or not, since they are on daughter boards) on a I2C bus are queried for, and subsequently deactivated in the device tree if they are not present.

Note that the dm_i2c_simple_probe function does not use the device tree, hence it is safe to call it after the tree has already been manipulated.

Work to be done

- The application of device tree overlay should be possible in board_fixup_fdt, but has not been tested at this stage.

4.1.6 File System Firmware Loader

This is file system firmware loader for U-Boot framework, which has very close to some Linux Firmware API. For the details of Linux Firmware API, you can refer to https://01.org/linuxgraphics/gfx-docs/drm/driver-api/firmware/index.html.

File system firmware loader can be used to load whatever(firmware, image, and binary) from the storage device in file system format into target location such as memory, then consumer driver such as FPGA driver can program FPGA image from the target location into FPGA.

To enable firmware loader, CONFIG_FS_LOADER need to be set at <board_name>_defconfig such as "CONFIG_FS_LOADER=y".

Firmware Loader API core features

Firmware storage device described in device tree source

For passing data like storage device phandle and partition where the firmware loading from to the firmware loader driver, those data could be defined in fs-loader node as shown in below:

Example for block device:

```
fs_loader0: fs-loader {
        u-boot,dm-pre-reloc;
        compatible = "u-boot,fs-loader";
        phandlepart = <&mmc 1>;
};
```

<&mmc 1> means block storage device pointer and its partition.

Above example is a description for block storage, but for UBI storage device, it can be described in FDT as shown in below:

Example for ubi:

```
fs_loader1: fs-loader {
        u-boot,dm-pre-reloc;
        compatible = "u-boot,fs-loader";
        mtdpart = "UBI",
        ubivol = "ubi0";
};
```

Then, firmware-loader property can be added with any device node, which driver would use the firmware loader for loading.

The value of the firmware-loader property should be set with phandle of the fs-loader node. For example:

```
firmware-loader = <&fs_loader0>;
```

If there are majority of devices using the same fs-loader node, then firmware-loader property can be added under /chosen node instead of adding to each of device node.

For example:

```
/{
        chosen {
                firmware-loader = <&fs_loader0>;
        };
};
```

In each respective driver of devices using firmware loader, the firmware loaded instance should be created by DT phandle.

For example of getting DT phandle from /chosen and creating instance:

```
chosen_node = ofnode_path("/chosen");
if (!ofnode_valid(chosen_node)) {
        debug("/chosen node was not found.\n");
        return -ENOENT;
}

phandle_p = ofnode_get_property(chosen_node, "firmware-loader", &size);
if (!phandle_p) {
        debug("firmware-loader property was not found.\n");
```

(continues on next page)

(continued from previous page)

```
        return -ENOENT;
}

phandle = fdt32_to_cpu(*phandle_p);
ret = uclass_get_device_by_phandle_id(UCLASS_FS_FIRMWARE_LOADER,
                                     phandle, &dev);
if (ret)
        return ret;
```

Firmware loader driver is also designed to support U-boot environment variables, so all these data from FDT can be overwritten through the U-boot environment variable during run time.

For examples:

storage_interface: Storage interface, it can be “mmc”, “usb”, “sata” or “ubi”.

fw_dev_part: Block device number and its partition, it can be “0:1”.

fw_ubi_mtdpart: UBI device mtd partition, it can be “UBI”.

fw_ubi_volume: UBI volume, it can be “ubi0”.

When above environment variables are set, environment values would be used instead of data from FDT. The benefit of this design allows user to change storage attribute data at run time through U-boot console and saving the setting as default environment values in the storage for the next power cycle, so no compilation is required for both driver and FDT.

File system firmware Loader API

```
int request_firmware_into_buf(struct udevice *dev,
                              const char *name,
                              void *buf, size_t size, u32 offset)
```

Load firmware into a previously allocated buffer

Parameters:

- struct udevice *dev: An instance of a driver
- const char *name: name of firmware file
- void *buf: address of buffer to load firmware into
- size_t size: size of buffer
- u32 offset: offset of a file for start reading into buffer

Returns: size of total read -ve when error

Description: The firmware is loaded directly into the buffer pointed to by buf

Example of calling request_firmware_into_buf API after creating firmware loader instance:

```
ret = uclass_get_device_by_phandle_id(UCLASS_FS_FIRMWARE_LOADER,
                                     phandle, &dev);
if (ret)
        return ret;

request_firmware_into_buf(dev, filename, buffer_location, buffer_size,
                         offset_ofreading);
```

4.1.7 How to port an I2C driver to driver model

Over half of the I2C drivers have been converted as at November 2016. These ones remain:

- adi_i2c
- davinci_i2c
- fti2c010
- ihs_i2c
- kona_i2c
- lpc32xx_i2c
- pca9564_i2c
- ppc4xx_i2c
- rcar_i2c
- sh_i2c
- soft_i2c
- zynq_i2c

The deadline for this work is the end of June 2017. If no one steps forward to convert these, at some point there may come a patch to remove them!

Here is a suggested approach for converting your I2C driver over to driver model. Please feel free to update this file with your ideas and suggestions.

- #ifdef out all your own I2C driver code (#ifndef CONFIG_DM_I2C)
- Define CONFIG_DM_I2C for your board, vendor or architecture
- If the board does not already use driver model, you need CONFIG_DM also
- Your board should then build, but will not work fully since there will be no I2C driver
- Add the U_BOOT_DRIVER piece at the end (e.g. copy tegra_i2c.c for example)
- Add a private struct for the driver data - avoid using static variables
- Implement each of the driver methods, perhaps by calling your old methods
- You may need to adjust the function parameters so that the old and new implementations can share most of the existing code
- If you convert all existing users of the driver, remove the pre-driver-model code

In terms of patches a conversion series typically has these patches: - clean up / prepare the driver for conversion - add driver model code - convert at least one existing board to use driver model serial - (if no boards remain that don't use driver model) remove the old code

This may be a good time to move your board to use device tree also. Mostly this involves these steps:

- define CONFIG_OF_CONTROL and CONFIG_OF_SEPARATE
- add your device tree files to arch/<arch>/dts
- update the Makefile there
- Add stdout-path to your /chosen device tree node if it is not already there
- build and get u-boot-dtb.bin so you can test it
- Your drivers can now use device tree
- For device tree in SPL, define CONFIG_SPL_OF_CONTROL

4.1.8 Live Device Tree

Introduction

Traditionally U-Boot has used a ‘flat’ device tree. This means that it reads directly from the device tree binary structure. It is called a flat device tree because nodes are listed one after the other, with the hierarchy detected by tags in the format.

This document describes U-Boot’s support for a ‘live’ device tree, meaning that the tree is loaded into a hierarchical data structure within U-Boot.

Motivation

The flat device tree has several advantages:

- it is the format produced by the device tree compiler, so no translation is needed
- it is fairly compact (e.g. there is no need for pointers)
- it is accessed by the libfdt library, which is well tested and stable

However the flat device tree does have some limitations. Adding new properties can involve copying large amounts of data around to make room. The overall tree has a fixed maximum size so sometimes the tree must be rebuilt in a new location to create more space. Even if not adding new properties or nodes, scanning the tree can be slow. For example, finding the parent of a node is a slow process. Reading from nodes involves a small amount parsing which takes a little time.

Driver model scans the entire device tree sequentially on start-up which avoids the worst of the flat tree’s limitations. But if the tree is to be modified at run-time, a live tree is much faster. Even if no modification is necessary, parsing the tree once and using a live tree from then on seems to save a little time.

Implementation

In U-Boot a live device tree (‘livetree’) is currently supported only after relocation. Therefore we need a mechanism to specify a device tree node regardless of whether it is in the flat tree or livetree.

The ‘ofnode’ type provides this. An ofnode can point to either a flat tree node (when the live tree node is not yet set up) or a livetree node. The caller of an ofnode function does not need to worry about these details.

The main users of the information in a device tree are drivers. These have a ‘struct udevice *’ which is attached to a device tree node. Therefore it makes sense to be able to read device tree properties using the ‘struct udevice *’, rather than having to obtain the ofnode first.

The ‘dev_read_...()’ interface provides this. It allows properties to be easily read from the device tree using only a device pointer. Under the hood it uses ofnode so it works with both flat and live device trees.

Enabling livetree

CONFIG_OF_LIVE enables livetree. When this option is enabled, the flat tree will be used in SPL and before relocation in U-Boot proper. Just before relocation a livetree is built, and this is used for U-Boot proper after relocation.

Most checks for livetree use CONFIG_IS_ENABLED(OF_LIVE). This means that for SPL, the CONFIG_SPL_OF_LIVE option is checked. At present this does not exist, since SPL does not support livetree.

Porting drivers

Many existing drivers use the fdtdec interface to read device tree properties. This only works with a flat device tree. The drivers should be converted to use the dev_read_() interface.

For example, the old code may be like this:

```
struct udevice *bus;
const void *blob = gd->fdt_blob;
int node = dev_of_offset(bus);

i2c_bus->regs = (struct i2c_ctlr *)devfdt_get_addr(dev);
plat->frequency = fdtdec_get_int(blob, node, "spi-max-frequency", 500000);
```

The new code is:

```
struct udevice *bus;

i2c_bus->regs = (struct i2c_ctlr *)dev_read_addr(dev);
plat->frequency = dev_read_u32_default(bus, "spi-max-frequency", 500000);
```

The dev_read_...() interface is more convenient and works with both the flat and live device trees. See include/dm/read.h for a list of functions.

Where properties must be read from sub-nodes or other nodes, you must fall back to using ofnode. For example, for old code like this:

```
const void *blob = gd->fdt_blob;
int subnode;

fdt_for_each_subnode(subnode, blob, dev_of_offset(dev)) {
    freq = fdtdec_get_int(blob, node, "spi-max-frequency", 500000);
    ...
}
```

you should use:

```
ofnode subnode;

ofnode_for_each_subnode(subnode, dev_ofnode(dev)) {
    freq = ofnode_read_u32(node, "spi-max-frequency", 500000);
    ...
}
```

Useful ofnode functions

The internal data structures of the livetree are defined in include/dm/of.h :

struct device_node holds information about a device tree node

struct property holds information about a property within a node

Nodes have pointers to their first property, their parent, their first child and their sibling. This allows nodes to be linked together in a hierarchical tree.

Properties have pointers to the next property. This allows all properties of a node to be linked together in a chain.

It should not be necessary to use these data structures in normal code. In particular, you should refrain from using functions which access the livetree directly, such as of_read_u32(). Use ofnode functions instead, to allow your code to work with a flat tree also.

Some conversion functions are used internally. Generally these are not needed for driver code. Note that they will not work if called in the wrong context. For example it is invalid to call ofnode_to_no() when a flat tree is being used. Similarly it is not possible to call ofnode_to_offset() on a livetree node.

ofnode_to_np(): converts ofnode to struct device_node *

ofnode_to_offset(): converts ofnode to offset

no_to_ofnode(): converts node pointer to ofnode

offset_to_ofnode(): converts offset to ofnode

Other useful functions:

of_live_active(): returns true if livetree is in use, false if flat tree

ofnode_valid(): return true if a given node is valid

ofnode_is_np(): returns true if a given node is a livetree node

ofnode_equal(): compares two ofnodes

ofnode_null(): returns a null ofnode (for which ofnode_valid() returns false)

Phandles

There is full phandle support for live tree. All functions make use of struct ofnode_phandle_args, which has an ofnode within it. This supports both livetree and flat tree transparently. See for example ofnode_parse_phandle_with_args().

Reading addresses

You should use dev_read_addr() and friends to read addresses from device-tree nodes.

fdtdec

The existing fdtdec interface will eventually be retired. Please try to avoid using it in new code.

Modifying the livetree

This is not currently supported. Once implemented it should provide a much more efficient implementation for modification of the device tree than using the flat tree.

Internal implementation

The dev_read_…() functions have two implementations. When CONFIG_DM_DEV_READ_INLINE is enabled, these functions simply call the ofnode functions directly. This is useful when livetree is not enabled. The ofnode functions call ofnode_is_np(node) which will always return false if livetree is disabled, just falling back to flat tree code.

This optimisation means that without livetree enabled, the dev_read_…() and ofnode interfaces do not noticeably add to code size.

The CONFIG_DM_DEV_READ_INLINE option defaults to enabled when livetree is disabled.

Most livetree code comes directly from Linux and is modified as little as possible. This is deliberate since this code is fairly stable and does what we want. Some features (such as get/put) are not supported. Internal macros take care of removing these features silently.

Within the of_access.c file there are pointers to the alias node, the chosen node and the stdout-path alias.

Errors

With a flat device tree, libfdt errors are returned (e.g. -FDT_ERR_NOTFOUND). For livetree normal 'errno' errors are returned (e.g. -ENOTFOUND). At present the ofnode and dev_read_...() functions return either one or other type of error. This is clearly not desirable. Once tests are added for all the functions this can be tidied up.

Adding new access functions

Adding a new function for device-tree access involves the following steps:

- **Add two dev_read() functions:**
  - inline version in the read.h header file, which calls an ofnode function
  - standard version in the read.c file (or perhaps another file), which also calls an ofnode function

    The implementations of these functions can be the same. The purpose of the inline version is purely to reduce code size impact.
- Add an ofnode function. This should call ofnode_is_np() to work out whether a livetree or flat tree is used. For the livetree it should call an of_...() function. For the flat tree it should call an fdt_...() function. The livetree version will be optimised out at compile time if livetree is not enabled.
- Add an of_...() function for the livetree implementation. If a similar function is available in Linux, the implementation should be taken from there and modified as little as possible (generally not at all).

Future work

Live tree support was introduced in U-Boot 2017.07. There is still quite a bit of work to do to flesh this out:

- tests for all access functions
- support for livetree modification
- addition of more access functions as needed
- support for livetree in SPL and before relocation (if desired)

4.1.9 Migration Schedule

U-Boot has been migrating to a new driver model since its introduction in 2014. This file describes the schedule for deprecation of pre-driver-model features.

CONFIG_DM

- Status: In progress
- Deadline: 2020.01

Starting with the 2010.01 release CONFIG_DM will be enabled for all boards. This does not concern CONFIG_DM_SPL and CONFIG_DM_TPL. The conversion date for these configuration items still needs to be defined.

CONFIG_DM_MMC

- Status: In progress
- Deadline: 2019.04

The subsystem itself has been converted and maintainers should submit patches switching over to using CONFIG_DM_MMC and other base driver model options in time for inclusion in the 2019.04 rerelease.

CONFIG_DM_USB

- Status: In progress
- Deadline: 2019.07

The subsystem itself has been converted along with many of the host controller and maintainers should submit patches switching over to using CONFIG_DM_USB and other base driver model options in time for inclusion in the 2019.07 rerelease.

CONFIG_SATA

- Status: In progress
- Deadline: 2019.07

The subsystem itself has been converted along with many of the host controller and maintainers should submit patches switching over to using CONFIG_AHCI and other base driver model options in time for inclusion in the 2019.07 rerelease.

CONFIG_BLK

- Status: In progress
- Deadline: 2019.07

In concert with maintainers migrating their block device usage to the appropriate DM driver, CONFIG_BLK needs to be set as well. The final deadline here coincides with the final deadline for migration of the various block subsystems. At this point we will be able to audit and correct the logic in Kconfig around using CONFIG_PARTITIONS and CONFIG_HAVE_BLOCK_DEVICE and make use of CONFIG_BLK / CONFIG_SPL_BLK as needed.

CONFIG_DM_SPI / CONFIG_DM_SPI_FLASH

Board Maintainers should submit the patches for enabling DM_SPI and DM_SPI_FLASH to move the migration with in the deadline.

Partially converted:

```
drivers/spi/fsl_espi.c
drivers/spi/mxc_spi.c
drivers/spi/sh_qspi.c
```

- Status: In progress
- Deadline: 2019.07

CONFIG_DM_PCI

Deadline: 2019.07

The PCI subsystem has supported driver model since mid 2015. Maintainers should submit patches switching over to using CONFIG_DM_PCI and other base driver model options in time for inclusion in the 2019.07 release.

CONFIG_DM_VIDEO

Deadline: 2019.07

The video subsystem has supported driver model since early 2016. Maintainers should submit patches switching over to using CONFIG_DM_VIDEO and other base driver model options in time for inclusion in the 2019.07 release.

CONFIG_DM_ETH

Deadline: 2020.07

The network subsystem has supported the driver model since early 2015. Maintainers should submit patches switching over to using CONFIG_DM_ETH and other base driver model options in time for inclusion in the 2020.07 release.

4.1.10 Compiled-in Device Tree / Platform Data

Introduction

Device tree is the standard configuration method in U-Boot. It is used to define what devices are in the system and provide configuration information to these devices.

The overhead of adding device tree access to U-Boot is fairly modest, approximately 3KB on Thumb 2 (plus the size of the DT itself). This means that in most cases it is best to use device tree for configuration.

However there are some very constrained environments where U-Boot needs to work. These include SPL with severe memory limitations. For example, some SoCs require a 16KB SPL image which must include a full MMC stack. In this case the overhead of device tree access may be too great.

It is possible to create platform data manually by defining C structures for it, and reference that data in a U_BOOT_DEVICE() declaration. This bypasses the use of device tree completely, effectively creating a parallel configuration mechanism. But it is an available option for SPL.

As an alternative, a new 'of-platdata' feature is provided. This converts the device tree contents into C code which can be compiled into the SPL binary. This saves the 3KB of code overhead and perhaps a few hundred more bytes due to more efficient storage of the data.

Note: Quite a bit of thought has gone into the design of this feature. However it still has many rough edges and comments and suggestions are strongly encouraged! Quite possibly there is a much better approach.

Caveats

There are many problems with this features. It should only be used when strictly necessary. Notable problems include:

- Device tree does not describe data types. But the C code must define a type for each property. These are guessed using heuristics which are wrong in several fairly common cases. For example an 8-byte value is considered to be a 2-item integer array, and is byte-swapped. A boolean value that is not present means 'false', but cannot be included in the structures since there is generally no mention of it in the device tree file.

- Naming of nodes and properties is automatic. This means that they follow the naming in the device tree, which may result in C identifiers that look a bit strange.
- It is not possible to find a value given a property name. Code must use the associated C member variable directly in the code. This makes the code less robust in the face of device-tree changes. It also makes it very unlikely that your driver code will be useful for more than one SoC. Even if the code is common, each SoC will end up with a different C struct name, and a likely a different format for the platform data.
- The platform data is provided to drivers as a C structure. The driver must use the same structure to access the data. Since a driver normally also supports device tree it must use #ifdef to separate out this code, since the structures are only available in SPL.

How it works

The feature is enabled by CONFIG OF_PLATDATA. This is only available in SPL/TPL and should be tested with:

```
#if CONFIG_IS_ENABLED(OF_PLATDATA)
```

A new tool called 'dtoc' converts a device tree file either into a set of struct declarations, one for each compatible node, and a set of U_BOOT_DEVICE() declarations along with the actual platform data for each device. As an example, consider this MMC node:

```
sdmmc: dwmmc@ff0c0000 {
        compatible = "rockchip,rk3288-dw-mshc";
        clock-freq-min-max = <400000 150000000>;
        clocks = <&cru HCLK_SDMMC>, <&cru SCLK_SDMMC>,
                 <&cru SCLK_SDMMC_DRV>, <&cru SCLK_SDMMC_SAMPLE>;
        clock-names = "biu", "ciu", "ciu_drv", "ciu_sample";
        fifo-depth = <0x100>;
        interrupts = <GIC_SPI 32 IRQ_TYPE_LEVEL_HIGH>;
        reg = <0xff0c0000 0x4000>;
        bus-width = <4>;
        cap-mmc-highspeed;
        cap-sd-highspeed;
        card-detect-delay = <200>;
        disable-wp;
        num-slots = <1>;
        pinctrl-names = "default";
        pinctrl-0 = <&sdmmc_clk>, <&sdmmc_cmd>, <&sdmmc_cd>, <&sdmmc_bus4>;
            vmmc-supply = <&vcc_sd>;
            status = "okay";
            u-boot,dm-pre-reloc;
    };
```

Some of these properties are dropped by U-Boot under control of the CONFIG_OF_SPL_REMOVE_PROPS option. The rest are processed. This will produce the following C struct declaration:

```
struct dtd_rockchip_rk3288_dw_mshc {
        fdt32_t         bus_width;
        bool            cap_mmc_highspeed;
        bool            cap_sd_highspeed;
        fdt32_t         card_detect_delay;
        fdt32_t         clock_freq_min_max[2];
        struct phandle_1_arg clocks[4];
        bool            disable_wp;
        fdt32_t         fifo_depth;
```

(continues on next page)

(continued from previous page)

```
        fdt32_t         interrupts[3];
        fdt32_t         num_slots;
        fdt32_t         reg[2];
        fdt32_t         vmmc_supply;
};
```

and the following device declarations:

```
/* Node /clock-controller@ff760000 index 0 */
...

/* Node /dwmmc@ff0c0000 index 2 */
static struct dtd_rockchip_rk3288_dw_mshc dtv_dwmmc_at_ff0c0000 = {
        .fifo_depth             = 0x100,
        .cap_sd_highspeed       = true,
        .interrupts             = {0x0, 0x20, 0x4},
        .clock_freq_min_max     = {0x61a80, 0x8f0d180},
        .vmmc_supply            = 0xb,
        .num_slots              = 0x1,
        .clocks                 = {{0, 456},
                                   {0, 68},
                                   {0, 114},
                                   {0, 118}},
        .cap_mmc_highspeed      = true,
        .disable_wp             = true,
        .bus_width              = 0x4,
        .u_boot_dm_pre_reloc    = true,
        .reg                    = {0xff0c0000, 0x4000},
        .card_detect_delay      = 0xc8,
};

U_BOOT_DEVICE(dwmmc_at_ff0c0000) = {
        .name           = "rockchip_rk3288_dw_mshc",
        .platdata       = &dtv_dwmmc_at_ff0c0000,
        .platdata_size  = sizeof(dtv_dwmmc_at_ff0c0000),
        .parent_idx     = -1,
};

void dm_populate_phandle_data(void) {
}
```

The device is then instantiated at run-time and the platform data can be accessed using:

```
struct udevice *dev;
struct dtd_rockchip_rk3288_dw_mshc *plat = dev_get_platdata(dev);
```

This avoids the code overhead of converting the device tree data to platform data in the driver. The ofdata_to_platdata() method should therefore do nothing in such a driver.

Note that for the platform data to be matched with a driver, the 'name' property of the U_BOOT_DEVICE() declaration has to match a driver declared via U_BOOT_DRIVER(). This effectively means that a U_BOOT_DRIVER() with a 'name' corresponding to the devicetree 'compatible' string (after converting it to a valid name for C) is needed, so a dedicated driver is required for each 'compatible' string.

In order to make this a bit more flexible U_BOOT_DRIVER_ALIAS macro can be used to declare an alias for a driver name, typically a 'compatible' string. This macro produces no code, but it is by dtoc tool.

The parent_idx is the index of the parent driver_info structure within its linker list (instantiated by the U_BOOT_DEVICE() macro). This is used to support dev_get_parent(). The dm_populate_phandle_data()

is included to allow for fix-ups required by dtoc. It is not currently used. The values in 'clocks' are the index of the driver_info for the target device followed by any phandle arguments. This is used to support device_get_by_driver_info_idx().

During the build process dtoc parses both U_BOOT_DRIVER and U_BOOT_DRIVER_ALIAS to build a list of valid driver names and driver aliases. If the 'compatible' string used for a device does not not match a valid driver name, it will be checked against the list of driver aliases in order to get the right driver name to use. If in this step there is no match found a warning is issued to avoid run-time failures.

Where a node has multiple compatible strings, a #define is used to make them equivalent, e.g.:

```
#define dtd_rockchip_rk3299_dw_mshc dtd_rockchip_rk3288_dw_mshc
```

Converting of-platdata to a useful form

Of course it would be possible to use the of-platdata directly in your driver whenever configuration information is required. However this means that the driver will not be able to support device tree, since the of-platdata structure is not available when device tree is used. It would make no sense to use this structure if device tree were available, since the structure has all the limitations metioned in caveats above.

Therefore it is recommended that the of-platdata structure should be used only in the probe() method of your driver. It cannot be used in the ofdata_to_platdata() method since this is not called when platform data is already present.

How to structure your driver

Drivers should always support device tree as an option. The of-platdata feature is intended as a add-on to existing drivers.

Your driver should convert the platdata struct in its probe() method. The existing device tree decoding logic should be kept in the ofdata_to_platdata() method and wrapped with #if.

For example:

```
#include <dt-structs.h>

struct mmc_platdata {
#if CONFIG_IS_ENABLED(OF_PLATDATA)
        /* Put this first since driver model will copy the data here */
        struct dtd_mmc dtplat;
#endif
        /*
         * Other fields can go here, to be filled in by decoding from
         * the device tree (or the C structures when of-platdata is used).
         */
        int fifo_depth;
};

static int mmc_ofdata_to_platdata(struct udevice *dev)
{
#if !CONFIG_IS_ENABLED(OF_PLATDATA)
        /* Decode the device tree data */
        struct mmc_platdata *plat = dev_get_platdata(dev);
        const void *blob = gd->fdt_blob;
        int node = dev_of_offset(dev);

        plat->fifo_depth = fdtdec_get_int(blob, node, "fifo-depth", 0);
#endif
```

(continues on next page)

(continued from previous page)

```
        return 0;
}

static int mmc_probe(struct udevice *dev)
{
        struct mmc_platdata *plat = dev_get_platdata(dev);

#if CONFIG_IS_ENABLED(OF_PLATDATA)
        /* Decode the of-platdata from the C structures */
        struct dtd_mmc *dtplat = &plat->dtplat;

        plat->fifo_depth = dtplat->fifo_depth;
#endif
        /* Set up the device from the plat data */
        writel(plat->fifo_depth, ...)
}

static const struct udevice_id mmc_ids[] = {
        { .compatible = "vendor,mmc" },
        { }
};

U_BOOT_DRIVER(mmc_drv) = {
        .name           = "mmc_drv",
        .id             = UCLASS_MMC,
        .of_match       = mmc_ids,
        .ofdata_to_platdata = mmc_ofdata_to_platdata,
        .probe          = mmc_probe,
        .priv_auto_alloc_size = sizeof(struct mmc_priv),
        .platdata_auto_alloc_size = sizeof(struct mmc_platdata),
};

U_BOOT_DRIVER_ALIAS(mmc_drv, vendor_mmc) /* matches compatible string */
```

Note that struct mmc_platdata is defined in the C file, not in a header. This is to avoid needing to include dt-structs.h in a header file. The idea is to keep the use of each of-platdata struct to the smallest possible code area. There is just one driver C file for each struct, that can convert from the of-platdata struct to the standard one used by the driver.

In the case where SPL_OF_PLATDATA is enabled, platdata_auto_alloc_size is still used to allocate space for the platform data. This is different from the normal behaviour and is triggered by the use of of-platdata (strictly speaking it is a non-zero platdata_size which triggers this).

The of-platdata struct contents is copied from the C structure data to the start of the newly allocated area. In the case where device tree is used, the platform data is allocated, and starts zeroed. In this case the ofdata_to_platdata() method should still set up the platform data (and the of-platdata struct will not be present).

SPL must use either of-platdata or device tree. Drivers cannot use both at the same time, but they must support device tree. Supporting of-platdata is optional.

The device tree becomes in accessible when CONFIG_SPL_OF_PLATDATA is enabled, since the device-tree access code is not compiled in. A corollary is that a board can only move to using of-platdata if all the drivers it uses support it. There would be little point in having some drivers require the device tree data, since then libfdt would still be needed for those drivers and there would be no code-size benefit.

Internals

The dt-structs.h file includes the generated file (include/generated//dt-structs.h) if CONFIG_SPL_OF_PLATDATA is enabled. Otherwise (such as in U-Boot proper) these structs are not available. This prevents them being used inadvertently. All usage must be bracketed with #if CONFIG_IS_ENABLED(OF_PLATDATA).

The dt-platdata.c file contains the device declarations and is is built in spl/dt-platdata.c. It additionally contains the definition of dm_populate_phandle_data() which is responsible of filling the phandle information by adding references to U_BOOT_DEVICE by using DM_GET_DEVICE

The pylibfdt Python module is used to access the devicetree.

Credits

This is an implementation of an idea by Tom Rini <trini@konsulko.com>.

Future work

- Consider programmatically reading binding files instead of device tree contents

4.1.11 PCI with Driver Model

How busses are scanned

Any config read will end up at pci_read_config(). This uses uclass_get_device_by_seq() to get the PCI bus for a particular bus number. Bus number 0 will need to be requested first, and the alias in the device tree file will point to the correct device:

```
aliases {
        pci0 = &pcic;
};

pcic: pci@0 {
        compatible = "sandbox,pci";
        ...
};
```

If there is no alias the devices will be numbered sequentially in the device tree.

The call to uclass_get_device() will cause the PCI bus to be probed. This does a scan of the bus to locate available devices. These devices are bound to their appropriate driver if available. If there is no driver, then they are bound to a generic PCI driver which does nothing.

After probing a bus, the available devices will appear in the device tree under that bus.

Note that this is all done on a lazy basis, as needed, so until something is touched on PCI (eg: a call to pci_find_devices()) it will not be probed.

PCI devices can appear in the flattened device tree. If they do, their node often contains extra information which cannot be derived from the PCI IDs or PCI class of the device. Each PCI device node must have a <reg> property, as defined by the IEEE Std 1275-1994 PCI bus binding document v2.1. Compatible string list is optional and generally not needed, since PCI is discoverable bus, albeit there are justified exceptions. If the compatible string is present, matching on it takes precedence over PCI IDs and PCI classes.

Note we must describe PCI devices with the same bus hierarchy as the hardware, otherwise driver model cannot detect the correct parent/children relationship during PCI bus enumeration thus PCI devices won't be bound to their drivers accordingly. A working example like below:

```
pci {
        #address-cells = <3>;
        #size-cells = <2>;
        compatible = "pci-x86";
        u-boot,dm-pre-reloc;
        ranges = <0x02000000 0x0 0x40000000 0x40000000 0 0x80000000
                  0x42000000 0x0 0xc0000000 0xc0000000 0 0x20000000
                  0x01000000 0x0 0x2000 0x2000 0 0xe000>;

        pcie@17,0 {
                #address-cells = <3>;
                #size-cells = <2>;
                compatible = "pci-bridge";
                u-boot,dm-pre-reloc;
                reg = <0x0000b800 0x0 0x0 0x0 0x0>;

                topcliff@0,0 {
                        #address-cells = <3>;
                        #size-cells = <2>;
                        compatible = "pci-bridge";
                        u-boot,dm-pre-reloc;
                        reg = <0x00010000 0x0 0x0 0x0 0x0>;

                        pciuart0: uart@a,1 {
                                compatible = "pci8086,8811.00",
                                                "pci8086,8811",
                                                "pciclass,070002",
                                                "pciclass,0700",
                                                "x86-uart";
                                u-boot,dm-pre-reloc;
                                reg = <0x00025100 0x0 0x0 0x0 0x0
                                       0x01025110 0x0 0x0 0x0 0x0>;
                                ......
                        };

                        ......
                };
        };

        ......
};
```

In this example, the root PCI bus node is the "/pci" which matches "pci-x86" driver. It has a subnode "pcie@17,0" with driver "pci-bridge". "pcie@17,0" also has subnode "topcliff@0,0" which is a "pci-bridge" too. Under that bridge, a PCI UART device "uart@a,1" is described. This exactly reflects the hardware bus hierarchy: on the root PCI bus, there is a PCIe root port which connects to a downstream device Topcliff chipset. Inside Topcliff chipset, it has a PCIe-to-PCI bridge and all the chipset integrated devices like the PCI UART device are on the PCI bus. Like other devices in the device tree, if we want to bind PCI devices before relocation, "u-boot,dm-pre-reloc" must be declared in each of these nodes.

If PCI devices are not listed in the device tree, U_BOOT_PCI_DEVICE can be used to specify the driver to use for the device. The device tree takes precedence over U_BOOT_PCI_DEVICE. Please note with U_BOOT_PCI_DEVICE, only drivers with DM_FLAG_PRE_RELOC will be bound before relocation. If neither device tree nor U_BOOT_PCI_DEVICE is provided, the built-in driver (either pci_bridge_drv or pci_generic_drv) will be used.

Sandbox

With sandbox we need a device emulator for each device on the bus since there is no real PCI bus. This works by looking in the device tree node for an emulator driver. For example:

```
pci@1f,0 {
        compatible = "pci-generic";
        reg = <0xf800 0 0 0 0>;
        sandbox,emul = <&emul_1f>;
};
pci-emul {
        compatible = "sandbox,pci-emul-parent";
        emul_1f: emul@1f,0 {
                compatible = "sandbox,swap-case";
        };
};
```

This means that there is a 'sandbox,swap-case' driver at that bus position. Note that the first cell in the 'reg' value is the bus/device/function. See PCI_BDF() for the encoding (it is also specified in the IEEE Std 1275-1994 PCI bus binding document, v2.1)

The pci-emul node should go outside the pci bus node, since otherwise it will be scanned as a PCI device, causing confusion.

When this bus is scanned we will end up with something like this:

```
`- * pci@0 @ 05c660c8, 0
 `-   pci@1f,0 @ 05c661c8, 63488
`-   emul@1f,0 @ 05c662c8
```

When accesses go to the pci@1f,0 device they are forwarded to its emulator.

The sandbox PCI drivers also support dynamic driver binding, allowing device driver to declare the driver binding information via U_BOOT_PCI_DEVICE(), eliminating the need to provide any device tree node under the host controller node. It is required a "sandbox,dev-info" property must be provided in the host controller node for this functionality to work.

```
pci1: pci@1 {
        compatible = "sandbox,pci";
        ...
        sandbox,dev-info = <0x08 0x00 0x1234 0x5678
                            0x0c 0x00 0x1234 0x5678>;
};
```

The "sandbox,dev-info" property specifies all dynamic PCI devices on this bus. Each dynamic PCI device is encoded as 4 cells a group. The first and second cells are PCI device number and function number respectively. The third and fourth cells are PCI vendor ID and device ID respectively.

When this bus is scanned we will end up with something like this:

```
pci        [ + ]   pci_sandbo  |-- pci1
pci_emul   [   ]   sandbox_sw  |   |-- sandbox_swap_case_emul
pci_emul   [   ]   sandbox_sw  |   `-- sandbox_swap_case_emul
```

4.1.12 PMIC framework based on Driver Model

Introduction

This is an introduction to driver-model multi uclass PMIC IC's support. At present it's based on two uclass types:

UCLASS_PMIC: basic uclass type for PMIC I/O, which provides common read/write interface.

UCLASS_REGULATOR: additional uclass type for specific PMIC features, which are Voltage/Current regulators.

New files:

UCLASS_PMIC:

- drivers/power/pmic/pmic-uclass.c
- include/power/pmic.h

UCLASS_REGULATOR:

- drivers/power/regulator/regulator-uclass.c
- include/power/regulator.h

Commands: - common/cmd_pmic.c - common/cmd_regulator.c

How doees it work

The Power Management Integrated Circuits (PMIC) are used in embedded systems to provide stable, precise and specific voltage power source with over-voltage and thermal protection circuits.

The single PMIC can provide various functions by single or multiple interfaces, like in the example below:

```
-- SoC
 |
 |            ______________________________________
 | BUS 0     |       Multi interface PMIC IC        |--> LDO out 1
 | e.g.I2C0  |                                      |--> LDO out N
 |-----------|---- PMIC device 0 (READ/WRITE ops)   |
 | or SPI0   |    |_ REGULATOR device (ldo/... ops) |--> BUCK out 1
 |           |    |_ CHARGER device (charger ops)   |--> BUCK out M
 |           |    |_ MUIC device (microUSB con ops) |
 | BUS 1     |    |_ ...                            |---> BATTERY
 | e.g.I2C1  |                                      |
 |-----------|---- PMIC device 1 (READ/WRITE ops)   |---> USB in 1
 . or SPI1   |    |_ RTC device (rtc ops)           |---> USB in 2
 .           |______________________________________|---> USB out
 .
```

Since U-Boot provides driver model features for I2C and SPI bus drivers, the PMIC devices should also support this. By the pmic and regulator API's, PMIC drivers can simply provide a common functions, for multi-interface and and multi-instance device support.

Basic design assumptions:

- **Common I/O API:** UCLASS_PMIC. For the multi-function PMIC devices, this can be used as parent I/O device for each IC's interface. Then, each children uses the same dev for read/write.
- **Common regulator API:** UCLASS_REGULATOR. For driving the regulator attributes, auto setting function or command line interface, based on kernel-style regulator device tree constraints.

For simple implementations, regulator drivers are not required, so the code can use pmic read/write directly.

Pmic uclass

The basic information:

- Uclass: 'UCLASS_PMIC'

- Header: 'include/power/pmic.h'
- Core: 'drivers/power/pmic/pmic-uclass.c' (config 'CONFIG_DM_PMIC')
- Command: 'common/cmd_pmic.c' (config 'CONFIG_CMD_PMIC')
- Example: 'drivers/power/pmic/max77686.c'

For detailed API description, please refer to the header file.

As an example of the pmic driver, please refer to the MAX77686 driver.

Please pay attention for the driver's bind() method. Exactly the function call: 'pmic_bind_children()', which is used to bind the regulators by using the array of regulator's node, compatible prefixes.

The 'pmic; command also supports the new API. So the pmic command can be enabled by adding CONFIG_CMD_PMIC. The new pmic command allows to: - list pmic devices - choose the current device (like the mmc command) - read or write the pmic register - dump all pmic registers

This command can use only UCLASS_PMIC devices, since this uclass is designed for pmic I/O operations only.

For more information, please refer to the core file.

Regulator uclass

The basic information:

- Uclass: 'UCLASS_REGULATOR'
- Header: 'include/power/regulator.h'
- Core: 'drivers/power/regulator/regulator-uclass.c' (config 'CONFIG_DM_REGULATOR')
- Binding: 'doc/device-tree-bindings/regulator/regulator.txt'
- Command: 'common/cmd_regulator.c' (config 'CONFIG_CMD_REGULATOR')
- Example: 'drivers/power/regulator/max77686.c' 'drivers/power/pmic/max77686.c' (required I/O driver for the above)
- Example: 'drivers/power/regulator/fixed.c' (config 'CONFIG_DM_REGULATOR_FIXED')

For detailed API description, please refer to the header file.

For the example regulator driver, please refer to the MAX77686 regulator driver, but this driver can't operate without pmic's example driver, which provides an I/O interface for MAX77686 regulator.

The second example is a fixed Voltage/Current regulator for a common use.

The 'regulator' command also supports the new API. The command allow: - list regulator devices - choose the current device (like the mmc command) - do all regulator-specific operations

For more information, please refer to the command file.

4.1.13 Remote Processor Framework

Introduction

This is an introduction to driver-model for Remote Processors found on various System on Chip(SoCs). The term remote processor is used to indicate that this is not the processor on which U-Boot is operating on, instead is yet another processing entity that may be controlled by the processor on which we are functional.

The simplified model depends on a single UCLASS - UCLASS_REMOTEPROC

UCLASS_REMOTEPROC:

- drivers/remoteproc/rproc-uclass.c
- include/remoteproc.h

Commands:

- common/cmd_remoteproc.c

Configuration:

- CONFIG_REMOTEPROC is selected by drivers as needed
- CONFIG_CMD_REMOTEPROC for the commands if required.

How does it work - The driver

Overall, the driver statemachine transitions are typically as follows:

```
        (entry)
        +-------+
    +---+ init  |
    |   |       | <---------------------+
    |   +-------+                       |
    |                                   |
    |                                   |
    |   +--------+                      |
Load|   |  reset |                      |
    |   |        | <----------+         |
    |   +--------+            |         |
    |        |Load            |         |
    |        |                |         |
    |   +----v----+   reset   |         |
    +-> |         |    (opt)  |         |
        |  Loaded +-----------+         |
        |         |                     |
        +----+----+                     |
             | Start                    |
         +---v-----+        (opt)       |
      +->| Running |        Stop        |
Ping  +- |         +--------------------+
(opt)    +---------+
```

(is_running does not change state) opt: Optional state transition implemented by driver.

NOTE: It depends on the remote processor as to the exact behavior of the statemachine, remoteproc core does not intent to implement statemachine logic. Certain processors may allow start/stop without reloading the image in the middle, certain other processors may only allow us to start the processor(image from a EEPROM/OTP) etc.

It is hence the responsibility of the driver to handle the requisite state transitions of the device as necessary.

Basic design assumptions:

Remote processor can operate on a certain firmware that maybe loaded and released from reset.

The driver follows a standard UCLASS DM.

in the bare minimum form:

```
static const struct dm_rproc_ops sandbox_testproc_ops = {
        .load = sandbox_testproc_load,
        .start = sandbox_testproc_start,
```

(continues on next page)

(continued from previous page)

```
};

static const struct udevice_id sandbox_ids[] = {
        {.compatible = "sandbox,test-processor"},
        {}
};

U_BOOT_DRIVER(sandbox_testproc) = {
        .name = "sandbox_test_proc",
        .of_match = sandbox_ids,
        .id = UCLASS_REMOTEPROC,
        .ops = &sandbox_testproc_ops,
        .probe = sandbox_testproc_probe,
};
```

This allows for the device to be probed as part of the "init" command or invocation of 'rproc_init()' function as the system dependencies define.

The driver is expected to maintain it's own statemachine which is appropriate for the device it maintains. It must, at the very least provide a load and start function. We assume here that the device needs to be loaded and started, else, there is no real purpose of using the remoteproc framework.

Describing the device using platform data

IMPORTANT NOTE: THIS SUPPORT IS NOT MEANT FOR USE WITH NEWER PLATFORM SUPPORT. THIS IS ONLY FOR LEGACY DEVICES. THIS MODE OF INITIALIZATION *WILL* BE EVENTUALLY REMOVED ONCE ALL NECESSARY PLATFORMS HAVE MOVED TO DM/FDT.

Considering that many platforms are yet to move to device-tree model, a simplified definition of a device is as follows:

```
struct dm_rproc_uclass_pdata proc_3_test = {
        .name = "proc_3_legacy",
        .mem_type = RPROC_INTERNAL_MEMORY_MAPPED,
        .driver_plat_data = &mydriver_data;
};

U_BOOT_DEVICE(proc_3_demo) = {
        .name = "sandbox_test_proc",
        .platdata = &proc_3_test,
};
```

There can be additional data that may be desired depending on the remoteproc driver specific needs (for example: SoC integration details such as clock handle or something similar). See appropriate documentation for specific remoteproc driver for further details. These are passed via driver_plat_data.

Describing the device using device tree

aliases usage is optional, but it is usually recommended to ensure the users have a consistent usage model for a platform. the compatible string used here is specific to the remoteproc driver involved.

4.1.14 How to port a serial driver to driver model

Almost all of the serial drivers have been converted as at January 2016. These ones remain:

- serial_bfin.c

- serial_pxa.c

The deadline for this work was the end of January 2016. If no one steps forward to convert these, at some point there may come a patch to remove them!

Here is a suggested approach for converting your serial driver over to driver model. Please feel free to update this file with your ideas and suggestions.

- #ifdef out all your own serial driver code (#ifndef CONFIG_DM_SERIAL)
- Define CONFIG_DM_SERIAL for your board, vendor or architecture
- If the board does not already use driver model, you need CONFIG_DM also
- Your board should then build, but will not boot since there will be no serial driver
- Add the U_BOOT_DRIVER piece at the end (e.g. copy serial_s5p.c for example)
- Add a private struct for the driver data - avoid using static variables
- Implement each of the driver methods, perhaps by calling your old methods
- You may need to adjust the function parameters so that the old and new implementations can share most of the existing code
- If you convert all existing users of the driver, remove the pre-driver-model code

In terms of patches a conversion series typically has these patches: - clean up / prepare the driver for conversion - add driver model code - convert at least one existing board to use driver model serial - (if no boards remain that don't use driver model) remove the old code

This may be a good time to move your board to use device tree also. Mostly this involves these steps:

- define CONFIG_OF_CONTROL and CONFIG_OF_SEPARATE
- add your device tree files to arch/<arch>/dts
- update the Makefile there
- Add stdout-path to your /chosen device tree node if it is not already there
- build and get u-boot-dtb.bin so you can test it
- Your drivers can now use device tree
- For device tree in SPL, define CONFIG_SPL_OF_CONTROL

4.1.15 SOC ID Framework

Introduction

The driver-model SOC ID framework is able to provide identification information about a specific SoC in use at runtime, and also provide matching from a set of identification information from an array. This can be useful for enabling small quirks in drivers that exist between SoC variants that are impractical to implement using device tree flags. It is based on UCLASS_SOC.

UCLASS_SOC:

- drivers/soc/soc-uclass.c
- include/soc.h

Configuration:

- CONFIG_SOC_DEVICE is selected by drivers as needed.

Implementing a UCLASS_SOC provider

The purpose of this framework is to allow UCLASS_SOC provider drivers to supply identification information about the SoC in use at runtime. The framework allows drivers to define soc_ops that return identification strings. All soc_ops need not be defined and can be left as NULL, in which case the framework will return -ENOSYS and not consider the value when doing an soc_device_match.

It is left to the driver implementor to decide how the information returned is determined, but in general the same SOC should always return the same set of identifying information. Information returned must be in the form of a NULL terminated string.

See include/soc.h for documentation of the available soc_ops and the intended meaning of the values that can be returned. See drivers/soc/soc_sandbox.c for an example UCLASS_SOC provider driver.

Using a UCLASS_SOC driver

The framework provides the ability to retrieve and use the identification strings directly. It also has the ability to return a match from a list of different sets of SoC data using soc_device_match.

An array of 'struct soc_attr' can be defined, each containing ID information for a specific SoC, and when passed to soc_device_match, the identifier values for each entry in the list will be compared against the values provided by the UCLASS_SOC driver that is in use. The first entry in the list that matches all non-null values will be returned by soc_device_match.

An example of various uses of the framework can be found at test/dm/soc.c.

Describing the device using device tree

```
chipid: chipid {
     compatible = "sandbox,soc";
};
```

All that is required in a DT node is a compatible for a corresponding UCLASS_SOC driver.

4.1.16 How to port a SPI driver to driver model

Here is a rough step-by-step guide. It is based around converting the exynos SPI driver to driver model (DM) and the example code is based around U-Boot v2014.10-rc2 (commit be9f643). This has been updated for v2015.04.

It is quite long since it includes actual code examples.

Before driver model, SPI drivers have their own private structure which contains 'struct spi_slave'. With driver model, 'struct spi_slave' still exists, but now it is 'per-child data' for the SPI bus. Each child of the SPI bus is a SPI slave. The information that was stored in the driver-specific slave structure can now be port in private data for the SPI bus.

For example, struct tegra_spi_slave looks like this:

```
struct tegra_spi_slave {
        struct spi_slave slave;
        struct tegra_spi_ctrl *ctrl;
};
```

In this case 'slave' will be in per-child data, and 'ctrl' will be in the SPI's buses private data.

How long does this take?

You should be able to complete this within 2 hours, including testing but excluding preparing the patches. The API is basically the same as before with only minor changes:

- methods to set speed and mode are separated out
- cs_info is used to get information on a chip select

Enable driver mode for SPI and SPI flash

Add these to your board config:

- CONFIG_DM_SPI
- CONFIG_DM_SPI_FLASH

Add the skeleton

Put this code at the bottom of your existing driver file:

```
struct spi_slave *spi_setup_slave(unsigned int busnum, unsigned int cs,
                                  unsigned int max_hz, unsigned int mode)
{
        return NULL;
}

struct spi_slave *spi_setup_slave_fdt(const void *blob, int slave_node,
                                      int spi_node)
{
        return NULL;
}

static int exynos_spi_ofdata_to_platdata(struct udevice *dev)
{
        return -ENODEV;
}

static int exynos_spi_probe(struct udevice *dev)
{
        return -ENODEV;
}

static int exynos_spi_remove(struct udevice *dev)
{
        return -ENODEV;
}

static int exynos_spi_claim_bus(struct udevice *dev)
{

        return -ENODEV;
}

static int exynos_spi_release_bus(struct udevice *dev)
{

        return -ENODEV;
```

(continues on next page)

(continued from previous page)

```
}

static int exynos_spi_xfer(struct udevice *dev, unsigned int bitlen,
                           const void *dout, void *din, unsigned long flags)
{

        return -ENODEV;
}

static int exynos_spi_set_speed(struct udevice *dev, uint speed)
{
        return -ENODEV;
}

static int exynos_spi_set_mode(struct udevice *dev, uint mode)
{
        return -ENODEV;
}

static int exynos_cs_info(struct udevice *bus, uint cs,
                          struct spi_cs_info *info)
{
        return -EINVAL;
}

static const struct dm_spi_ops exynos_spi_ops = {
        .claim_bus      = exynos_spi_claim_bus,
        .release_bus    = exynos_spi_release_bus,
        .xfer           = exynos_spi_xfer,
        .set_speed      = exynos_spi_set_speed,
        .set_mode       = exynos_spi_set_mode,
        .cs_info        = exynos_cs_info,
};

static const struct udevice_id exynos_spi_ids[] = {
        { .compatible = "samsung,exynos-spi" },
        { }
};

U_BOOT_DRIVER(exynos_spi) = {
        .name   = "exynos_spi",
        .id     = UCLASS_SPI,
        .of_match = exynos_spi_ids,
        .ops    = &exynos_spi_ops,
        .ofdata_to_platdata = exynos_spi_ofdata_to_platdata,
        .probe  = exynos_spi_probe,
        .remove = exynos_spi_remove,
};
```

Replace 'exynos' in the above code with your driver name

#ifdef out all of the code in your driver except for the above

This will allow you to get it building, which means you can work incrementally. Since all the methods return an error initially, there is less chance that you will accidentally leave something in.

Also, even though your conversion is basically a rewrite, it might help reviewers if you leave functions in the same place in the file, particularly for large drivers.

Add some includes

Add these includes to your driver:

```
#include <dm.h>
#include <errno.h>
```

Build

At this point you should be able to build U-Boot for your board with the empty SPI driver. You still have empty methods in your driver, but we will write these one by one.

Set up your platform data structure

This will hold the information your driver to operate, like its hardware address or maximum frequency.

You may already have a struct like this, or you may need to create one from some of the #defines or global variables in the driver.

Note that this information is not the run-time information. It should not include state that changes. It should be fixed throughout the live of U-Boot. Run-time information comes later.

Here is what was in the exynos spi driver:

```
struct spi_bus {
        enum periph_id periph_id;
        s32 frequency;          /* Default clock frequency, -1 for none */
        struct exynos_spi *regs;
        int inited;             /* 1 if this bus is ready for use */
        int node;
        uint deactivate_delay_us;       /* Delay to wait after deactivate */
};
```

Of these, inited is handled by DM and node is the device tree node, which DM tells you. The name is not quite right. So in this case we would use:

```
struct exynos_spi_platdata {
        enum periph_id periph_id;
        s32 frequency;          /* Default clock frequency, -1 for none */
        struct exynos_spi *regs;
        uint deactivate_delay_us;       /* Delay to wait after deactivate */
};
```

Write ofdata_to_platdata() [for device tree only]

This method will convert information in the device tree node into a C structure in your driver (called platform data). If you are not using device tree, go to 8b.

DM will automatically allocate the struct for us when we are using device tree, but we need to tell it the size:

```
U_BOOT_DRIVER(spi_exynos) = {
...
        .platdata_auto_alloc_size = sizeof(struct exynos_spi_platdata),
```

Here is a sample function. It gets a pointer to the platform data and fills in the fields from device tree.

```
static int exynos_spi_ofdata_to_platdata(struct udevice *bus)
{
        struct exynos_spi_platdata *plat = bus->platdata;
        const void *blob = gd->fdt_blob;
        int node = dev_of_offset(bus);

        plat->regs = (struct exynos_spi *)fdtdec_get_addr(blob, node, "reg");
        plat->periph_id = pinmux_decode_periph_id(blob, node);

        if (plat->periph_id == PERIPH_ID_NONE) {
                debug("%s: Invalid peripheral ID %d\n", __func__,
                        plat->periph_id);
                return -FDT_ERR_NOTFOUND;
        }

        /* Use 500KHz as a suitable default */
        plat->frequency = fdtdec_get_int(blob, node, "spi-max-frequency",
                                        500000);
        plat->deactivate_delay_us = fdtdec_get_int(blob, node,
                                        "spi-deactivate-delay", 0);
        debug("%s: regs=%p, periph_id=%d, max-frequency=%d, deactivate_delay=%d\n",
              __func__, plat->regs, plat->periph_id, plat->frequency,
              plat->deactivate_delay_us);

        return 0;
}
```

Add the platform data [non-device-tree only]

Specify this data in a U_BOOT_DEVICE() declaration in your board file:

```
struct exynos_spi_platdata platdata_spi0 = {
        .periph_id = ...
        .frequency = ...
        .regs = ...
        .deactivate_delay_us = ...
};

U_BOOT_DEVICE(board_spi0) = {
        .name = "exynos_spi",
        .platdata = &platdata_spi0,
};
```

You will unfortunately need to put the struct definition into a header file in this case so that your board file can use it.

Add the device private data

Most devices have some private data which they use to keep track of things while active. This is the run-time information and needs to be stored in a structure. There is probably a structure in the driver that includes a 'struct spi_slave', so you can use that.

```
struct exynos_spi_slave {
        struct spi_slave slave;
```

(continues on next page)

(continued from previous page)

```
        struct exynos_spi *regs;
        unsigned int freq;              /* Default frequency */
        unsigned int mode;
        enum periph_id periph_id;       /* Peripheral ID for this device */
        unsigned int fifo_size;
        int skip_preamble;
        struct spi_bus *bus;            /* Pointer to our SPI bus info */
        ulong last_transaction_us;      /* Time of last transaction end */
};
```

We should rename this to make its purpose more obvious, and get rid of the slave structure, so we have:

```
struct exynos_spi_priv {
        struct exynos_spi *regs;
        unsigned int freq;              /* Default frequency */
        unsigned int mode;
        enum periph_id periph_id;       /* Peripheral ID for this device */
        unsigned int fifo_size;
        int skip_preamble;
        ulong last_transaction_us;      /* Time of last transaction end */
};
```

DM can auto-allocate this also:

```
U_BOOT_DRIVER(spi_exynos) = {
...
        .priv_auto_alloc_size = sizeof(struct exynos_spi_priv),
```

Note that this is created before the probe method is called, and destroyed after the remove method is called. It will be zeroed when the probe method is called.

Add the probe() and remove() methods

Note: It's a good idea to build repeatedly as you are working, to avoid a huge amount of work getting things compiling at the end.

The probe method is supposed to set up the hardware. U-Boot used to use spi_setup_slave() to do this. So take a look at this function and see what you can copy out to set things up.

```
static int exynos_spi_probe(struct udevice *bus)
{
        struct exynos_spi_platdata *plat = dev_get_platdata(bus);
        struct exynos_spi_priv *priv = dev_get_priv(bus);

        priv->regs = plat->regs;
        if (plat->periph_id == PERIPH_ID_SPI1 ||
            plat->periph_id == PERIPH_ID_SPI2)
                priv->fifo_size = 64;
        else
                priv->fifo_size = 256;

        priv->skip_preamble = 0;
        priv->last_transaction_us = timer_get_us();
        priv->freq = plat->frequency;
        priv->periph_id = plat->periph_id;

```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

This implementation doesn't actually touch the hardware, which is somewhat unusual for a driver. In this case we will do that when the device is claimed by something that wants to use the SPI bus.

For remove we could shut down the clocks, but in this case there is nothing to do. DM frees any memory that it allocated, so we can just remove exynos_spi_remove() and its reference in U_BOOT_DRIVER.

Implement set_speed()

This should set up clocks so that the SPI bus is running at the right speed. With the old API spi_claim_bus() would normally do this and several of the following functions, so let's look at that function:

```
int spi_claim_bus(struct spi_slave *slave)
{
        struct exynos_spi_slave *spi_slave = to_exynos_spi(slave);
        struct exynos_spi *regs = spi_slave->regs;
        u32 reg = 0;
        int ret;

        ret = set_spi_clk(spi_slave->periph_id,
                                        spi_slave->freq);
        if (ret < 0) {
                debug("%s: Failed to setup spi clock\n", __func__);
                return ret;
        }

        exynos_pinmux_config(spi_slave->periph_id, PINMUX_FLAG_NONE);

        spi_flush_fifo(slave);

        reg = readl(&regs->ch_cfg);
        reg &= ~(SPI_CH_CPHA_B | SPI_CH_CPOL_L);

        if (spi_slave->mode & SPI_CPHA)
                reg |= SPI_CH_CPHA_B;

        if (spi_slave->mode & SPI_CPOL)
                reg |= SPI_CH_CPOL_L;

        writel(reg, &regs->ch_cfg);
        writel(SPI_FB_DELAY_180, &regs->fb_clk);

        return 0;
}
```

It sets up the speed, mode, pinmux, feedback delay and clears the FIFOs. With DM these will happen in separate methods.

Here is an example for the speed part:

```
static int exynos_spi_set_speed(struct udevice *bus, uint speed)
{
        struct exynos_spi_platdata *plat = bus->platdata;
        struct exynos_spi_priv *priv = dev_get_priv(bus);
        int ret;
```

(continues on next page)

(continued from previous page)

```
        if (speed > plat->frequency)
                speed = plat->frequency;
        ret = set_spi_clk(priv->periph_id, speed);
        if (ret)
                return ret;
        priv->freq = speed;
        debug("%s: regs=%p, speed=%d\n", __func__, priv->regs, priv->freq);

        return 0;
}
```

Implement set_mode()

This should adjust the SPI mode (polarity, etc.). Again this code probably comes from the old spi_claim_bus(). Here is an example:

```
static int exynos_spi_set_mode(struct udevice *bus, uint mode)
{
        struct exynos_spi_priv *priv = dev_get_priv(bus);
        uint32_t reg;

        reg = readl(&priv->regs->ch_cfg);
        reg &= ~(SPI_CH_CPHA_B | SPI_CH_CPOL_L);

        if (mode & SPI_CPHA)
                reg |= SPI_CH_CPHA_B;

        if (mode & SPI_CPOL)
                reg |= SPI_CH_CPOL_L;

        writel(reg, &priv->regs->ch_cfg);
        priv->mode = mode;
        debug("%s: regs=%p, mode=%d\n", __func__, priv->regs, priv->mode);

        return 0;
}
```

Implement claim_bus()

This is where a client wants to make use of the bus, so claims it first. At this point we need to make sure everything is set up ready for data transfer. Note that this function is wholly internal to the driver - at present the SPI uclass never calls it.

Here again we look at the old claim function and see some code that is needed. It is anything unrelated to speed and mode:

```
static int exynos_spi_claim_bus(struct udevice *bus)
{
        struct exynos_spi_priv *priv = dev_get_priv(bus);

        exynos_pinmux_config(priv->periph_id, PINMUX_FLAG_NONE);
        spi_flush_fifo(priv->regs);

```

(continues on next page)

(continued from previous page)

```
        writel(SPI_FB_DELAY_180, &priv->regs->fb_clk);

        return 0;
}
```

The spi_flush_fifo() function is in the removed part of the code, so we need to expose it again (perhaps with an #endif before it and '#if 0' after it). It only needs access to priv->regs which is why we have passed that in:

```
/**
 * Flush spi tx, rx fifos and reset the SPI controller
 *
 * @param regs  Pointer to SPI registers
 */
static void spi_flush_fifo(struct exynos_spi *regs)
{
        clrsetbits_le32(&regs->ch_cfg, SPI_CH_HS_EN, SPI_CH_RST);
        clrbits_le32(&regs->ch_cfg, SPI_CH_RST);
        setbits_le32(&regs->ch_cfg, SPI_TX_CH_ON | SPI_RX_CH_ON);
}
```

Implement release_bus()

This releases the bus - in our example the old code in spi_release_bus() is a call to spi_flush_fifo, so we add:

```
static int exynos_spi_release_bus(struct udevice *bus)
{
        struct exynos_spi_priv *priv = dev_get_priv(bus);

        spi_flush_fifo(priv->regs);

        return 0;
}
```

Implement xfer()

This is the final method that we need to create, and it is where all the work happens. The method parameters are the same as the old spi_xfer() with the addition of a 'struct udevice' so conversion is pretty easy. Start by copying the contents of spi_xfer() to your new xfer() method and proceed from there.

If (flags & SPI_XFER_BEGIN) is non-zero then xfer() normally calls an activate function, something like this:

```
void spi_cs_activate(struct spi_slave *slave)
{
        struct exynos_spi_slave *spi_slave = to_exynos_spi(slave);

        /* If it's too soon to do another transaction, wait */
        if (spi_slave->bus->deactivate_delay_us &&
            spi_slave->last_transaction_us) {
                ulong delay_us;         /* The delay completed so far */
                delay_us = timer_get_us() - spi_slave->last_transaction_us;
                if (delay_us < spi_slave->bus->deactivate_delay_us)
                        udelay(spi_slave->bus->deactivate_delay_us - delay_us);
        }
```

(continues on next page)

(continued from previous page)

```
        clrbits_le32(&spi_slave->regs->cs_reg, SPI_SLAVE_SIG_INACT);
        debug("Activate CS, bus %d\n", spi_slave->slave.bus);
        spi_slave->skip_preamble = spi_slave->mode & SPI_PREAMBLE;
}
```

The new version looks like this:

```
static void spi_cs_activate(struct udevice *dev)
{
        struct udevice *bus = dev->parent;
        struct exynos_spi_platdata *pdata = dev_get_platdata(bus);
        struct exynos_spi_priv *priv = dev_get_priv(bus);

        /* If it's too soon to do another transaction, wait */
        if (pdata->deactivate_delay_us &&
            priv->last_transaction_us) {
                ulong delay_us;         /* The delay completed so far */
                delay_us = timer_get_us() - priv->last_transaction_us;
                if (delay_us < pdata->deactivate_delay_us)
                        udelay(pdata->deactivate_delay_us - delay_us);
        }

        clrbits_le32(&priv->regs->cs_reg, SPI_SLAVE_SIG_INACT);
        debug("Activate CS, bus '%s'\n", bus->name);
        priv->skip_preamble = priv->mode & SPI_PREAMBLE;
}
```

All we have really done here is change the pointers and print the device name instead of the bus number. Other local static functions can be treated in the same way.

Set up the per-child data and child pre-probe function

To minimise the pain and complexity of the SPI subsystem while the driver model change-over is in place, struct spi_slave is used to reference a SPI bus slave, even though that slave is actually a struct udevice. In fact struct spi_slave is the device's child data. We need to make sure this space is available. It is possible to allocate more space that struct spi_slave needs, but this is the minimum.

```
U_BOOT_DRIVER(exynos_spi) = {
...
        .per_child_auto_alloc_size      = sizeof(struct spi_slave),
}
```

Optional: Set up cs_info() if you want it

Sometimes it is useful to know whether a SPI chip select is valid, but this is not obvious from outside the driver. In this case you can provide a method for cs_info() to deal with this. If you don't provide it, then the device tree will be used to determine what chip selects are valid.

Return -EINVAL if the supplied chip select is invalid, or 0 if it is valid. If you don't provide the cs_info() method, 0 is assumed for all chip selects that do not appear in the device tree.

Test it

Now that you have the code written and it compiles, try testing it using the ‘sf test’ command. You may need to enable CONFIG_CMD_SF_TEST for your board.

Prepare patches and send them to the mailing lists

You can use ‘tools/patman/patman’ to prepare, check and send patches for your work. See tools/patman/README for details.

A little note about SPI uclass features

The SPI uclass keeps some information about each device ‘dev’ on the bus:

struct dm_spi_slave_platdata: This is device_get_parent_platdata(dev). This is where the chip select number is stored, along with the default bus speed and mode. It is automatically read from the device tree in spi_child_post_bind(). It must not be changed at run-time after being set up because platform data is supposed to be immutable at run-time.

struct spi_slave: This is device_get_parentdata(dev). Already mentioned above. It holds run-time information about the device.

There are also some SPI uclass methods that get called behind the scenes:

spi_post_bind(): Called when a new bus is bound. This scans the device tree for devices on the bus, and binds each one. This in turn causes spi_child_post_bind() to be called for each, which reads the device tree information into the parent (per-child) platform data.

spi_child_post_bind(): Called when a new child is bound. As mentioned above this reads the device tree information into the per-child platform data

spi_child_pre_probe(): Called before a new child is probed. This sets up the mode and speed in struct spi_slave by copying it from the parent’s platform data for this child. It also sets the ‘dev’ pointer, needed to permit passing ‘struct spi_slave’ around the place without needing a separate ‘struct udevice’ pointer.

The above housekeeping makes it easier to write your SPI driver.

4.1.17 How USB works with driver model

Introduction

Driver model USB support makes use of existing features but changes how drivers are found. This document provides some information intended to help understand how things work with USB in U-Boot when driver model is enabled.

Enabling driver model for USB

A new CONFIG_DM_USB option is provided to enable driver model for USB. This causes the USB uclass to be included, and drops the equivalent code in usb.c. In particular the usb_init() function is then implemented by the uclass.

Support for EHCI and XHCI

So far OHCI is not supported. Both EHCI and XHCI drivers should be declared as drivers in the USB uclass. For example:

```
static const struct udevice_id ehci_usb_ids[] = {
        { .compatible = "nvidia,tegra20-ehci", .data = USB_CTLR_T20 },
        { .compatible = "nvidia,tegra30-ehci", .data = USB_CTLR_T30 },
        { .compatible = "nvidia,tegra114-ehci", .data = USB_CTLR_T114 },
        { }
};

U_BOOT_DRIVER(usb_ehci) = {
        .name   = "ehci_tegra",
        .id     = UCLASS_USB,
        .of_match = ehci_usb_ids,
        .ofdata_to_platdata = ehci_usb_ofdata_to_platdata,
        .probe = tegra_ehci_usb_probe,
        .remove = tegra_ehci_usb_remove,
        .ops    = &ehci_usb_ops,
        .platdata_auto_alloc_size = sizeof(struct usb_platdata),
        .priv_auto_alloc_size = sizeof(struct fdt_usb),
        .flags  = DM_FLAG_ALLOC_PRIV_DMA,
};
```

Here ehci_usb_ids is used to list the controllers that the driver supports. Each has its own data value. Controllers must be in the UCLASS_USB uclass.

The ofdata_to_platdata() method allows the controller driver to grab any necessary settings from the device tree.

The ops here are ehci_usb_ops. All EHCI drivers will use these same ops in most cases, since they are all EHCI-compatible. For EHCI there are also some special operations that can be overridden when calling ehci_register().

The driver can use priv_auto_alloc_size to set the size of its private data. This can hold run-time information needed by the driver for operation. It exists when the device is probed (not when it is bound) and is removed when the driver is removed.

Note that usb_platdata is currently only used to deal with setting up a bus in USB device mode (OTG operation). It can be omitted if that is not supported.

The driver's probe() method should do the basic controller init and then call ehci_register() to register itself as an EHCI device. It should call ehci_deregister() in the remove() method. Registering a new EHCI device does not by itself cause the bus to be scanned.

The old ehci_hcd_init() function is no-longer used. Nor is it necessary to set up the USB controllers from board init code. When 'usb start' is used, each controller will be probed and its bus scanned.

XHCI works in a similar way.

Data structures

The following primary data structures are in use:

- **struct usb_device:** This holds information about a device on the bus. All devices have this structure, even the root hub. The controller itself does not have this structure. You can access it for a device 'dev' with dev_get_parent_priv(dev). It matches the old structure except that the parent and child information is not present (since driver model handles that). Once the device is set up, you can find the device descriptor and current configuration descriptor in this structure.
- **struct usb_platdata:** This holds platform data for a controller. So far this is only used as a work-around for controllers which can act as USB devices in OTG mode, since the gadget framework does not use driver model.
- **struct usb_dev_platdata:** This holds platform data for a device. You can access it for a device 'dev' with dev_get_parent_platdata(dev). It holds the device address and speed - anything that can

be determined before the device driver is actually set up. When probing the bus this structure is used to provide essential information to the device driver.

- **struct usb_bus_priv:** This is private information for each controller, maintained by the controller uclass. It is mostly used to keep track of the next device address to use.

Of these, only struct usb_device was used prior to driver model.

USB buses

Given a controller, you know the bus - it is the one attached to the controller. Each controller handles exactly one bus. Every controller has a root hub attached to it. This hub, which is itself a USB device, can provide one or more 'ports' to which additional devices can be attached. It is possible to power up a hub and find out which of its ports have devices attached.

Devices are given addresses starting at 1. The root hub is always address 1, and from there the devices are numbered in sequence. The USB uclass takes care of this numbering automatically during enumeration.

USB devices are enumerated by finding a device on a particular hub, and setting its address to the next available address. The USB bus stretches out in a tree structure, potentially with multiple hubs each with several ports and perhaps other hubs. Some hubs will have their own power since otherwise the 5V 500mA power supplied by the controller will not be sufficient to run very many devices.

Enumeration in U-Boot takes a long time since devices are probed one at a time, and each is given sufficient time to wake up and announce itself. The timeouts are set for the slowest device.

Up to 127 devices can be on each bus. USB has four bus speeds: low (1.5Mbps), full (12Mbps), high (480Mbps) which is only available with USB2 and newer (EHCI), and super (5Gbps) which is only available with USB3 and newer (XHCI). If you connect a super-speed device to a high-speed hub, you will only get high-speed.

USB operations

As before driver model, messages can be sent using submit_bulk_msg() and the like. These are now implemented by the USB uclass and route through the controller drivers. Note that messages are not sent to the driver of the device itself - i.e. they don't pass down the stack to the controller. U-Boot simply finds the controller to which the device is attached, and sends the message there with an appropriate 'pipe' value so it can be addressed properly. Having said that, the USB device which should receive the message is passed in to the driver methods, for use by sandbox. This design decision is open for review and the code impact of changing it is small since the methods are typically implemented by the EHCI and XHCI stacks.

Controller drivers (in UCLASS_USB) themselves provide methods for sending each message type. For XHCI an additional alloc_device() method is provided since XHCI needs to allocate a device context before it can even read the device's descriptor.

These methods use a 'pipe' which is a collection of bit fields used to describe the type of message, direction of transfer and the intended recipient (device number).

USB Devices

USB devices are found using a simple algorithm which works through the available hubs in a depth-first search. Devices can be in any uclass, but are attached to a parent hub (or controller in the case of the root hub) and so have parent data attached to them (this is struct usb_device).

By the time the device's probe() method is called, it is enumerated and is ready to talk to the host.

The enumeration process needs to work out which driver to attach to each USB device. It does this by examining the device class, interface class, vendor ID, product ID, etc. See struct usb_driver_entry for how drivers are matched with USB devices - you can use the USB_DEVICE() macro to declare a USB driver.

For example, usb_storage.c defines a USB_DEVICE() to handle storage devices, and it will be used for all USB devices which match.

Technical details on enumeration flow

It is useful to understand precisely how a USB bus is enumerating to avoid confusion when dealing with USB devices.

Device initialisation happens roughly like this:

- At some point the 'usb start' command is run
- This calls usb_init() which works through each controller in turn
- The controller is probed(). This does no enumeration.
- Then usb_scan_bus() is called. This calls usb_scan_device() to scan the (only) device that is attached to the controller - a root hub
- usb_scan_device() sets up a fake struct usb_device and calls usb_setup_device(), passing the port number to be scanned, in this case port 0
- usb_setup_device() first calls usb_prepare_device() to set the device address, then usb_select_config() to select the first configuration
- at this point the device is enumerated but we do not have a real struct udevice for it. But we do have the descriptor in struct usb_device so we can use this to figure out what driver to use
- back in usb_scan_device(), we call usb_find_child() to try to find an existing device which matches the one we just found on the bus. This can happen if the device is mentioned in the device tree, or if we previously scanned the bus and so the device was created before
- if usb_find_child() does not find an existing device, we call usb_find_and_bind_driver() which tries to bind one
- usb_find_and_bind_driver() searches all available USB drivers (declared with USB_DEVICE()). If it finds a match it binds that driver to create a new device.
- If it does not, it binds a generic driver. A generic driver is good enough to allow access to the device (sending it packets, etc.) but all functionality will need to be implemented outside the driver model.
- in any case, when usb_find_child() and/or usb_find_and_bind_driver() are done, we have a device with the correct uclass. At this point we want to probe the device
- first we store basic information about the new device (address, port, speed) in its parent platform data. We cannot store it its private data since that will not exist until the device is probed.
- then we call device_probe() which probes the device
- the first probe step is actually the USB controller's (or USB hubs's) child_pre_probe() method. This gets called before anything else and is intended to set up a child device ready to be used with its parent bus. For USB this calls usb_child_pre_probe() which grabs the information that was stored in the parent platform data and stores it in the parent private data (which is struct usb_device, a real one this time). It then calls usb_select_config() again to make sure that everything about the device is set up
- note that we have called usb_select_config() twice. This is inefficient but the alternative is to store additional information in the platform data. The time taken is minimal and this way is simpler
- at this point the device is set up and ready for use so far as the USB subsystem is concerned
- the device's probe() method is then called. It can send messages and do whatever else it wants to make the device work.

Note that the first device is always a root hub, and this must be scanned to find any devices. The above steps will have created a hub (UCLASS_USB_HUB), given it address 1 and set the configuration.

For hubs, the hub uclass has a post_probe() method. This means that after any hub is probed, the uclass gets to do some processing. In this case usb_hub_post_probe() is called, and the following steps take place:

- usb_hub_post_probe() calls usb_hub_scan() to scan the hub, which in turn calls usb_hub_configure()
- hub power is enabled
- we loop through each port on the hub, performing the same steps for each
- first, check if there is a device present. This happens in usb_hub_port_connect_change(). If so, then usb_scan_device() is called to scan the device, passing the appropriate port number.
- you will recognise usb_scan_device() from the steps above. It sets up the device ready for use. If it is a hub, it will scan that hub before it continues here (recursively, depth-first)
- once all hub ports are scanned in this way, the hub is ready for use and all of its downstream devices also
- additional controllers are scanned in the same way

The above method has some nice properties:

- the bus enumeration happens by virtue of driver model's natural device flow
- most logic is in the USB controller and hub uclasses; the actual device drivers do not need to know they are on a USB bus, at least so far as enumeration goes
- hub scanning happens automatically after a hub is probed

Hubs

USB hubs are scanned as in the section above. While hubs have their own uclass, they share some common elements with controllers:

- they both attach private data to their children (struct usb_device, accessible for a child with dev_get_parent_priv(child))
- they both use usb_child_pre_probe() to set up their children as proper USB devices

Example - Mass Storage

As an example of a USB device driver, see usb_storage.c. It uses its own uclass and declares itself as follows:

```
U_BOOT_DRIVER(usb_mass_storage) = {
        .name   = "usb_mass_storage",
        .id     = UCLASS_MASS_STORAGE,
        .of_match = usb_mass_storage_ids,
        .probe = usb_mass_storage_probe,
};

static const struct usb_device_id mass_storage_id_table[] = {
        { .match_flags = USB_DEVICE_ID_MATCH_INT_CLASS,
          .bInterfaceClass = USB_CLASS_MASS_STORAGE},
        { }     /* Terminating entry */
};

USB_DEVICE(usb_mass_storage, mass_storage_id_table);
```

The USB_DEVICE() macro attaches the given table of matching information to the given driver. Note that the driver is declared in U_BOOT_DRIVER() as 'usb_mass_storage' and this must match the first parameter of USB_DEVICE.

When usb_find_and_bind_driver() is called on a USB device with the bInterfaceClass value of USB_CLASS_MASS_STORAGE, it will automatically find this driver and use it.

Counter-example: USB Ethernet

As an example of the old way of doing things, see usb_ether.c. When the bus is scanned, all Ethernet devices will be created as generic USB devices (in uclass UCLASS_USB_DEV_GENERIC). Then, when the scan is completed, usb_host_eth_scan() will be called. This looks through all the devices on each bus and manually figures out which are Ethernet devices in the ways of yore.

In fact, usb_ether should be moved to driver model. Each USB Ethernet driver (e.g drivers/usb/eth/asix.c) should include a USB_DEVICE() declaration, so that it will be found as part of normal USB enumeration. Then, instead of a generic USB driver, a real (driver-model-aware) driver will be used. Since Ethernet now supports driver model, this should be fairly easy to achieve, and then usb_ether.c and the usb_host_eth_scan() will melt away.

Sandbox

All driver model uclasses must have tests and USB is no exception. To achieve this, a sandbox USB controller is provided. This can make use of emulation drivers which pretend to be USB devices. Emulations are provided for a hub and a flash stick. These are enough to create a pretend USB bus (defined by the sandbox device tree sandbox.dts) which can be scanned and used.

Tests in test/dm/usb.c make use of this feature. It allows much of the USB stack to be tested without real hardware being needed.

Here is an example device tree fragment:

```
usb@1 {
        compatible = "sandbox,usb";
        hub {
                compatible = "usb-hub";
                usb,device-class = <USB_CLASS_HUB>;
                hub-emul {
                        compatible = "sandbox,usb-hub";
                        #address-cells = <1>;
                        #size-cells = <0>;
                        flash-stick {
                                reg = <0>;
                                compatible = "sandbox,usb-flash";
                                sandbox,filepath = "flash.bin";
                        };
                };
        };
};
```

This defines a single controller, containing a root hub (which is required). The hub is emulated by a hub emulator, and the emulated hub has a single flash stick to emulate on one of its ports.

When ‘usb start’ is used, the following ‘dm tree’ output will be available:

```
usb         [ + ]    `-- usb@1
usb_hub     [ + ]        `-- hub
usb_emul    [ + ]            |-- hub-emul
usb_emul    [ + ]            |   `-- flash-stick
usb_mass_st [ + ]            `-- usb_mass_storage
```

This may look confusing. Most of it mirrors the device tree, but the ‘usb_mass_storage’ device is not in the device tree. This is created by usb_find_and_bind_driver() based on the USB_DRIVER in usb_storage.c.

While 'flash-stick' is the emulation device, 'usb_mass_storage' is the real U-Boot USB device driver that talks to it.

Future work

It is pretty uncommon to have a large USB bus with lots of hubs on an embedded system. In fact anything other than a root hub is uncommon. Still it would be possible to speed up enumeration in two ways:

- breadth-first search would allow devices to be reset and probed in parallel to some extent
- enumeration could be lazy, in the sense that we could enumerate just the root hub at first, then only progress to the next 'level' when a device is used that we cannot find. This could be made easier if the devices were statically declared in the device tree (which is acceptable for production boards where the same, known, things are on each bus).

But in common cases the current algorithm is sufficient.

Other things that need doing: - Convert usb_ether to use driver model as described above - Test that keyboards work (and convert to driver model) - Move the USB gadget framework to driver model - Implement OHCI in driver model - Implement USB PHYs in driver model - Work out a clever way to provide lazy init for USB devices

CHAPTER

FIVE

U-BOOT API DOCUMENTATION

These books get into the details of how specific U-Boot subsystems work from the point of view of a U-Boot developer. Much of the information here is taken directly from the U-Boot source, with supplemental material added as needed (or at least as we managed to add it - probably *not* all that is needed).

5.1 U-Boot API documentation

5.1.1 Device firmware update

void **set_dfu_alt_info**(char * *interface*, char * *devstr*)
set dfu_alt_info environment variable

Parameters

char * interface dfu interface, e.g. "mmc" or "nand"

char * devstr device number as string

Description

If CONFIG_SET_DFU_ALT_INFO=y, this board specific function is called to set environment variable dfu_alt_info.

int **dfu_alt_init**(int *num*, struct dfu_entity ** *dfu*)
initialize buffer for dfu entities

Parameters

int num number of entities

struct dfu_entity ** dfu on return allocated buffer

Return

0 on success

int **dfu_alt_add**(struct dfu_entity * *dfu*, char * *interface*, char * *devstr*, char * *s*)
add alternate to dfu entity buffer

Parameters

struct dfu_entity * dfu dfu entity

char * interface dfu interface, e.g. "mmc" or "nand"

char * devstr device number as string

char * s string description of alternate

Return

0 on success

int **dfu_config_entities**(char * *s*, char * *interface*, char * *devstr*)
initialize dfu entitities from envirionment

Parameters

char * s string with alternates

char * interface interface, e.g. "mmc" or "nand"

char * devstr device number as string

Description

Initialize the list of dfu entities from environment variable dfu_alt_info. The list must be freed by calling *dfu_free_entities()*. This function bypasses *set_dfu_alt_info()*. So typically you should use *dfu_init_env_entities()* instead.

See function *dfu_free_entities()* See function *dfu_init_env_entities()*

Return

0 on success, a negative error code otherwise

void **dfu_free_entities**(void)
free the list of dfu entities

Parameters

void no arguments

Description

Free the internal list of dfu entities.

See function *dfu_init_env_entities()*

void **dfu_show_entities**(void)
print DFU alt settings list

Parameters

void no arguments

int **dfu_get_alt_number**(void)
get number of alternates

Parameters

void no arguments

Return

number of alternates in the dfu entities list

const char * **dfu_get_dev_type**(enum dfu_device_type *type*)
get string representation for dfu device type

Parameters

enum dfu_device_type type device type

Return

string representation for device type

const char * **dfu_get_layout**(enum dfu_layout *layout*)
get string describing layout

Parameters

enum dfu_layout layout layout Result: string representation for the layout

Description

Internally layouts are represented by enum dfu_device_type values. This function translates an enum value to a human readable string, e.g. DFU_FS_FAT is translated to "FAT".

struct dfu_entity * **dfu_get_entity**(int *alt*)
get dfu entity for an alternate id

Parameters

int alt alternate id

Return

dfu entity

int **dfu_get_alt**(char * *name*)
get alternate id for filename

Parameters

char * name filename

Description

Environment variable dfu_alt_info defines the write destinations (alternates) for different filenames. This function get the index of the alternate for a filename. If an absolute filename is provided (starting with '/'), the directory path is ignored.

Return

id of the alternate or negative error number (-ENODEV)

int **dfu_init_env_entities**(char * *interface*, char * *devstr*)
initialize dfu entitities from envirionment

Parameters

char * interface interface, e.g. "mmc" or "nand"

char * devstr device number as string

Description

Initialize the list of dfu entities from environment variable dfu_alt_info. The list must be freed by calling *dfu_free_entities()*. **interface** and **devstr** are used to select the relevant set of alternates from environment variable dfu_alt_info.

If environment variable dfu_alt_info specifies the interface and the device, use NULL for **interface** and **devstr**.

See function *dfu_free_entities()*

Return

0 on success, a negative error code otherwise

int **dfu_read**(struct dfu_entity * *de*, void * *buf*, int *size*, int *blk_seq_num*)
read from dfu entity

Parameters

struct dfu_entity * de dfu entity

void * buf buffer

int size size of buffer

int blk_seq_num block sequence number

Description

The block sequence number **blk_seq_num** is a 16 bit counter that must be incremented with each call for the same dfu entity **de**.

Return

0 for success, -1 for error

int **dfu_write**(struct dfu_entity * *de*, void * *buf*, int *size*, int *blk_seq_num*)
write to dfu entity

Parameters

struct dfu_entity * de dfu entity

void * buf buffer

int size size of buffer

int blk_seq_num block sequence number

Description

Write the contents of a buffer **buf** to the dfu entity **de**. After writing the last block call *dfu_flush()*. If a file is already loaded completely into memory it is preferable to use *dfu_write_from_mem_addr()* which takes care of blockwise transfer and flushing.

The block sequence number **blk_seq_num** is a 16 bit counter that must be incremented with each call for the same dfu entity **de**.

See function *dfu_flush()* See function *dfu_write_from_mem_addr()*

Return

0 for success, -1 for error

int **dfu_flush**(struct dfu_entity * *de*, void * *buf*, int *size*, int *blk_seq_num*)
flush to dfu entity

Parameters

struct dfu_entity * de dfu entity

void * buf ignored

int size ignored

int blk_seq_num block sequence number of last write - ignored

Description

This function has to be called after writing the last block to the dfu entity **de**.

The block sequence number **blk_seq_num** is a 16 bit counter that must be incremented with each call for the same dfu entity **de**.

See function *dfu_write()*

Return

0 for success, -1 for error

void **dfu_initiated_callback**(struct dfu_entity * *dfu*)
weak callback called on DFU transaction start

Parameters

struct dfu_entity * dfu pointer to the dfu_entity, which should be initialized

Description

It is a callback function called by DFU stack when a DFU transaction is initiated. This function allows to manage some board specific behavior on DFU targets.

void **dfu_flush_callback**(struct dfu_entity * *dfu*)
weak callback called at the end of the DFU write

Parameters

struct dfu_entity * dfu pointer to the dfu_entity, which should be flushed

Description

It is a callback function called by DFU stack after DFU manifestation. This function allows to manage some board specific behavior on DFU targets

struct dfu_entity * **dfu_get_defer_flush**(void)
get current value of dfu_defer_flush pointer

Parameters

void no arguments

Return

value of the dfu_defer_flush pointer

void **dfu_set_defer_flush**(struct dfu_entity * *dfu*)
set the dfu_defer_flush pointer

Parameters

struct dfu_entity * dfu pointer to the dfu_entity, which should be written

int **dfu_write_from_mem_addr**(struct dfu_entity * *dfu*, void * *buf*, int *size*)
write data from memory to DFU managed medium

Parameters

struct dfu_entity * dfu dfu entity to which we want to store data

void * buf fixed memory address from where data starts

int size number of bytes to write

Description

This function adds support for writing data starting from fixed memory address (like $loadaddr) to dfu managed medium (e.g. NAND, MMC, file system)

Return

0 on success, other value on failure

int **dfu_write_by_name**(char * *dfu_entity_name*, void * *addr*, unsigned int *len*, char * *interface*, char * *devstring*)
write data to DFU medium

Parameters

char * dfu_entity_name Name of DFU entity to write

void * addr Address of data buffer to write

unsigned int len Number of bytes

char * interface Destination DFU medium (e.g. “mmc”)

char * devstring Instance number of destination DFU medium (e.g. “1”)

Description

This function is storing data received on DFU supported medium which is specified by **dfu_entity_name**.

Return

0 - on success, error code - otherwise

int **dfu_write_by_alt**(int *dfu_alt_num*, void * *addr*, unsigned int *len*, char * *interface*, char * *devstring*)
write data to DFU medium

Parameters

int dfu_alt_num DFU alt setting number

void * addr Address of data buffer to write

unsigned int len Number of bytes

char * interface Destination DFU medium (e.g. "mmc")

char * devstring Instance number of destination DFU medium (e.g. "1")

Description

This function is storing data received on DFU supported medium which is specified by **dfu_alt_name**.

Return

0 - on success, error code - otherwise

5.1.2 UEFI subsystem

Lauching UEFI images

Bootefi command

The bootefi command is used to start UEFI applications or to install UEFI drivers. It takes two parameters

bootefi <image address> [fdt address]

- image address - the memory address of the UEFI binary
- fdt address - the memory address of the flattened device tree

The environment variable 'bootargs' is passed as load options in the UEFI system table. The Linux kernel EFI stub uses the load options as command line arguments.

efi_status_t **efi_env_set_load_options**(efi_handle_t *handle*, const char * *env_var*, u16 ** *load_options*)

set load options from environment variable

Parameters

efi_handle_t handle the image handle

const char * env_var name of the environment variable

u16 ** load_options pointer to load options (output)

Return

status code

efi_status_t **copy_fdt**(void ** *fdtp*)

Copy the device tree to a new location available to EFI

Parameters

void ** fdtp On entry a pointer to the flattened device tree. On exit a pointer to the copy of the flattened device tree. FDT start

Description

The FDT is copied to a suitable location within the EFI memory map. Additional 12 KiB are added to the space in case the device tree needs to be expanded later with `fdt_open_into()`.

Return

status code

void **efi_reserve_memory**(u64 *addr*, u64 *size*, bool *nomap*)

add reserved memory to memory map

Parameters

u64 addr start address of the reserved memory range

u64 size size of the reserved memory range

bool nomap indicates that the memory range shall not be accessed by the UEFI payload

void **efi_carve_out_dt_rsv**(void * *fdt*)
Carve out DT reserved memory ranges

Parameters

void * fdt Pointer to device tree

Description

The mem_rsv entries of the FDT are added to the memory map. Any failures are ignored because this is not critical and we would rather continue to try to boot.

void * **get_config_table**(const efi_guid_t * *guid*)
get configuration table

Parameters

const efi_guid_t * guid GUID of the configuration table

Return

pointer to configuration table or NULL

efi_status_t **efi_install_fdt**(void * *fdt*)
install device tree

Parameters

void * fdt address of device tree or EFI_FDT_USE_INTERNAL to use the the hardware device tree as indicated by environment variable fdt_addr or as fallback the internal device tree as indicated by the environment variable fdtcontroladdr

Description

If fdt is not EFI_FDT_USE_INTERNAL, the device tree located at that memory address will will be installed as configuration table, otherwise the device tree located at the address indicated by environment variable fdt_addr or as fallback fdtcontroladdr will be used.

On architectures using ACPI tables device trees shall not be installed as configuration table.

Return

status code

efi_status_t **do_bootefi_exec**(efi_handle_t *handle*, void * *load_options*)
execute EFI binary

Parameters

efi_handle_t handle handle of loaded image

void * load_options load options

Description

The image indicated by **handle** is started. When it returns the allocated memory for the **load_options** is freed.

Return

status code

Load the EFI binary into a newly assigned memory unwinding the relocation information, install the loaded image protocol, and call the binary.

int **do_efibootmgr**(void)
execute EFI boot manager

Parameters

void no arguments

Return

status code

int **do_bootefi_image**(const char * *image_opt*)
execute EFI binary

Parameters

const char * image_opt string of image start address

Description

Set up memory image for the binary to be loaded, prepare device path, and then call *do_bootefi_exec()* to execute it.

Return

status code

efi_status_t **efi_run_image**(void * *source_buffer*, efi_uintn_t *source_size*)
run loaded UEFI image

Parameters

void * source_buffer memory address of the UEFI image

efi_uintn_t source_size size of the UEFI image

Return

status code

efi_status_t **bootefi_test_prepare**(struct efi_loaded_image_obj ** *image_objp*, struct efi_loaded_image ** *loaded_image_infop*, const char * *path*, const char * *load_options_path*)
prepare to run an EFI test

Parameters

struct efi_loaded_image_obj ** image_objp pointer to be set to the loaded image handle

struct efi_loaded_image ** loaded_image_infop pointer to be set to the loaded image protocol

const char * path dummy file path used to construct the device path set in the loaded image protocol

const char * load_options_path name of a U-Boot environment variable. Its value is set as load options in the loaded image protocol.

Description

Prepare to run a test as if it were provided by a loaded image.

Return

status code

void **bootefi_run_finish**(struct efi_loaded_image_obj * *image_obj*, struct efi_loaded_image * *loaded_image_info*)
finish up after running an EFI test

Parameters

struct efi_loaded_image_obj * image_obj Pointer to a struct which holds the loaded image object

struct efi_loaded_image * loaded_image_info Pointer to a struct which holds the loaded image info

int **do_efi_selftest**(void)
execute EFI selftest

Parameters

void no arguments

Return

status code

int **do_bootefi**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
execute *bootefi* command

Parameters

struct cmd_tbl * cmdtp table entry describing command

int flag bitmap indicating how the command was invoked

int argc number of arguments

char *const argv command line arguments

Return

status code

void **efi_set_bootdev**(const char * *dev*, const char * *devnr*, const char * *path*)
set boot device

Parameters

const char * dev device, e.g. "MMC"

const char * devnr number of the device, e.g. "1:2"

const char * path path to file loaded

Description

This function is called when a file is loaded, e.g. via the 'load' command. We use the path to this file to inform the UEFI binary about the boot device.

Boot manager

The UEFI specification foresees to define boot entries and boot sequence via UEFI variables. Booting according to these variables is possible via

bootefi bootmgr [fdt address]

- fdt address - the memory address of the flattened device tree

The relevant variables are:

- Boot0000-BootFFFF define boot entries
- BootNext specifies next boot option to be booted
- BootOrder specifies in which sequence the boot options shall be tried if BootNext is not defined or booting via BootNext fails

efi_status_t **efi_set_load_options**(efi_handle_t *handle*, efi_uintn_t *load_options_size*, void * *load_options*)
set the load options of a loaded image

Parameters

efi_handle_t handle the image handle

efi_uintn_t load_options_size size of load options

void * load_options pointer to load options

Return

status code

efi_status_t **efi_deserialize_load_option**(struct efi_load_option * *lo*, u8 * *data*, efi_uintn_t * *size*)
parse serialized data

Parameters

struct efi_load_option * lo pointer to target

u8 * data serialized data

efi_uintn_t * size size of the load option, on return size of the optional data

Description

Parse serialized data describing a load option and transform it to the efi_load_option structure.

Return

status code

unsigned long **efi_serialize_load_option**(struct efi_load_option * *lo*, u8 ** *data*)
serialize load option

Parameters

struct efi_load_option * lo load option

u8 ** data buffer for serialized data

Description

Serialize efi_load_option structure into byte stream for BootXXXX.

Return

size of allocated buffer

void * **get_var**(u16 * *name*, const efi_guid_t * *vendor*, efi_uintn_t * *size*)
get UEFI variable

Parameters

u16 * name name of variable

const efi_guid_t * vendor vendor GUID of variable

efi_uintn_t * size size of allocated buffer

Description

It is the caller's duty to free the returned buffer.

Return

buffer with variable data or NULL

efi_status_t **try_load_entry**(u16 *n*, efi_handle_t * *handle*, void ** *load_options*)
try to load image for boot option

Parameters

u16 n number of the boot option, e.g. 0x0a13 for Boot0A13

efi_handle_t * handle on return handle for the newly installed image

void ** load_options load options set on the loaded image protocol

Description

Attempt to load load-option number 'n', returning device_path and file_path if successful. This checks that the EFI_LOAD_OPTION is active (enabled) and that the specified file to boot exists.

Return

status code

efi_status_t **efi_bootmgr_load**(efi_handle_t * *handle*, void ** *load_options*)
try to load from BootNext or BootOrder

Parameters

efi_handle_t * handle on return handle for the newly installed image

void ** load_options load options set on the loaded image protocol

Description

Attempt to load from BootNext or in the order specified by BootOrder EFI variable, the available load-options, finding and returning the first one that can be loaded successfully.

Return

status code

Efidebug command

The efidebug command is used to set and display boot options as well as to display information about internal data of the UEFI subsystem (devices, drivers, handles, loaded images, and the memory map).

int **efi_get_device_handle_info**(efi_handle_t *handle*, u16 ** *dev_path_text*)
get information of UEFI device

Parameters

efi_handle_t handle Handle of UEFI device

u16 ** dev_path_text Pointer to text of device path

Return

0 on success, -1 on failure

Currently return a formatted text of device path.

int **do_efi_show_devices**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
show UEFI devices

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug "devices" sub-command. Show all UEFI devices and their information.

int **efi_get_driver_handle_info**(efi_handle_t *handle*, u16 ** *driver_name*, u16 ** *image_path*)
get information of UEFI driver

Parameters

efi_handle_t handle Handle of UEFI device

u16 ** driver_name Driver name

u16 ** image_path Pointer to text of device path

Return

0 on success, -1 on failure

Currently return no useful information as all UEFI drivers are built-in..

int **do_efi_show_drivers**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
show UEFI drivers

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug "drivers" sub-command. Show all UEFI drivers and their information.

const char * **get_guid_text**(const void * *guid*)
get string of GUID

Parameters

const void * guid GUID

Description

Return description of GUID.

Return

description of GUID or NULL

int **do_efi_show_handles**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
show UEFI handles

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug "dh" sub-command. Show all UEFI handles and their information, currently all protocols added to handle.

int **do_efi_show_images**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
show UEFI images

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug “images” sub-command. Show all UEFI loaded images and their information.

void **print_memory_attributes**(u64 *attributes*)
print memory map attributes

Parameters

u64 attributes Attribute value

Description

Print memory map attributes

int **do_efi_show_memmap**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
show UEFI memory map

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug “memmap” sub-command. Show UEFI memory map.

int **do_efi_show_tables**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
show UEFI configuration tables

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug “tables” sub-command. Show UEFI configuration tables.

int **do_efi_boot_add**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
set UEFI load option

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_USAGE or CMD_RET_RET_FAILURE on failure

Implement efidebug “boot add” sub-command. Create or change UEFI load option.

efidebug boot add <id> <label> <interface> <devnum>[:<part>] <file> <options>

int **do_efi_boot_rm**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
delete UEFI load options

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug "boot rm" sub-command. Delete UEFI load options.

efidebug boot rm <id> ...

void **show_efi_boot_opt_data**(u16 * *varname16*, void * *data*, size_t * *size*)
dump UEFI load option

Parameters

u16 * varname16 variable name

void * data value of UEFI load option variable

size_t * size size of the boot option

Description

Decode the value of UEFI load option variable and print information.

void **show_efi_boot_opt**(u16 * *varname16*)
dump UEFI load option

Parameters

u16 * varname16 variable name

Description

Dump information defined by UEFI load option.

int **do_efi_boot_dump**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
dump all UEFI load options

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug "boot dump" sub-command. Dump information of all UEFI load options defined.

efidebug boot dump

int **show_efi_boot_order**(void)
show order of UEFI load options

Parameters

void no arguments

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Show order of UEFI load options defined by BootOrder variable.

int **do_efi_boot_next**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
manage UEFI BootNext variable

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_USAGE or CMD_RET_RET_FAILURE on failure

Implement efidebug "boot next" sub-command. Set BootNext variable.

efidebug boot next <id>

int **do_efi_boot_order**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
manage UEFI BootOrder variable

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_RET_FAILURE on failure

Implement efidebug "boot order" sub-command. Show order of UEFI load options, or change it in BootOrder variable.

efidebug boot order [<id> …]

int **do_efi_boot_opt**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
manage UEFI load options

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_USAGE or CMD_RET_RET_FAILURE on failure

Implement efidebug "boot" sub-command.

int **do_efi_test_bootmgr**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
run simple bootmgr for test

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_USAGE or CMD_RET_RET_FAILURE on failure

Implement efidebug "test bootmgr" sub-command. Run simple bootmgr for test.

efidebug test bootmgr

int **do_efi_test**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
manage UEFI load options

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_USAGE or CMD_RET_RET_FAILURE on failure

Implement efidebug "test" sub-command.

int **do_efi_query_info**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
QueryVariableInfo EFI service

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_USAGE or CMD_RET_FAILURE on failure

Implement efidebug "test" sub-command.

int **do_efidebug**(struct cmd_tbl * *cmdtp*, int *flag*, int *argc*, char *const *argv*)
display and configure UEFI environment

Parameters

struct cmd_tbl * cmdtp Command table

int flag Command flag

int argc Number of arguments

char *const argv Argument array

Return

CMD_RET_SUCCESS on success, CMD_RET_USAGE or CMD_RET_RET_FAILURE on failure

Implement efidebug command which allows us to display and configure UEFI environment.

Initialization of the UEFI sub-system

efi_status_t **efi_init_platform_lang**(void)
define supported languages

Parameters

void no arguments

Description

Set the PlatformLangCodes and PlatformLang variables.

Return

status code

efi_status_t **efi_init_secure_boot**(void)
initialize secure boot state

Parameters

void no arguments

Return

status code

efi_status_t **efi_init_obj_list**(void)
Initialize and populate EFI object list

Parameters

void no arguments

Return

status code

Boot services

void **efi_save_gd**(void)
save global data register

Parameters

void no arguments

Description

On the ARM and RISC-V architectures gd is mapped to a fixed register. As this register may be overwritten by an EFI payload we save it here and restore it on every callback entered.

This function is called after relocation from `initr_reloc_global_data()`.

void **efi_restore_gd**(void)
restore global data register

Parameters

void no arguments

Description

On the ARM and RISC-V architectures gd is mapped to a fixed register. Restore it after returning from the UEFI world to the value saved via *efi_save_gd()*.

const char * **indent_string**(int *level*)
returns a string for indenting with two spaces per level

Parameters

int level indent level

Description

A maximum of ten indent levels is supported. Higher indent levels will be truncated.

Return

A string for indenting with two spaces per level is returned.

bool **efi_event_is_queued**(struct efi_event * *event*)
check if an event is queued

Parameters

struct efi_event * event event

Return

true if event is queued

void **efi_process_event_queue**(void)
process event queue

Parameters

void no arguments

void **efi_queue_event**(struct efi_event * *event*)
queue an EFI event

Parameters

struct efi_event * event event to signal

Description

This function queues the notification function of the event for future execution.

efi_status_t **is_valid_tpl**(efi_uintn_t *tpl*)
check if the task priority level is valid

Parameters

efi_uintn_t tpl TPL level to check

Return

status code

void **efi_signal_event**(struct efi_event * *event*)
signal an EFI event

Parameters

struct efi_event * event event to signal

Description

This function signals an event. If the event belongs to an event group all events of the group are signaled. If they are of type EVT_NOTIFY_SIGNAL their notification function is queued.

For the SignalEvent service see efi_signal_event_ext.

unsigned long EFIAPI **efi_raise_tpl**(efi_uintn_t *new_tpl*)
raise the task priority level

Parameters

efi_uintn_t new_tpl new value of the task priority level

Description

This function implements the RaiseTpl service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

old value of the task priority level

void EFIAPI **efi_restore_tpl**(efi_uintn_t *old_tpl*)
lower the task priority level

Parameters

efi_uintn_t old_tpl value of the task priority level to be restored

Description

This function implements the RestoreTpl service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t EFIAPI **efi_allocate_pages_ext**(int *type*, int *memory_type*, efi_uintn_t *pages*, uint64_t * *memory*)
allocate memory pages

Parameters

int type type of allocation to be performed

int memory_type usage type of the allocated memory

efi_uintn_t pages number of pages to be allocated

uint64_t * memory allocated memory

Description

This function implements the AllocatePages service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_free_pages_ext**(uint64_t *memory*, efi_uintn_t *pages*)
Free memory pages.

Parameters

uint64_t memory start of the memory area to be freed

efi_uintn_t pages number of pages to be freed

Description

This function implements the FreePages service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_get_memory_map_ext**(efi_uintn_t * *memory_map_size*, struct efi_mem_desc * *memory_map*, efi_uintn_t * *map_key*, efi_uintn_t * *descriptor_size*, uint32_t * *descriptor_version*)
get map describing memory usage

Parameters

efi_uintn_t * memory_map_size on entry the size, in bytes, of the memory map buffer, on exit the size of the copied memory map

struct efi_mem_desc * memory_map buffer to which the memory map is written

efi_uintn_t * map_key key for the memory map

efi_uintn_t * descriptor_size size of an individual memory descriptor

uint32_t * descriptor_version version number of the memory descriptor structure

Description

This function implements the GetMemoryMap service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_allocate_pool_ext**(int *pool_type*, efi_uintn_t *size*, void ** *buffer*)
allocate memory from pool

Parameters

int pool_type type of the pool from which memory is to be allocated

efi_uintn_t size number of bytes to be allocated

void ** buffer allocated memory

Description

This function implements the AllocatePool service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_free_pool_ext**(void * *buffer*)
free memory from pool

Parameters

void * buffer start of memory to be freed

Description

This function implements the FreePool service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

void **efi_add_handle**(efi_handle_t *handle*)
add a new handle to the object list

Parameters

efi_handle_t handle handle to be added

Description

The protocols list is initialized. The handle is added to the list of known UEFI objects.

efi_status_t **efi_create_handle**(efi_handle_t * *handle*)
create handle

Parameters

efi_handle_t * handle new handle

Return

status code

efi_status_t **efi_search_protocol**(const efi_handle_t *handle*, const efi_guid_t * *protocol_guid*, struct efi_handler ** *handler*)
find a protocol on a handle.

Parameters

const efi_handle_t handle handle

const efi_guid_t * protocol_guid GUID of the protocol

struct efi_handler ** handler reference to the protocol

Return

status code

efi_status_t **efi_remove_protocol**(const efi_handle_t *handle*, const efi_guid_t * *protocol*, void * *protocol_interface*)
delete protocol from a handle

Parameters

const efi_handle_t handle handle from which the protocol shall be deleted

const efi_guid_t * protocol GUID of the protocol to be deleted

void * protocol_interface interface of the protocol implementation

Return

status code

efi_status_t **efi_remove_all_protocols**(const efi_handle_t *handle*)
delete all protocols from a handle

Parameters

const efi_handle_t handle handle from which the protocols shall be deleted

Return

status code

void **efi_delete_handle**(efi_handle_t *handle*)
delete handle

Parameters

efi_handle_t handle handle to delete

efi_status_t **efi_is_event**(const struct efi_event * *event*)
check if a pointer is a valid event

Parameters

const struct efi_event * event pointer to check

Return

status code

efi_status_t **efi_create_event**(uint32_t *type*, efi_uintn_t *notify_tpl*, void (EFIAPI *notify_function) (struct efi_event **event*, void *context) *notify_function*, void * *notify_context*, efi_guid_t * *group*, struct efi_event ** *event*)
create an event

Parameters

uint32_t type type of the event to create

efi_uintn_t notify_tpl task priority level of the event

void (EFIAPI *notify_function) (struct efi_event *event, void *context) notify_function
notification function of the event

void * notify_context pointer passed to the notification function

efi_guid_t * group event group

struct efi_event ** event created event

Description

This function is used inside U-Boot code to create an event.

For the API function implementing the CreateEvent service see efi_create_event_ext.

Return

status code

efi_status_t EFIAPI **efi_create_event_ext**(uint32_t *type*, efi_uintn_t *notify_tpl*, void (EFIAPI *notify_function) (struct efi_event **event*, void *context) *notify_function*, void * *notify_context*, struct efi_event ** *event*)
create an event

Parameters

uint32_t type type of the event to create

efi_uintn_t notify_tpl task priority level of the event

void (EFIAPI *notify_function) (struct efi_event *event, void *context) notify_function
notification function of the event

void * notify_context pointer passed to the notification function

struct efi_event ** event created event

Description

This function implements the CreateEvent service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

void **efi_timer_check**(void)
check if a timer event has occurred

Parameters

void no arguments

Description

Check if a timer event has occurred or a queued notification function should be called.

Our timers have to work without interrupts, so we check whenever keyboard input or disk accesses happen if enough time elapsed for them to fire.

efi_status_t **efi_set_timer**(struct efi_event * *event*, enum efi_timer_delay *type*, uint64_t *trigger_time*)
set the trigger time for a timer event or stop the event

Parameters

struct efi_event * event event for which the timer is set

enum efi_timer_delay type type of the timer

uint64_t trigger_time trigger period in multiples of 100 ns

Description

This is the function for internal usage in U-Boot. For the API function implementing the SetTimer service see efi_set_timer_ext.

Return

status code

efi_status_t EFIAPI **efi_set_timer_ext**(struct efi_event * *event*, enum efi_timer_delay *type*, uint64_t *trigger_time*)
Set the trigger time for a timer event or stop the event

Parameters

struct efi_event * event event for which the timer is set

enum efi_timer_delay type type of the timer

uint64_t trigger_time trigger period in multiples of 100 ns

Description

This function implements the SetTimer service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_wait_for_event**(efi_uintn_t *num_events*, struct efi_event ** *event*, efi_uintn_t * *index*)
wait for events to be signaled

Parameters

efi_uintn_t num_events number of events to be waited for

struct efi_event ** event events to be waited for

efi_uintn_t * index index of the event that was signaled

Description

This function implements the WaitForEvent service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_signal_event_ext**(struct efi_event * *event*)
signal an EFI event

Parameters

struct efi_event * event event to signal

Description

This function implements the SignalEvent service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

This functions sets the signaled state of the event and queues the notification function for execution.

Return

status code

efi_status_t EFIAPI **efi_close_event**(struct efi_event * *event*)
close an EFI event

Parameters

struct efi_event * event event to close

Description

This function implements the CloseEvent service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_check_event**(struct efi_event * *event*)
check if an event is signaled

Parameters

struct efi_event * event event to check

Description

This function implements the CheckEvent service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

If an event is not signaled yet, the notification function is queued. The signaled state is cleared.

Return

status code

struct efi_object * **efi_search_obj**(const efi_handle_t *handle*)
find the internal EFI object for a handle

Parameters

const efi_handle_t handle handle to find

Return

EFI object

struct efi_open_protocol_info_entry * **efi_create_open_info**(struct efi_handler * *handler*)
create open protocol info entry and add it to a protocol

Parameters

struct efi_handler * handler handler of a protocol

Return

open protocol info entry

efi_status_t **efi_delete_open_info**(struct efi_open_protocol_info_item * *item*)
remove an open protocol info entry from a protocol

Parameters

struct efi_open_protocol_info_item * item open protocol info entry to delete

Return

status code

efi_status_t **efi_add_protocol**(const efi_handle_t *handle*, const efi_guid_t * *protocol*, void * *protocol_interface*)
install new protocol on a handle

Parameters

const efi_handle_t handle handle on which the protocol shall be installed

const efi_guid_t * protocol GUID of the protocol to be installed

void * protocol_interface interface of the protocol implementation

Return

status code

efi_status_t EFIAPI **efi_install_protocol_interface**(efi_handle_t * *handle*, const efi_guid_t * *protocol*, int *protocol_interface_type*, void * *protocol_interface*)

install protocol interface

Parameters

efi_handle_t * handle handle on which the protocol shall be installed

const efi_guid_t * protocol GUID of the protocol to be installed

int protocol_interface_type type of the interface to be installed, always EFI_NATIVE_INTERFACE

void * protocol_interface interface of the protocol implementation

Description

This function implements the InstallProtocolInterface service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_get_drivers**(efi_handle_t *handle*, const efi_guid_t * *protocol*, efi_uintn_t * *number_of_drivers*, efi_handle_t ** *driver_handle_buffer*)

get all drivers associated to a controller

Parameters

efi_handle_t handle handle of the controller

const efi_guid_t * protocol protocol GUID (optional)

efi_uintn_t * number_of_drivers number of child controllers

efi_handle_t ** driver_handle_buffer handles of the the drivers

Description

The allocated buffer has to be freed with `free()`.

Return

status code

efi_status_t **efi_disconnect_all_drivers**(efi_handle_t *handle*, const efi_guid_t * *protocol*, efi_handle_t *child_handle*)

disconnect all drivers from a controller

Parameters

efi_handle_t handle handle of the controller

const efi_guid_t * protocol protocol GUID (optional)

efi_handle_t child_handle handle of the child to destroy

Description

This function implements the DisconnectController service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_uninstall_protocol**(efi_handle_t *handle*, const efi_guid_t * *protocol*, void * *protocol_interface*)
uninstall protocol interface

Parameters

efi_handle_t handle handle from which the protocol shall be removed

const efi_guid_t * protocol GUID of the protocol to be removed

void * protocol_interface interface to be removed

Description

This function DOES NOT delete a handle without installed protocol.

Return

status code

efi_status_t EFIAPI **efi_uninstall_protocol_interface**(efi_handle_t *handle*, const efi_guid_t * *protocol*, void * *protocol_interface*)
uninstall protocol interface

Parameters

efi_handle_t handle handle from which the protocol shall be removed

const efi_guid_t * protocol GUID of the protocol to be removed

void * protocol_interface interface to be removed

Description

This function implements the UninstallProtocolInterface service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_register_protocol_notify**(const efi_guid_t * *protocol*, struct efi_event * *event*, void ** *registration*)
register an event for notification when a protocol is installed.

Parameters

const efi_guid_t * protocol GUID of the protocol whose installation shall be notified

struct efi_event * event event to be signaled upon installation of the protocol

void ** registration key for retrieving the registration information

Description

This function implements the RegisterProtocolNotify service. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

int **efi_search**(enum efi_locate_search_type *search_type*, const efi_guid_t * *protocol*, efi_handle_t *handle*)
determine if an EFI handle implements a protocol

Parameters

enum efi_locate_search_type search_type selection criterion

const efi_guid_t * protocol GUID of the protocol

efi_handle_t handle handle

Description

See the documentation of the LocateHandle service in the UEFI specification.

Return

0 if the handle implements the protocol

struct efi_register_notify_event * **efi_check_register_notify_event**(void * *key*)
check if registration key is valid

Parameters

void * key registration key

Description

Check that a pointer is a valid registration key as returned by `RegisterProtocolNotify()`.

Return

valid registration key or NULL

efi_status_t **efi_locate_handle**(enum efi_locate_search_type *search_type*, const efi_guid_t * *protocol*, void * *search_key*, efi_uintn_t * *buffer_size*, efi_handle_t * *buffer*)
locate handles implementing a protocol

Parameters

enum efi_locate_search_type search_type selection criterion

const efi_guid_t * protocol GUID of the protocol

void * search_key registration key

efi_uintn_t * buffer_size size of the buffer to receive the handles in bytes

efi_handle_t * buffer buffer to receive the relevant handles

Description

This function is meant for U-Boot internal calls. For the API implementation of the LocateHandle service see efi_locate_handle_ext.

Return

status code

efi_status_t EFIAPI **efi_locate_handle_ext**(enum efi_locate_search_type *search_type*, const efi_guid_t * *protocol*, void * *search_key*, efi_uintn_t * *buffer_size*, efi_handle_t * *buffer*)
locate handles implementing a protocol.

Parameters

enum efi_locate_search_type search_type selection criterion

const efi_guid_t * protocol GUID of the protocol

void * search_key registration key

efi_uintn_t * buffer_size size of the buffer to receive the handles in bytes

efi_handle_t * buffer buffer to receive the relevant handles

Description

This function implements the LocateHandle service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

0 if the handle implements the protocol

void **efi_remove_configuration_table**(int *i*)
collapses configuration table entries, removing index i

Parameters

int i index of the table entry to be removed

efi_status_t **efi_install_configuration_table**(const efi_guid_t * *guid*, void * *table*)
adds, updates, or removes a configuration table

Parameters

const efi_guid_t * guid GUID of the installed table

void * table table to be installed

Description

This function is used for internal calls. For the API implementation of the InstallConfigurationTable service see efi_install_configuration_table_ext.

Return

status code

efi_status_t EFIAPI **efi_install_configuration_table_ext**(efi_guid_t * *guid*, void * *table*)
Adds, updates, or removes a configuration table.

Parameters

efi_guid_t * guid GUID of the installed table

void * table table to be installed

Description

This function implements the InstallConfigurationTable service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_setup_loaded_image**(struct efi_device_path * *device_path*, struct efi_device_path * *file_path*, struct efi_loaded_image_obj ** *handle_ptr*, struct efi_loaded_image ** *info_ptr*)
initialize a loaded image

Parameters

struct efi_device_path * device_path device path of the loaded image

struct efi_device_path * file_path file path of the loaded image

struct efi_loaded_image_obj ** handle_ptr handle of the loaded image

struct efi_loaded_image ** info_ptr loaded image protocol

Description

Initialize a loaded_image_info and loaded_image_info object with correct protocols, boot-device, etc.

In case of an error *handle_ptr and *info_ptr are set to NULL and an error code is returned.

Return

status code

efi_status_t **efi_load_image_from_path**(struct efi_device_path * *file_path*, void ** *buffer*, efi_uintn_t * *size*)
load an image using a file path

Parameters

struct efi_device_path * file_path the path of the image to load

void ** buffer buffer containing the loaded image

efi_uintn_t * size size of the loaded image

Description

Read a file into a buffer allocated as EFI_BOOT_SERVICES_DATA. It is the callers obligation to update the memory type as needed.

Return

status code

efi_status_t EFIAPI **efi_load_image**(bool *boot_policy*, efi_handle_t *parent_image*, struct efi_device_path * *file_path*, void * *source_buffer*, efi_uintn_t *source_size*, efi_handle_t * *image_handle*)
load an EFI image into memory

Parameters

bool boot_policy true for request originating from the boot manager

efi_handle_t parent_image the caller's image handle

struct efi_device_path * file_path the path of the image to load

void * source_buffer memory location from which the image is installed

efi_uintn_t source_size size of the memory area from which the image is installed

efi_handle_t * image_handle handle for the newly installed image

Description

This function implements the LoadImage service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

void **efi_exit_caches**(void)
fix up caches for EFI payloads if necessary

Parameters

void no arguments

efi_status_t EFIAPI **efi_exit_boot_services**(efi_handle_t *image_handle*, efi_uintn_t *map_key*)
stop all boot services

Parameters

efi_handle_t image_handle handle of the loaded image

efi_uintn_t map_key key of the memory map

Description

This function implements the ExitBootServices service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

All timer events are disabled. For exit boot services events the notification function is called. The boot services are disabled in the system table.

Return

status code

efi_status_t EFIAPI **efi_get_next_monotonic_count**(uint64_t * *count*)
get next value of the counter

Parameters

uint64_t * count returned value of the counter

Description

This function implements the NextMonotonicCount service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_stall**(unsigned long *microseconds*)
sleep

Parameters

unsigned long microseconds period to sleep in microseconds

Description

This function implements the Stall service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_set_watchdog_timer**(unsigned long *timeout*, uint64_t *watchdog_code*, unsigned long *data_size*, uint16_t * *watchdog_data*)
reset the watchdog timer

Parameters

unsigned long timeout seconds before reset by watchdog

uint64_t watchdog_code code to be logged when resetting

unsigned long data_size size of buffer in bytes

uint16_t * watchdog_data buffer with data describing the reset reason

Description

This function implements the SetWatchdogTimer service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_close_protocol**(efi_handle_t *handle*, const efi_guid_t * *protocol*, efi_handle_t *agent_handle*, efi_handle_t *controller_handle*)
close a protocol

Parameters

efi_handle_t handle handle on which the protocol shall be closed

const efi_guid_t * protocol GUID of the protocol to close

efi_handle_t agent_handle handle of the driver

efi_handle_t controller_handle handle of the controller

Description

This function implements the CloseProtocol service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_open_protocol_information** (efi_handle_t *handle*, const efi_guid_t * *protocol*, struct efi_open_protocol_info_entry ** *entry_buffer*, efi_uintn_t * *entry_count*)

provide information about then open status of a protocol on a handle

Parameters

efi_handle_t handle handle for which the information shall be retrieved

const efi_guid_t * protocol GUID of the protocol

struct efi_open_protocol_info_entry ** entry_buffer buffer to receive the open protocol information

efi_uintn_t * entry_count number of entries available in the buffer

Description

This function implements the OpenProtocolInformation service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_protocols_per_handle** (efi_handle_t *handle*, efi_guid_t *** *protocol_buffer*, efi_uintn_t * *protocol_buffer_count*)

get protocols installed on a handle

Parameters

efi_handle_t handle handle for which the information is retrieved

efi_guid_t * protocol_buffer** buffer with protocol GUIDs

efi_uintn_t * protocol_buffer_count number of entries in the buffer

Description

This function implements the ProtocolsPerHandleService.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_locate_handle_buffer** (enum efi_locate_search_type *search_type*, const efi_guid_t * *protocol*, void * *search_key*, efi_uintn_t * *no_handles*, efi_handle_t ** *buffer*)

locate handles implementing a protocol

Parameters

enum efi_locate_search_type search_type selection criterion

const efi_guid_t * protocol GUID of the protocol

void * search_key registration key

efi_uintn_t * no_handles number of returned handles

efi_handle_t ** buffer buffer with the returned handles

Description

This function implements the LocateHandleBuffer service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_locate_protocol**(const efi_guid_t * *protocol*, void * *registration*, void ** *protocol_interface*)
find an interface implementing a protocol

Parameters

const efi_guid_t * protocol GUID of the protocol

void * registration registration key passed to the notification function

void ** protocol_interface interface implementing the protocol

Description

This function implements the LocateProtocol service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_locate_device_path**(const efi_guid_t * *protocol*, struct efi_device_path ** *device_path*, efi_handle_t * *device*)
Get the device path and handle of an device implementing a protocol

Parameters

const efi_guid_t * protocol GUID of the protocol

struct efi_device_path ** device_path device path

efi_handle_t * device handle of the device

Description

This function implements the LocateDevicePath service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_install_multiple_protocol_interfaces**(efi_handle_t * *handle*, ...)
Install multiple protocol interfaces

Parameters

efi_handle_t * handle handle on which the protocol interfaces shall be installed

... NULL terminated argument list with pairs of protocol GUIDS and interfaces

Description

This function implements the MultipleProtocolInterfaces service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_uninstall_multiple_protocol_interfaces**(efi_handle_t *handle*, ...)
uninstall multiple protocol interfaces

Parameters

efi_handle_t handle handle from which the protocol interfaces shall be removed

... NULL terminated argument list with pairs of protocol GUIDS and interfaces

Description

This function implements the UninstallMultipleProtocolInterfaces service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_calculate_crc32**(const void * *data*, efi_uintn_t *data_size*, u32 * *crc32_p*)
calculate cyclic redundancy code

Parameters

const void * data buffer with data

efi_uintn_t data_size size of buffer in bytes

u32 * crc32_p cyclic redundancy code

Description

This function implements the CalculateCrc32 service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

void EFIAPI **efi_copy_mem**(void * *destination*, const void * *source*, size_t *length*)
copy memory

Parameters

void * destination destination of the copy operation

const void * source source of the copy operation

size_t length number of bytes to copy

Description

This function implements the CopyMem service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

void EFIAPI **efi_set_mem**(void * *buffer*, size_t *size*, uint8_t *value*)
Fill memory with a byte value.

Parameters

void * buffer buffer to fill

size_t size size of buffer in bytes

uint8_t value byte to copy to the buffer

Description

This function implements the SetMem service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t **efi_protocol_open**(struct efi_handler * *handler*, void ** *protocol_interface*, void * *agent_handle*, void * *controller_handle*, uint32_t *attributes*)
open protocol interface on a handle

Parameters

struct efi_handler * handler handler of a protocol

void ** protocol_interface interface implementing the protocol

void * agent_handle handle of the driver

void * controller_handle handle of the controller

uint32_t attributes attributes indicating how to open the protocol

Return

status code

efi_status_t EFIAPI **efi_open_protocol**(efi_handle_t *handle*, const efi_guid_t * *protocol*, void ** *protocol_interface*, efi_handle_t *agent_handle*, efi_handle_t *controller_handle*, uint32_t *attributes*)
open protocol interface on a handle

Parameters

efi_handle_t handle handle on which the protocol shall be opened

const efi_guid_t * protocol GUID of the protocol

void ** protocol_interface interface implementing the protocol

efi_handle_t agent_handle handle of the driver

efi_handle_t controller_handle handle of the controller

uint32_t attributes attributes indicating how to open the protocol

Description

This function implements the OpenProtocol interface.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_start_image**(efi_handle_t *image_handle*, efi_uintn_t * *exit_data_size*, u16 ** *exit_data*)
call the entry point of an image

Parameters

efi_handle_t image_handle handle of the image

efi_uintn_t * exit_data_size size of the buffer

u16 ** exit_data buffer to receive the exit data of the called image

Description

This function implements the StartImage service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_delete_image**(struct efi_loaded_image_obj * *image_obj*, struct efi_loaded_image * *loaded_image_protocol*)
delete loaded image from memory)

Parameters

struct efi_loaded_image_obj * image_obj handle of the loaded image

struct efi_loaded_image * loaded_image_protocol loaded image protocol

efi_status_t EFIAPI **efi_unload_image**(efi_handle_t *image_handle*)
unload an EFI image

Parameters

efi_handle_t image_handle handle of the image to be unloaded

Description

This function implements the UnloadImage service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_update_exit_data**(struct efi_loaded_image_obj * *image_obj*, efi_uintn_t *exit_data_size*, u16 * *exit_data*)
fill exit data parameters of `StartImage()`

Parameters

struct efi_loaded_image_obj * image_obj image handle

efi_uintn_t exit_data_size size of the exit data buffer

u16 * exit_data buffer with data returned by UEFI payload

Return

status code

efi_status_t EFIAPI **efi_exit**(efi_handle_t *image_handle*, efi_status_t *exit_status*, efi_uintn_t *exit_data_size*, u16 * *exit_data*)
leave an EFI application or driver

Parameters

efi_handle_t image_handle handle of the application or driver that is exiting

efi_status_t exit_status status code

efi_uintn_t exit_data_size size of the buffer in bytes

u16 * exit_data buffer with data describing an error

Description

This function implements the Exit service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_handle_protocol**(efi_handle_t *handle*, const efi_guid_t * *protocol*, void ** *protocol_interface*)
get interface of a protocol on a handle

Parameters

efi_handle_t handle handle on which the protocol shall be opened

const efi_guid_t * protocol GUID of the protocol

void ** protocol_interface interface implementing the protocol

Description

This function implements the HandleProtocol service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_bind_controller**(efi_handle_t *controller_handle*, efi_handle_t *driver_image_handle*, struct efi_device_path * *remain_device_path*)

bind a single driver to a controller

Parameters

efi_handle_t controller_handle controller handle

efi_handle_t driver_image_handle driver handle

struct efi_device_path * remain_device_path remaining path

Return

status code

efi_status_t **efi_connect_single_controller**(efi_handle_t *controller_handle*, efi_handle_t * *driver_image_handle*, struct efi_device_path * *remain_device_path*)

connect a single driver to a controller

Parameters

efi_handle_t controller_handle controller

efi_handle_t * driver_image_handle driver

struct efi_device_path * remain_device_path remaining path

Return

status code

efi_status_t EFIAPI **efi_connect_controller**(efi_handle_t *controller_handle*, efi_handle_t * *driver_image_handle*, struct efi_device_path * *remain_device_path*, bool *recursive*)

connect a controller to a driver

Parameters

efi_handle_t controller_handle handle of the controller

efi_handle_t * driver_image_handle handle of the driver

struct efi_device_path * remain_device_path device path of a child controller

bool recursive true to connect all child controllers

Description

This function implements the ConnectController service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

First all driver binding protocol handles are tried for binding drivers. Afterwards all handles that have opened a protocol of the controller with EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER are connected to drivers.

Return

status code

efi_status_t EFIAPI **efi_reinstall_protocol_interface**(efi_handle_t *handle*, const efi_guid_t * *protocol*, void * *old_interface*, void * *new_interface*)

reinstall protocol interface

Parameters

efi_handle_t handle handle on which the protocol shall be reinstalled

const efi_guid_t * protocol GUID of the protocol to be installed

void * old_interface interface to be removed

void * new_interface interface to be installed

Description

This function implements the ReinstallProtocolInterface service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

The old interface is uninstalled. The new interface is installed. Drivers are connected.

Return

status code

efi_status_t **efi_get_child_controllers**(struct efi_object * *efiobj*, efi_handle_t *driver_handle*, efi_uintn_t * *number_of_children*, efi_handle_t ** *child_handle_buffer*)

get all child controllers associated to a driver

Parameters

struct efi_object * efiobj handle of the controller

efi_handle_t driver_handle handle of the driver

efi_uintn_t * number_of_children number of child controllers

efi_handle_t ** child_handle_buffer handles of the the child controllers

Description

The allocated buffer has to be freed with `free()`.

Return

status code

efi_status_t EFIAPI **efi_disconnect_controller**(efi_handle_t *controller_handle*, efi_handle_t *driver_image_handle*, efi_handle_t *child_handle*)

disconnect a controller from a driver

Parameters

efi_handle_t controller_handle handle of the controller

efi_handle_t driver_image_handle handle of the driver

efi_handle_t child_handle handle of the child to destroy

Description

This function implements the DisconnectController service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_initialize_system_table**(void)
Initialize system table

Parameters

void no arguments

Return

status code

Image relocation

efi_status_t **efi_print_image_info**(struct efi_loaded_image_obj * *obj*, struct efi_loaded_image * *image*, void * *pc*)
print information about a loaded image

Parameters

struct efi_loaded_image_obj * obj EFI object

struct efi_loaded_image * image loaded image

void * pc program counter (use NULL to suppress offset output)

Description

If the program counter is located within the image the offset to the base address is shown.

Return

status code

void **efi_print_image_infos**(void * *pc*)
print information about all loaded images

Parameters

void * pc program counter (use NULL to suppress offset output)

efi_status_t **efi_loader_relocate**(const IMAGE_BASE_RELOCATION * *rel*, unsigned long *rel_size*, void * *efi_reloc*, unsigned long *pref_address*)
relocate UEFI binary

Parameters

const IMAGE_BASE_RELOCATION * rel pointer to the relocation table

unsigned long rel_size size of the relocation table in bytes

void * efi_reloc actual load address of the image

unsigned long pref_address preferred load address of the image

Return

status code

void **efi_set_code_and_data_type**(struct efi_loaded_image * *loaded_image_info*, uint16_t *image_type*)
determine the memory types to be used for code and data.

Parameters

struct efi_loaded_image * loaded_image_info image descriptor

uint16_t image_type field Subsystem of the optional header for Windows specific field

int **cmp_pe_section**(const void * *arg1*, const void * *arg2*)
compare virtual addresses of two PE image sections

Parameters

const void * arg1 pointer to pointer to first section header

const void * arg2 pointer to pointer to second section header

Description

Compare the virtual addresses of two sections of an portable executable. The arguments are defined as const void * to allow usage with `qsort()`.

Return

-1 if the virtual address of arg1 is less than that of arg2, 0 if the virtual addresses are equal, 1 if the virtual address of arg1 is greater than that of arg2.

bool **efi_image_parse**(void * *efi*, size_t *len*, struct efi_image_regions ** *regp*, WIN_CERTIFICATE ** *auth*, size_t * *auth_len*)
parse a PE image

Parameters

void * efi Pointer to image

size_t len Size of **efi**

struct efi_image_regions ** regp Pointer to a list of regions

WIN_CERTIFICATE ** auth Pointer to a pointer to authentication data in PE

size_t * auth_len Size of **auth**

Description

Parse image binary in PE32(+) format, assuming that sanity of PE image has been checked by a caller. On success, an address of authentication data in **efi** and its size will be returned in **auth** and **auth_len**, respectively.

Return

true on success, false on error

bool **efi_image_unsigned_authenticate**(struct efi_image_regions * *regs*)
authenticate unsigned image with SHA256 hash

Parameters

struct efi_image_regions * regs List of regions to be verified

Description

If an image is not signed, it doesn't have a signature. In this case, its message digest is calculated and it will be compared with one of hash values stored in signature databases.

Return

true if authenticated, false if not

bool **efi_image_authenticate**(void * *efi*, size_t *efi_size*)
verify a signature of signed image

Parameters

void * efi Pointer to image

size_t efi_size Size of **efi**

Description

A signed image should have its signature stored in a table of its PE header. So if an image is signed and only if if its signature is verified using signature databases, an image is authenticated. If an image is not signed, its validity is checked by using `efi_image_unsigned_authenticated()`. TODO: When AuditMode==0, if the image's signature is not found in the authorized database, or is found in the forbidden database, the image will not be started and instead, information about it will be placed in this table. When AuditMode==1, an EFI_IMAGE_EXECUTION_INFO element is created in the EFI_IMAGE_EXECUTION_INFO_TABLE for every certificate found in the certificate table of every image that is validated.

Return

true if authenticated, false if not

efi_status_t **efi_load_pe**(struct efi_loaded_image_obj * *handle*, void * *efi*, size_t *efi_size*, struct efi_loaded_image * *loaded_image_info*)
relocate EFI binary

Parameters

struct efi_loaded_image_obj * handle loaded image handle

void * efi pointer to the EFI binary

size_t efi_size size of **efi** binary

struct efi_loaded_image * loaded_image_info loaded image protocol

Description

This function loads all sections from a PE binary into a newly reserved piece of memory. On success the entry point is returned as handle->entry.

Return

status code

Memory services

struct **efi_pool_allocation**
memory block allocated from pool

Definition

```
struct efi_pool_allocation {
  u64 num_pages;
  u64 checksum;
  char data[];
};
```

Members

num_pages number of pages allocated

checksum checksum

data allocated pool memory

Description

U-Boot services each UEFI `AllocatePool()` request as a separate (multiple) page allocation. We have to track the number of pages to be able to free the correct amount later.

The checksum calculated in function *checksum()* is used in `FreePool()` to avoid freeing memory not allocated by `AllocatePool()` and duplicate freeing.

EFI requires 8 byte alignment for pool allocations, so we can prepend each allocation with these header fields.

u64 **checksum**(struct *efi_pool_allocation* * *alloc*)
calculate checksum for memory allocated from pool

Parameters

struct efi_pool_allocation * alloc allocation header

Return

checksum, always non-zero

efi_status_t **efi_add_memory_map_pg**(u64 *start*, u64 *pages*, int *memory_type*, bool *overlap_only_ram*)
add pages to the memory map

Parameters

u64 start start address, must be a multiple of EFI_PAGE_SIZE

u64 pages number of pages to add

int memory_type type of memory added

bool overlap_only_ram region may only overlap RAM

Return

status code

efi_status_t **efi_add_memory_map**(u64 *start*, u64 *size*, int *memory_type*)
add memory area to the memory map

Parameters

u64 start start address of the memory area

u64 size length in bytes of the memory area

int memory_type type of memory added

Return

status code

This function automatically aligns the start and size of the memory area to EFI_PAGE_SIZE.

efi_status_t **efi_check_allocated**(u64 *addr*, bool *must_be_allocated*)
validate address to be freed

Parameters

u64 addr address of page to be freed

bool must_be_allocated return success if the page is allocated

Description

Check that the address is within allocated memory:

- The address must be in a range of the memory map.
- The address may not point to EFI_CONVENTIONAL_MEMORY.

Page alignment is not checked as this is not a requirement of `efi_free_pool()`.

Return

status code

efi_status_t **efi_free_pages**(uint64_t *memory*, efi_uintn_t *pages*)
free memory pages

Parameters

uint64_t memory start of the memory area to be freed

efi_uintn_t pages number of pages to be freed

Return

status code

efi_status_t **efi_allocate_pool**(int *pool_type*, efi_uintn_t *size*, void ** *buffer*)
allocate memory from pool

Parameters

int pool_type type of the pool from which memory is to be allocated

efi_uintn_t size number of bytes to be allocated

void ** buffer allocated memory

Return

status code

efi_status_t **efi_free_pool**(void * *buffer*)
free memory from pool

Parameters

void * buffer start of memory to be freed

Return

status code

efi_status_t **efi_add_conventional_memory_map**(u64 *ram_start*, u64 *ram_end*, u64 *ram_top*)
add a RAM memory area to the map

Parameters

u64 ram_start start address of a RAM memory area

u64 ram_end end address of a RAM memory area

u64 ram_top max address to be used as conventional memory

Return

status code

SetWatchdogTimer service

void EFIAPI **efi_watchdog_timer_notify**(struct efi_event * *event*, void * *context*)
resets system upon watchdog event

Parameters

struct efi_event * event the watchdog event

void * context not used

Description

Reset the system when the watchdog event is notified.

efi_status_t **efi_set_watchdog**(unsigned long *timeout*)
resets the watchdog timer

Parameters

unsigned long timeout seconds before reset by watchdog

Description

This function is used by the SetWatchdogTimer service.

Return

status code

efi_status_t **efi_watchdog_register**(void)
initializes the EFI watchdog

Parameters

void no arguments

Description

This function is called by *efi_init_obj_list()*.

Return

status code

Runtime services

efi_status_t **efi_init_runtime_supported**(void)
create runtime properties table

Parameters

void no arguments

Description

Create a configuration table specifying which services are available at runtime.

Return

status code

void __efi_runtime **efi_memcpy_runtime**(void * *dest*, const void * *src*, size_t *n*)
copy memory area

Parameters

void * dest destination buffer

const void * src source buffer

size_t n number of bytes to copy

Description

At runtime `memcpy()` is not available.

Overlapping memory areas can be copied safely if src >= dest.

Return

pointer to destination buffer

void __efi_runtime **efi_update_table_header_crc32**(struct efi_table_hdr * *table*)
Update crc32 in table header

Parameters

struct efi_table_hdr * table EFI table

void EFIAPI **efi_reset_system_boottime**(enum efi_reset_type *reset_type*, efi_status_t *reset_status*, unsigned long *data_size*, void * *reset_data*)
reset system at boot time

Parameters

enum efi_reset_type reset_type type of reset to perform

efi_status_t reset_status status code for the reset

unsigned long data_size size of reset_data

void * reset_data information about the reset

Description

This function implements the `ResetSystem()` runtime service before `SetVirtualAddressMap()` is called.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t EFIAPI **efi_get_time_boottime**(struct efi_time * *time*, struct efi_time_cap * *capabilities*)
get current time at boot time

Parameters

struct efi_time * time pointer to structure to receive current time

struct efi_time_cap * capabilities pointer to structure to receive RTC properties

Description

This function implements the GetTime runtime service before `SetVirtualAddressMap()` is called.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

int **efi_validate_time**(struct efi_time * *time*)
checks if timestamp is valid

Parameters

struct efi_time * time timestamp to validate

Return

0 if timestamp is valid, 1 otherwise

efi_status_t EFIAPI **efi_set_time_boottime**(struct efi_time * *time*)
set current time

Parameters

struct efi_time * time pointer to structure to with current time

Description

This function implements the `SetTime()` runtime service before `SetVirtualAddressMap()` is called.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

void __efi_runtime EFIAPI **efi_reset_system**(enum efi_reset_type *reset_type*, efi_status_t *reset_status*, unsigned long *data_size*, void * *reset_data*)
reset system

Parameters

enum efi_reset_type reset_type type of reset to perform

efi_status_t reset_status status code for the reset

unsigned long data_size size of reset_data

void * reset_data information about the reset

Description

This function implements the `ResetSystem()` runtime service after `SetVirtualAddressMap()` is called. As this placeholder cannot reset the system it simply return to the caller.

Boards may override the helpers below to implement reset functionality.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t **efi_reset_system_init**(void)
initialize the reset driver

Parameters

void no arguments

Description

Boards may override this function to initialize the reset driver.

efi_status_t __efi_runtime EFIAPI **efi_get_time**(struct efi_time * *time*, struct efi_time_cap * *capabilities*)
get current time

Parameters

struct efi_time * time pointer to structure to receive current time

struct efi_time_cap * capabilities pointer to structure to receive RTC properties

Description

This function implements the GetTime runtime service after `SetVirtualAddressMap()` is called. As the U-Boot driver are not available anymore only an error code is returned.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t __efi_runtime EFIAPI **efi_set_time**(struct efi_time * *time*)
set current time

Parameters

struct efi_time * time pointer to structure to with current time

Description

This function implements the SetTime runtime service after `SetVirtualAddressMap()` is called. As the U-Boot driver are not available anymore only an error code is returned.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

bool **efi_is_runtime_service_pointer**(void * *p*)
check if pointer points to runtime table

Parameters

void * p pointer to check

Return

true if the pointer points to a service function pointer in the runtime table

void **efi_runtime_detach**(void)
detach unimplemented runtime functions

Parameters

void no arguments

__efi_runtime efi_status_t EFIAPI **efi_set_virtual_address_map_runtime**(efi_uintn_t *memory_map_size*, efi_uintn_t *descriptor_size*, uint32_t *descriptor_version*, struct efi_mem_desc * *virtmap*)

change from physical to virtual mapping

Parameters

efi_uintn_t memory_map_size size of the virtual map

efi_uintn_t descriptor_size size of an entry in the map

uint32_t descriptor_version version of the map entries

struct efi_mem_desc * virtmap virtual address mapping information

Description

This function implements the `SetVirtualAddressMap()` runtime service after it is first called.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code EFI_UNSUPPORTED

__efi_runtime efi_status_t EFIAPI **efi_convert_pointer_runtime**(efi_uintn_t *debug_disposition*, void ** *address*)

convert from physical to virtual pointer

Parameters

efi_uintn_t debug_disposition indicates if pointer may be converted to NULL

void ** address pointer to be converted

Description

This function implements the `ConvertPointer()` runtime service after the first call to `SetVirtualAddressMap()`.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code EFI_UNSUPPORTED

__efi_runtime efi_status_t EFIAPI **efi_convert_pointer**(efi_uintn_t *debug_disposition*, void ** *address*)

convert from physical to virtual pointer

Parameters

efi_uintn_t debug_disposition indicates if pointer may be converted to NULL

void ** address pointer to be converted

Description

This function implements the `ConvertPointer()` runtime service until the first call to `SetVirtualAddressMap()`.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_set_virtual_address_map**(efi_uintn_t *memory_map_size*, efi_uintn_t *descriptor_size*, uint32_t *descriptor_version*, struct efi_mem_desc * *virtmap*)

change from physical to virtual mapping

Parameters

efi_uintn_t memory_map_size size of the virtual map

efi_uintn_t descriptor_size size of an entry in the map

uint32_t descriptor_version version of the map entries

struct efi_mem_desc * virtmap virtual address mapping information

Description

This function implements the `SetVirtualAddressMap()` runtime service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_add_runtime_mmio**(void * *mmio_ptr*, u64 *len*)

add memory-mapped IO region

Parameters

void * mmio_ptr pointer to a pointer to the start of the memory-mapped IO region

u64 len size of the memory-mapped IO region

Description

This function adds a memory-mapped IO region to the memory map to make it available at runtime.

Return

status code

efi_status_t __efi_runtime EFIAPI **efi_unimplemented**(void)

replacement function, returns EFI_UNSUPPORTED

Parameters

void no arguments

Description

This function is used after `SetVirtualAddressMap()` is called as replacement for services that are not available anymore due to constraints of the U-Boot implementation.

Return

EFI_UNSUPPORTED

efi_status_t __efi_runtime EFIAPI **efi_update_capsule**(struct efi_capsule_header ** *capsule_header_array*, efi_uintn_t *capsule_count*, u64 *scatter_gather_list*)

process information from operating system

Parameters

struct efi_capsule_header ** capsule_header_array pointer to array of virtual pointers

efi_uintn_t capsule_count number of pointers in capsule_header_array

u64 scatter_gather_list pointer to arry of physical pointers

Description

This function implements the `UpdateCapsule()` runtime service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t __efi_runtime EFIAPI **efi_query_capsule_caps**(struct efi_capsule_header ** *capsule_header_array*, efi_uintn_t *capsule_count*, u64 * *maximum_capsule_size*, u32 * *reset_type*)

check if capsule is supported

Parameters

struct efi_capsule_header ** capsule_header_array pointer to array of virtual pointers

efi_uintn_t capsule_count number of pointers in capsule_header_array

u64 * maximum_capsule_size maximum capsule size

u32 * reset_type type of reset needed for capsule update

Description

This function implements the `QueryCapsuleCapabilities()` runtime service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

Variable services

efi_status_t **efi_get_variable_int**(u16 * *variable_name*, const efi_guid_t * *vendor*, u32 * *attributes*, efi_uintn_t * *data_size*, void * *data*, u64 * *timep*)

retrieve value of a UEFI variable

Parameters

u16 * variable_name name of the variable

const efi_guid_t * vendor vendor GUID

u32 * attributes attributes of the variable

efi_uintn_t * data_size size of the buffer to which the variable value is copied

void * data buffer to which the variable value is copied

u64 * timep authentication time (seconds since start of epoch)

Return

status code

efi_status_t **efi_set_variable_int**(u16 * *variable_name*, const efi_guid_t * *vendor*, u32 *attributes*, efi_uintn_t *data_size*, const void * *data*, bool *ro_check*)

set value of a UEFI variable

Parameters

u16 * variable_name name of the variable

const efi_guid_t * vendor vendor GUID

u32 attributes attributes of the variable

efi_uintn_t data_size size of the buffer with the variable value

const void * data buffer with the variable value

bool ro_check check the read only read only bit in attributes

Return

status code

efi_status_t **efi_get_next_variable_name_int**(efi_uintn_t * *variable_name_size*, u16 * *variable_name*, efi_guid_t * *vendor*)

enumerate the current variable names

Parameters

efi_uintn_t * variable_name_size size of variable_name buffer in byte

u16 * variable_name name of uefi variable's name in u16

efi_guid_t * vendor vendor's guid

Description

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t **efi_query_variable_info_int**(u32 *attributes*, u64 * *maximum_variable_storage_size*, u64 * *remaining_variable_storage_size*, u64 * *maximum_variable_size*)

get information about EFI variables

Parameters

u32 attributes bitmask to select variables to be queried

u64 * maximum_variable_storage_size maximum size of storage area for the selected variable types

u64 * remaining_variable_storage_size remaining size of storage are for the selected variable types

u64 * maximum_variable_size maximum size of a variable of the selected type

Description

This function implements the `QueryVariableInfo()` runtime service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

struct **efi_var_entry**

UEFI variable file entry

Definition

```
struct efi_var_entry {
  u32 length;
  u32 attr;
  u64 time;
  efi_guid_t guid;
  u16 name[];
};
```

Members

length length of enty, multiple of 8

attr variable attributes

time authentication time (seconds since start of epoch)

guid vendor GUID

name UTF16 variable name

struct **efi_var_file**
file for storing UEFI variables

Definition

```
struct efi_var_file {
  u64 reserved;
  u64 magic;
  u32 length;
  u32 crc32;
  struct efi_var_entry var[];
};
```

Members

reserved unused, may be overwritten by memory probing

magic identifies file format, takes value EFI_VAR_FILE_MAGIC

length length including header

crc32 CRC32 without header

var variables

efi_status_t **efi_var_to_file**(void)
save non-volatile variables as file

Parameters

void no arguments

Description

File ubootefi.var is created on the EFI system partion.

Return

status code

efi_status_t __maybe_unused **efi_var_collect**(struct *efi_var_file* ** *bufp*, loff_t * *lenp*, u32 *check_attr_mask*)
collect variables in buffer

Parameters

struct efi_var_file ** bufp pointer to pointer of buffer with collected variables

loff_t * lenp pointer to length of buffer

u32 check_attr_mask bitmask with required attributes of variables to be collected. variables are only collected if all of the required attributes are set.

Description

A buffer is allocated and filled with variables in a format ready to be written to disk.

Return

status code

efi_status_t **efi_var_restore**(struct *efi_var_file* * *buf*)
restore EFI variables from buffer

Parameters

struct efi_var_file * buf buffer

Return

status code

efi_status_t **efi_var_from_file**(void)
read variables from file

Parameters

void no arguments

Description

File ubootefi.var is read from the EFI system partitions and the variables stored in the file are created.

In case the file does not exist yet or a variable cannot be set EFI_SUCCESS is returned.

Return

status code

efi_status_t **efi_var_mem_init**(void)
set-up variable list

Parameters

void no arguments

Return

status code

struct *efi_var_entry* * **efi_var_mem_find**(const efi_guid_t * *guid*, const u16 * *name*, struct *efi_var_entry* ** *next*)
find a variable in the list

Parameters

const efi_guid_t * guid GUID of the variable

const u16 * name name of the variable

struct efi_var_entry ** next on exit pointer to the next variable after the found one

Return

found variable

void **efi_var_mem_del**(struct *efi_var_entry* * *var*)
delete a variable from the list of variables

Parameters

struct efi_var_entry * var variable to delete

efi_status_t **efi_var_mem_ins**(u16 * *variable_name*, const efi_guid_t * *vendor*, u32 *attributes*, const efi_uintn_t *size1*, const void * *data1*, const efi_uintn_t *size2*, const void * *data2*, const u64 *time*)
append a variable to the list of variables

Parameters

u16 * variable_name variable name

const efi_guid_t * vendor GUID

u32 attributes variable attributes

const efi_uintn_t size1 size of the first data buffer

const void * data1 first data buffer

const efi_uintn_t size2 size of the second data field

const void * data2 second data buffer

const u64 time time of authentication (as seconds since start of epoch) Result: status code

Description

The variable is appended without checking if a variable of the same name already exists. The two data buffers are concatenated.

u64 **efi_var_mem_free**(void)
determine free memory for variables

Parameters

void no arguments

Return

maximum data size plus variable name size

efi_status_t **efi_init_secure_state**(void)
initialize secure boot state

Parameters

void no arguments

Return

status code

enum efi_auth_var_type **efi_auth_var_get_type**(u16 * *name*, const efi_guid_t * *guid*)
convert variable name and guid to enum

Parameters

u16 * name name of UEFI variable

const efi_guid_t * guid guid of UEFI variable

Return

identifier for authentication related variables

efi_status_t __efi_runtime **efi_get_next_variable_name_mem**(efi_uintn_t * *variable_name_size*, u16 * *variable_name*, efi_guid_t * *vendor*)
Runtime common code across efi variable implementations for `GetNextVariable()` from the cached memory copy

Parameters

efi_uintn_t * variable_name_size size of variable_name buffer in byte

u16 * variable_name name of uefi variable's name in u16

efi_guid_t * vendor vendor's guid

Return

status code

efi_status_t __efi_runtime **efi_get_variable_mem**(u16 * *variable_name*, const efi_guid_t * *vendor*, u32 * *attributes*, efi_uintn_t * *data_size*, void * *data*, u64 * *timep*)
Runtime common code across efi variable implementations for `GetVariable()` from the cached memory copy

Parameters

u16 * variable_name name of the variable

const efi_guid_t * vendor vendor GUID

u32 * attributes attributes of the variable

efi_uintn_t * data_size size of the buffer to which the variable value is copied

void * data buffer to which the variable value is copied

u64 * timep authentication time (seconds since start of epoch)

Return

status code

efi_status_t __efi_runtime EFIAPI **efi_get_variable_runtime**(u16 * *variable_name*, const efi_guid_t * *guid*, u32 * *attributes*, efi_uintn_t * *data_size*, void * *data*)

runtime implementation of `GetVariable()`

Parameters

u16 * variable_name name of the variable

const efi_guid_t * guid vendor GUID

u32 * attributes attributes of the variable

efi_uintn_t * data_size size of the buffer to which the variable value is copied

void * data buffer to which the variable value is copied

Return

status code

efi_status_t __efi_runtime EFIAPI **efi_get_next_variable_name_runtime**(efi_uintn_t * *variable_name_size*, u16 * *variable_name*, efi_guid_t * *guid*)

runtime implementation of `GetNextVariable()`

Parameters

efi_uintn_t * variable_name_size size of variable_name buffer in byte

u16 * variable_name name of uefi variable's name in u16

efi_guid_t * guid vendor's guid

Return

status code

struct pkcs7_message * **efi_variable_parse_signature**(const void * *buf*, size_t *buflen*, u8 ** *tmpbuf*)

parse a signature in variable

Parameters

const void * buf Pointer to variable's value

size_t buflen Length of **buf**

u8 ** tmpbuf Pointer to temporary buffer

Description

Parse a signature embedded in variable's value and instantiate a pkcs7_message structure. Since `pkcs7_parse_message()` accepts only pkcs7's signedData, some header needed be prepended for correctly parsing authentication data, particularly for variable's. A temporary buffer will be allocated if needed, and it should be kept valid during the authentication because some data in the buffer will be referenced by `efi_signature_verify()`.

Return

Pointer to pkcs7_message structure on success, NULL on error

efi_status_t **efi_variable_authenticate**(u16 * *variable*, const efi_guid_t * *vendor*, efi_uintn_t * *data_size*, const void ** *data*, u32 *given_attr*, u32 * *env_attr*, u64 * *time*)

authenticate a variable

Parameters

u16 * variable Variable name in u16

const efi_guid_t * vendor Guid of variable

efi_uintn_t * data_size Size of **data**

const void ** data Pointer to variable's value

u32 given_attr Attributes to be given at `SetVariable()`

u32 * env_attr Attributes that an existing variable holds

u64 * time signed time that an existing variable holds

Description

Called by `efi_set_variable()` to verify that the input is correct. Will replace the given data pointer with another that points to the actual data to store in the internal memory. On success, **data** and **data_size** will be replaced with variable's actual data, excluding authentication data, and its size, and variable's attributes and signed time will also be returned in **env_attr** and **time**, respectively.

Return

status code

efi_status_t __efi_runtime EFIAPI **efi_query_variable_info_runtime**(u32 *attributes*, u64 * *maximum_variable_storage_size*, u64 * *remaining_variable_storage_size*, u64 * *maximum_variable_size*)

runtime implementation of `QueryVariableInfo()`

Parameters

u32 attributes bitmask to select variables to be queried

u64 * maximum_variable_storage_size maximum size of storage area for the selected variable types

u64 * remaining_variable_storage_size remaining size of storage are for the selected variable types

u64 * maximum_variable_size maximum size of a variable of the selected type

Return

status code

efi_status_t __efi_runtime EFIAPI **efi_set_variable_runtime**(u16 * *variable_name*, const efi_guid_t * *vendor*, u32 *attributes*, efi_uintn_t *data_size*, const void * *data*)

runtime implementation of `SetVariable()`

Parameters

u16 * variable_name name of the variable

const efi_guid_t * vendor vendor GUID

u32 attributes attributes of the variable

efi_uintn_t data_size size of the buffer with the variable value

const void * data buffer with the variable value

Return

status code

void **efi_variables_boot_exit_notify**(void)
notify ExitBootServices() is called

Parameters

void no arguments

efi_status_t **efi_init_variables**(void)
initialize variable services

Parameters

void no arguments

Return

status code

UEFI drivers

UEFI driver uclass

efi_status_t **check_node_type**(efi_handle_t *handle*)
check node type

Parameters

efi_handle_t handle handle to be checked

Description

We do not support partitions as controller handles.

Return

status code

efi_status_t EFIAPI **efi_uc_supported**(struct efi_driver_binding_protocol * *this*, efi_handle_t *controller_handle*, struct efi_device_path * *remaining_device_path*)
check if the driver supports the controller

Parameters

struct efi_driver_binding_protocol * this driver binding protocol

efi_handle_t controller_handle handle of the controller

struct efi_device_path * remaining_device_path path specifying the child controller

Return

status code

efi_status_t EFIAPI **efi_uc_start**(struct efi_driver_binding_protocol * *this*, efi_handle_t *controller_handle*, struct efi_device_path * *remaining_device_path*)
create child controllers and attach driver

Parameters

struct efi_driver_binding_protocol * this driver binding protocol

efi_handle_t controller_handle handle of the controller

struct efi_device_path * remaining_device_path path specifying the child controller

Return

status code

efi_status_t **disconnect_child**(efi_handle_t *controller_handle*, efi_handle_t *child_handle*)
remove a single child controller from the parent controller

Parameters

efi_handle_t controller_handle parent controller

efi_handle_t child_handle child controller

Return

status code

efi_status_t EFIAPI **efi_uc_stop**(struct efi_driver_binding_protocol * *this*, efi_handle_t *controller_handle*, size_t *number_of_children*, efi_handle_t * *child_handle_buffer*)
Remove child controllers and disconnect the controller

Parameters

struct efi_driver_binding_protocol * this driver binding protocol

efi_handle_t controller_handle handle of the controller

size_t number_of_children number of child controllers to remove

efi_handle_t * child_handle_buffer handles of the child controllers to remove

Return

status code

efi_status_t **efi_add_driver**(struct driver * *drv*)
add driver

Parameters

struct driver * drv driver to add

Return

status code

efi_status_t **efi_driver_init**(void)
initialize the EFI drivers

Parameters

void no arguments

Description

Called by `efi_init_obj_list()`.

Return

0 = success, any other value will stop further execution

int **efi_uc_init**(struct uclass * *class*)
initialize the EFI uclass

Parameters

struct uclass * class the EFI uclass

Return

0 = success

int **efi_uc_destroy**(struct uclass * *class*)
destroy the EFI uclass

Parameters

struct uclass * class the EFI uclass

Return

0 = success

Block device driver

ulong **efi_bl_read**(struct udevice * *dev*, lbaint_t *blknr*, lbaint_t *blkcnt*, void * *buffer*)

Parameters

struct udevice * dev device

lbaint_t blknr first block to be read

lbaint_t blkcnt number of blocks to read

void * buffer output buffer

Return

number of blocks transferred

ulong **efi_bl_write**(struct udevice * *dev*, lbaint_t *blknr*, lbaint_t *blkcnt*, const void * *buffer*)

Parameters

struct udevice * dev device

lbaint_t blknr first block to be write

lbaint_t blkcnt number of blocks to write

const void * buffer input buffer

Return

number of blocks transferred

int **efi_bl_bind_partitions**(efi_handle_t *handle*, struct udevice * *dev*)

Parameters

efi_handle_t handle EFI handle of the block device

struct udevice * dev udevice of the block device

Return

number of partitions created

int **efi_bl_bind**(efi_handle_t *handle*, void * *interface*)

Parameters

efi_handle_t handle handle

void * interface block io protocol

Return

0 = success

Protocols

Block IO protocol

struct **efi_disk_obj**
EFI disk object

Definition

```
struct efi_disk_obj {
  struct efi_object header;
  struct efi_block_io ops;
  const char *ifname;
  int dev_index;
  struct efi_block_io_media media;
  struct efi_device_path *dp;
  unsigned int part;
  struct efi_simple_file_system_protocol *volume;
  lbaint_t offset;
  struct blk_desc *desc;
};
```

Members

header EFI object header

ops EFI disk I/O protocol interface

ifname interface name for block device

dev_index device index of block device

media block I/O media information

dp device path to the block device

part partition

volume simple file system protocol of the partition

offset offset into disk for simple partition

desc internal block device descriptor

efi_status_t EFIAPI **efi_disk_reset**(struct efi_block_io * *this*, char *extended_verification*)
reset block device

Parameters

struct efi_block_io * this pointer to the BLOCK_IO_PROTOCOL

char extended_verification extended verification

Description

This function implements the Reset service of the EFI_BLOCK_IO_PROTOCOL.

As U-Boot's block devices do not have a reset function simply return EFI_SUCCESS.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_disk_read_blocks**(struct efi_block_io * *this*, u32 *media_id*, u64 *lba*, efi_uintn_t *buffer_size*, void * *buffer*)
reads blocks from device

Parameters

struct efi_block_io * this pointer to the BLOCK_IO_PROTOCOL

u32 media_id id of the medium to be read from

u64 lba starting logical block for reading

efi_uintn_t buffer_size size of the read buffer

void * buffer pointer to the destination buffer

Description

This function implements the ReadBlocks service of the EFI_BLOCK_IO_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_disk_write_blocks**(struct efi_block_io * *this*, u32 *media_id*, u64 *lba*, efi_uintn_t *buffer_size*, void * *buffer*)

writes blocks to device

Parameters

struct efi_block_io * this pointer to the BLOCK_IO_PROTOCOL

u32 media_id id of the medium to be written to

u64 lba starting logical block for writing

efi_uintn_t buffer_size size of the write buffer

void * buffer pointer to the source buffer

Description

This function implements the WriteBlocks service of the EFI_BLOCK_IO_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_disk_flush_blocks**(struct efi_block_io * *this*)

flushes modified data to the device

Parameters

struct efi_block_io * this pointer to the BLOCK_IO_PROTOCOL

Description

This function implements the FlushBlocks service of the EFI_BLOCK_IO_PROTOCOL.

As we always write synchronously nothing is done here.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

struct efi_simple_file_system_protocol * **efi_fs_from_path**(struct efi_device_path * *full_path*)

retrieve simple file system protocol

Parameters

struct efi_device_path * full_path device path including device and file

Description

Gets the simple file system protocol for a file device path.

The full path provided is split into device part and into a file part. The device part is used to find the handle on which the simple file system protocol is installed.

Return

simple file system protocol

int **efi_fs_exists**(struct blk_desc * *desc*, int *part*)
check if a partition bears a file system

Parameters

struct blk_desc * desc block device descriptor

int part partition number

Return

1 if a file system exists on the partition 0 otherwise

efi_status_t **efi_disk_add_dev**(efi_handle_t *parent*, struct efi_device_path * *dp_parent*, const char * *if_typename*, struct blk_desc * *desc*, int *dev_index*, lbaint_t *offset*, unsigned int *part*, struct *efi_disk_obj* ** *disk*)
create a handle for a partition or disk

Parameters

efi_handle_t parent parent handle

struct efi_device_path * dp_parent parent device path

const char * if_typename interface name for block device

struct blk_desc * desc internal block device

int dev_index device index for block device

lbaint_t offset offset into disk for simple partitions

unsigned int part partition

struct efi_disk_obj ** disk pointer to receive the created handle

Return

disk object

int **efi_disk_create_partitions**(efi_handle_t *parent*, struct blk_desc * *desc*, const char * *if_typename*, int *diskid*, const char * *pdevname*)
create handles and protocols for partitions

Parameters

efi_handle_t parent handle of the parent disk

struct blk_desc * desc block device

const char * if_typename interface type

int diskid device number

const char * pdevname device name

Description

Create handles and protocols for the partitions of a block device.

Return

number of partitions created

efi_status_t **efi_disk_register**(void)
register block devices

Parameters

void no arguments

Description

U-Boot doesn't have a list of all online disk devices. So when running our EFI payload, we scan through all of the potentially available ones and store them in our object pool.

This function is called in *efi_init_obj_list()*.

TODO(sjg**chromium.org**): Actually with CONFIG_BLK, U-Boot does have this. Consider converting the code to look up devices as needed. The EFI device could be a child of the UCLASS_BLK block device, perhaps.

Return

status code

bool **efi_disk_is_system_part**(efi_handle_t *handle*)
check if handle refers to an EFI system partition

Parameters

efi_handle_t handle handle of partition

Return

true if handle refers to an EFI system partition

File protocol

int **is_dir**(struct file_handle * *fh*)
check if file handle points to directory

Parameters

struct file_handle * fh file handle

Description

We assume that set_blk_dev(fh) has been called already.

Return

true if file handle points to a directory

int **efi_create_file**(struct file_handle * *fh*, u64 *attributes*)
create file or directory

Parameters

struct file_handle * fh file handle

u64 attributes attributes for newly created file

Return

0 for success

struct efi_file_handle * **file_open**(struct file_system * *fs*, struct file_handle * *parent*, u16 * *file_name*, u64 *open_mode*, u64 *attributes*)
open a file handle

Parameters

struct file_system * fs file system

struct file_handle * parent directory relative to which the file is to be opened

u16 * file_name path of the file to be opened. '', '.', or '..' may be used as modifiers. A leading backslash indicates an absolute path.

u64 open_mode bit mask indicating the access mode (read, write, create)

u64 attributes attributes for newly created file

Return

handle to the opened file or NULL

efi_status_t **efi_get_file_size**(struct file_handle * *fh*, loff_t * *file_size*)
determine the size of a file

Parameters

struct file_handle * fh file handle

loff_t * file_size pointer to receive file size

Return

status code

efi_status_t EFIAPI **efi_file_write**(struct efi_file_handle * *file*, efi_uintn_t * *buffer_size*, void * *buffer*)
write to file

Parameters

struct efi_file_handle * file file handle

efi_uintn_t * buffer_size number of bytes to write

void * buffer buffer with the bytes to write

Description

This function implements the `Write()` service of the EFI_FILE_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_file_getpos**(struct efi_file_handle * *file*, u64 * *pos*)
get current position in file

Parameters

struct efi_file_handle * file file handle

u64 * pos pointer to file position

Description

This function implements the GetPosition service of the EFI file protocol. See the UEFI spec for details.

Return

status code

efi_status_t EFIAPI **efi_file_setpos**(struct efi_file_handle * *file*, u64 *pos*)
set current position in file

Parameters

struct efi_file_handle * file file handle

u64 pos new file position

Description

This function implements the SetPosition service of the EFI file protocol. See the UEFI spec for details.

Return

status code

struct efi_file_handle * **efi_file_from_path**(struct efi_device_path * *fp*)
open file via device path

Parameters

struct efi_device_path * fp device path

Return

EFI_FILE_PROTOCOL for the file or NULL

Graphical output protocol

struct **efi_gop_obj**
graphical output protocol object

Definition

```
struct efi_gop_obj {
  struct efi_object header;
  struct efi_gop ops;
  struct efi_gop_mode_info info;
  struct efi_gop_mode mode;
  u32 bpix;
  void *fb;
};
```

Members

header EFI object header

ops graphical output protocol interface

info graphical output mode information

mode graphical output mode

bpix bits per pixel

fb frame buffer

efi_status_t EFIAPI **gop_set_mode**(struct efi_gop * *this*, u32 *mode_number*)
set graphical output mode

Parameters

struct efi_gop * this the graphical output protocol

u32 mode_number the mode to be set

Description

This function implements the `SetMode()` service.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

Load file 2 protocol

The load file 2 protocol can be used by the Linux kernel to load the initial RAM disk. U-Boot can be configured to provide an implementation.

loff_t **get_file_size**(const char * *dev*, const char * *part*, const char * *file*, efi_status_t * *status*)
retrieve the size of initramfs, set efi status on error

Parameters

const char * dev device to read from, e.g. "mmc"

const char * part device partition, e.g. "0:1"

const char * file name of file

efi_status_t * status EFI exit code in case of failure

Return

size of file

efi_status_t EFIAPI **efi_load_file2_initrd**(struct efi_load_file_protocol * *this*, struct efi_device_path * *file_path*, bool *boot_policy*, efi_uintn_t * *buffer_size*, void * *buffer*)
load initial RAM disk

Parameters

struct efi_load_file_protocol * this EFI_LOAD_FILE2_PROTOCOL instance

struct efi_device_path * file_path media device path of the file, "" in this case

bool boot_policy must be false

efi_uintn_t * buffer_size size of allocated buffer

void * buffer buffer to load the file

Description

This function implements the LoadFile service of the EFI_LOAD_FILE2_PROTOCOL in order to load an initial RAM disk requested by the Linux kernel stub.

See the UEFI spec for details.

Return

status code

efi_status_t **efi_initrd_register**(void)
create handle for loading initial RAM disk

Parameters

void no arguments

Description

This function creates a new handle and installs a Linux specific vendor device path and an EFI_LOAD_FILE_2_PROTOCOL. Linux uses the device path to identify the handle and then calls the LoadFile service of the EFI_LOAD_FILE_2_PROTOCOL to read the initial RAM disk.

Return

status code

Network protocols

struct **efi_net_obj**
EFI object representing a network interface

Definition

```
struct efi_net_obj {
  struct efi_object header;
  struct efi_simple_network net;
  struct efi_simple_network_mode net_mode;
  struct efi_pxe_base_code_protocol pxe;
  struct efi_pxe_mode pxe_mode;
};
```

Members

header EFI object header

net simple network protocol interface

net_mode status of the network interface

pxe PXE base code protocol interface

pxe_mode status of the PXE base code protocol

efi_status_t EFIAPI **efi_net_nvdata**(struct efi_simple_network * *this*, int *read_write*, ulong *offset*, ulong *buffer_size*, char * *buffer*)
read or write NVRAM

Parameters

struct efi_simple_network * this the instance of the Simple Network Protocol

int read_write true for read, false for write

ulong offset offset in NVRAM

ulong buffer_size size of buffer

char * buffer buffer

Description

This function implements the GetStatus service of the Simple Network Protocol. See the UEFI spec for details.

Return

status code

efi_status_t EFIAPI **efi_net_get_status**(struct efi_simple_network * *this*, u32 * *int_status*, void ** *txbuf*)
get interrupt status

Parameters

struct efi_simple_network * this the instance of the Simple Network Protocol

u32 * int_status interface status

void ** txbuf transmission buffer

Description

This function implements the GetStatus service of the Simple Network Protocol. See the UEFI spec for details.

efi_status_t EFIAPI **efi_net_transmit**(struct efi_simple_network * *this*, size_t *header_size*, size_t *buffer_size*, void * *buffer*, struct efi_mac_address * *src_addr*, struct efi_mac_address * *dest_addr*, u16 * *protocol*)

transmit a packet

Parameters

struct efi_simple_network * this the instance of the Simple Network Protocol

size_t header_size size of the media header

size_t buffer_size size of the buffer to receive the packet

void * buffer buffer to receive the packet

struct efi_mac_address * src_addr source hardware MAC address

struct efi_mac_address * dest_addr destination hardware MAC address

u16 * protocol type of header to build

Description

This function implements the Transmit service of the Simple Network Protocol. See the UEFI spec for details.

Return

status code

efi_status_t EFIAPI **efi_net_receive**(struct efi_simple_network * *this*, size_t * *header_size*, size_t * *buffer_size*, void * *buffer*, struct efi_mac_address * *src_addr*, struct efi_mac_address * *dest_addr*, u16 * *protocol*)

receive a packet from a network interface

Parameters

struct efi_simple_network * this the instance of the Simple Network Protocol

size_t * header_size size of the media header

size_t * buffer_size size of the buffer to receive the packet

void * buffer buffer to receive the packet

struct efi_mac_address * src_addr source MAC address

struct efi_mac_address * dest_addr destination MAC address

u16 * protocol protocol

Description

This function implements the Receive service of the Simple Network Protocol. See the UEFI spec for details.

Return

status code

void **efi_net_set_dhcp_ack**(void * *pkt*, int *len*)

take note of a selected DHCP IP address

Parameters

void * pkt packet received by `dhcp_handler()`

int len length of the packet received

Description

This function is called by `dhcp_handler()`.

void **efi_net_push**(void * *pkt*, int *len*)
callback for received network packet

Parameters

void * pkt network packet

int len length

Description

This function is called when a network packet is received by `eth_rx()`.

void EFIAPI **efi_network_timer_notify**(struct efi_event * *event*, void * *context*)
check if a new network packet has been received

Parameters

struct efi_event * event the event for which this notification function is registered

void * context event context - not used in this function

Description

This notification function is called in every timer cycle.

efi_status_t **efi_net_register**(void)
register the simple network protocol

Parameters

void no arguments

Description

This gets called from *do_bootefi_exec()*.

Random number generator protocol

efi_status_t **platform_get_rng_device**(struct udevice ** *dev*)
retrieve random number generator

Parameters

struct udevice ** dev udevice

Description

This function retrieves the udevice implementing a hardware random number generator.

This function may be overridden if special initialization is needed.

Return

status code

efi_status_t EFIAPI **rng_getinfo**(struct efi_rng_protocol * *this*, efi_uintn_t * *rng_algorithm_list_size*, efi_guid_t * *rng_algorithm_list*)
get information about random number generation

Parameters

struct efi_rng_protocol * this random number generator protocol instance

efi_uintn_t * rng_algorithm_list_size number of random number generation algorithms

efi_guid_t * rng_algorithm_list descriptions of random number generation algorithms

Description

This function implement the `GetInfo()` service of the EFI random number generator protocol. See the UEFI spec for details.

Return

status code

efi_status_t EFIAPI **getrng**(struct efi_rng_protocol * *this*, efi_guid_t * *rng_algorithm*, efi_uintn_t *rng_value_length*, uint8_t * *rng_value*)
get random value

Parameters

struct efi_rng_protocol * this random number generator protocol instance

efi_guid_t * rng_algorithm random number generation algorithm

efi_uintn_t rng_value_length number of random bytes to generate, buffer length

uint8_t * rng_value buffer to receive random bytes

Description

This function implement the `GetRng()` service of the EFI random number generator protocol. See the UEFI spec for details.

Return

status code

efi_status_t **efi_rng_register**(void)
register EFI_RNG_PROTOCOL

Parameters

void no arguments

Description

If a RNG device is available, the Random Number Generator Protocol is registered.

Return

An error status is only returned if adding the protocol fails.

Text IO protocols

int **term_read_reply**(int * *n*, int *num*, char *end_char*)

Parameters

int * n array of return values

int num number of return values expected

char end_char character indicating end of terminal message

Return

non-zero indicates error

efi_status_t EFIAPI **efi_cout_output_string**(struct efi_simple_text_output_protocol * *this*, const efi_string_t *string*)
write Unicode string to console

Parameters

struct efi_simple_text_output_protocol * this simple text output protocol

const efi_string_t string u16 string

Description

This function implements the OutputString service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_cout_test_string**(struct efi_simple_text_output_protocol * *this*, const efi_string_t *string*)
test writing Unicode string to console

Parameters

struct efi_simple_text_output_protocol * this simple text output protocol

const efi_string_t string u16 string

Description

This function implements the TestString service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

As in OutputString we simply convert UTF-16 to UTF-8 there are no unsupported code points and we can always return EFI_SUCCESS.

Return

status code

bool **cout_mode_matches**(struct cout_mode * *mode*, int *rows*, int *cols*)
check if mode has given terminal size

Parameters

struct cout_mode * mode text mode

int rows number of rows

int cols number of columns

Return

true if number of rows and columns matches the mode and the mode is present

int **query_console_serial**(int * *rows*, int * *cols*)
query console size

Parameters

int * rows pointer to return number of rows

int * cols pointer to return number of columns

Description

When using a serial console or the net console we can only devise the terminal size by querying the terminal using ECMA-48 control sequences.

Return

0 on success

void **query_console_size**(void)
update the mode table.

Parameters

void no arguments

Description

By default the only mode available is 80x25. If the console has at least 50 lines, enable mode 80x50. If we can query the console size and it is neither 80x25 nor 80x50, set it as an additional mode.

efi_status_t EFIAPI **efi_cout_query_mode**(struct efi_simple_text_output_protocol * *this*, unsigned long *mode_number*, unsigned long * *columns*, unsigned long * *rows*)

get terminal size for a text mode

Parameters

struct efi_simple_text_output_protocol * this simple text output protocol

unsigned long mode_number mode number to retrieve information on

unsigned long * columns number of columns

unsigned long * rows number of rows

Description

This function implements the QueryMode service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_cout_set_attribute**(struct efi_simple_text_output_protocol * *this*, unsigned long *attribute*)

set fore- and background color

Parameters

struct efi_simple_text_output_protocol * this simple text output protocol

unsigned long attribute foreground color - bits 0-3, background color - bits 4-6

Description

This function implements the SetAttribute service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_cout_clear_screen**(struct efi_simple_text_output_protocol * *this*)

clear screen

Parameters

struct efi_simple_text_output_protocol * this pointer to the protocol instance

Description

This function implements the ClearScreen service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_cout_set_mode**(struct efi_simple_text_output_protocol * *this*, unsigned long *mode_number*)

set text model

Parameters

struct efi_simple_text_output_protocol * this pointer to the protocol instance

unsigned long mode_number number of the text mode to set

Description

This function implements the SetMode service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_cout_reset**(struct efi_simple_text_output_protocol * *this*, char *extended_verification*)

reset the terminal

Parameters

struct efi_simple_text_output_protocol * this pointer to the protocol instance

char extended_verification if set an extended verification may be executed

Description

This function implements the Reset service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_cout_set_cursor_position**(struct efi_simple_text_output_protocol * *this*, unsigned long *column*, unsigned long *row*)

reset the terminal

Parameters

struct efi_simple_text_output_protocol * this pointer to the protocol instance

unsigned long column column to move to

unsigned long row row to move to

Description

This function implements the SetCursorPosition service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

efi_status_t EFIAPI **efi_cout_enable_cursor**(struct efi_simple_text_output_protocol * *this*, bool *enable*)

enable the cursor

Parameters

struct efi_simple_text_output_protocol * this pointer to the protocol instance

bool enable if true enable, if false disable the cursor

Description

This function implements the EnableCursor service of the simple text output protocol. See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

status code

struct **efi_cin_notify_function**

registered console input notify function

Definition

```
struct efi_cin_notify_function {
  struct list_head link;
  struct efi_key_data key;
```

(continues on next page)

(continued from previous page)

```
  efi_status_t (EFIAPI *function) (struct efi_key_data *key_data);
};
```

Members

link link to list

key key to notify

function function to call

void **set_shift_mask**(int *mod*, struct efi_key_state * *key_state*)
set shift mask

Parameters

int mod Xterm shift mask

struct efi_key_state * key_state receives the state of the shift, alt, control, and logo keys

int **analyze_modifiers**(struct efi_key_state * *key_state*)
analyze modifiers (shift, alt, ctrl) for function keys

Parameters

struct efi_key_state * key_state receives the state of the shift, alt, control, and logo keys

Description

This gets called when we have already parsed CSI.

Return

the unmodified code

efi_status_t **efi_cin_read_key**(struct efi_key_data * *key*)
read a key from the console input

Parameters

struct efi_key_data * key

- key received

Return

- status code

void **efi_cin_notify**(void)
notify registered functions

Parameters

void no arguments

void **efi_cin_check**(void)
check if keyboard input is available

Parameters

void no arguments

void **efi_cin_empty_buffer**(void)
empty input buffer

Parameters

void no arguments

efi_status_t EFIAPI **efi_cin_reset_ex**(struct efi_simple_text_input_ex_protocol * *this*, bool *extended_verification*)
reset console input

Parameters

struct efi_simple_text_input_ex_protocol * this

- the extended simple text input protocol

bool extended_verification

- extended verification

Description

This function implements the reset service of the EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

old value of the task priority level

efi_status_t EFIAPI **efi_cin_read_key_stroke_ex**(struct efi_simple_text_input_ex_protocol * *this*, struct efi_key_data * *key_data*)

read key stroke

Parameters

struct efi_simple_text_input_ex_protocol * this instance of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL

struct efi_key_data * key_data key read from console

Return

status code

This function implements the ReadKeyStrokeEx service of the EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t EFIAPI **efi_cin_set_state**(struct efi_simple_text_input_ex_protocol * *this*, u8 * *key_toggle_state*)

set toggle key state

Parameters

struct efi_simple_text_input_ex_protocol * this instance of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL

u8 * key_toggle_state pointer to key toggle state

Return

status code

This function implements the SetState service of the EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t EFIAPI **efi_cin_register_key_notify**(struct efi_simple_text_input_ex_protocol * *this*, struct efi_key_data * *key_data*, efi_status_t (EFIAPI *key_notify_function)(struct efi_key_data *key_data) *key_notify_function*, void ** *notify_handle*)

register key notification function

Parameters

struct efi_simple_text_input_ex_protocol * this instance of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL

struct efi_key_data * key_data key to be notified

efi_status_t (EFIAPI *key_notify_function)(struct efi_key_data *key_data) key_notify_functio
function to be called if the key is pressed

void ** notify_handle handle for unregistering the notification

Return

status code

This function implements the SetState service of the EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t EFIAPI **efi_cin_unregister_key_notify**(struct efi_simple_text_input_ex_protocol * *this*, void * *notification_handle*)
unregister key notification function

Parameters

struct efi_simple_text_input_ex_protocol * this instance of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL

void * notification_handle handle received when registering

Return

status code

This function implements the SetState service of the EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t EFIAPI **efi_cin_reset**(struct efi_simple_text_input_protocol * *this*, bool *extended_verification*)
drain the input buffer

Parameters

struct efi_simple_text_input_protocol * this instance of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL

bool extended_verification allow for exhaustive verification

Return

status code

This function implements the Reset service of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

efi_status_t EFIAPI **efi_cin_read_key_stroke**(struct efi_simple_text_input_protocol * *this*, struct efi_input_key * *key*)
read key stroke

Parameters

struct efi_simple_text_input_protocol * this instance of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL

struct efi_input_key * key key read from console

Return

status code

This function implements the ReadKeyStroke service of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

void EFIAPI **efi_key_notify**(struct efi_event * *event*, void * *context*)
notify the wait for key event

Parameters

struct efi_event * event wait for key event

void * context not used

efi_status_t **efi_console_register**(void)
install the console protocols

Parameters

void no arguments

Description

This function is called from *do_bootefi_exec()*.

Return

status code

Unicode Collation protocol

efi_intn_t EFIAPI **efi_stri_coll**(struct efi_unicode_collation_protocol * *this*, u16 * *s1*, u16 * *s2*)
compare utf-16 strings case-insenitively

Parameters

struct efi_unicode_collation_protocol * this unicode collation protocol instance

u16 * s1 first string

u16 * s2 second string

Description

This function implements the `StriColl()` service of the EFI_UNICODE_COLLATION_PROTOCOL.

See the Unified Extensible Firmware Interface (UEFI) specification for details.

Return

0: s1 == s2, > 0: s1 > s2, < 0: s1 < s2

s32 **next_lower**(const u16 ** *string*)
get next codepoint converted to lower case

Parameters

const u16 ** string pointer to u16 string, on return advanced by one codepoint

Return

first codepoint of string converted to lower case

bool **metai_match**(const u16 * *string*, const u16 * *pattern*)
compare utf-16 string with a pattern string case-insenitively

Parameters

const u16 * string string to compare

const u16 * pattern pattern string

Description

The pattern string may use these:

- **–** matches >= 0 characters
- ? matches 1 character
- [<char1><char2>…<charN>] match any character in the set
- [<char1>-<char2>] matches any character in the range

This function is called my *efi_metai_match()*.

For '*' pattern searches this function calls itself recursively. Performance-wise this is suboptimal, especially for multiple '*' wildcards. But it results in simple code.

Return

true if the string is matched.

bool EFIAPI **efi_metai_match**(struct efi_unicode_collation_protocol * *this*, const u16 * *string*, const u16 * *pattern*)
compare utf-16 string with a pattern string case-insenitively

Parameters

struct efi_unicode_collation_protocol * this unicode collation protocol instance

const u16 * string string to compare

const u16 * pattern pattern string

Description

The pattern string may use these:

- **-** matches >= 0 characters
- ? matches 1 character
- [<char1><char2>…<charN>] match any character in the set
- [<char1>-<char2>] matches any character in the range

This function implements the `MetaMatch()` service of the EFI_UNICODE_COLLATION_PROTOCOL.

Return

true if the string is matched.

void EFIAPI **efi_str_lwr**(struct efi_unicode_collation_protocol * *this*, u16 * *string*)
convert to lower case

Parameters

struct efi_unicode_collation_protocol * this unicode collation protocol instance

u16 * string string to convert

Description

The conversion is done in place. As long as upper and lower letters use the same number of words this does not pose a problem.

This function implements the `StrLwr()` service of the EFI_UNICODE_COLLATION_PROTOCOL.

void EFIAPI **efi_str_upr**(struct efi_unicode_collation_protocol * *this*, u16 * *string*)
convert to upper case

Parameters

struct efi_unicode_collation_protocol * this unicode collation protocol instance

u16 * string string to convert

Description

The conversion is done in place. As long as upper and lower letters use the same number of words this does not pose a problem.

This function implements the `StrUpr()` service of the EFI_UNICODE_COLLATION_PROTOCOL.

void EFIAPI **efi_fat_to_str**(struct efi_unicode_collation_protocol * *this*, efi_uintn_t *fat_size*, char * *fat*, u16 * *string*)
convert an 8.3 file name from an OEM codepage to Unicode

Parameters

struct efi_unicode_collation_protocol * this unicode collation protocol instance

efi_uintn_t fat_size size of the string to convert

char * fat string to convert

u16 * string converted string

Description

This function implements the `FatToStr()` service of the EFI_UNICODE_COLLATION_PROTOCOL.

bool EFIAPI **efi_str_to_fat**(struct efi_unicode_collation_protocol * *this*, const u16 * *string*, efi_uintn_t *fat_size*, char * *fat*)

convert a utf-16 string to legal characters for a FAT file name in an OEM code page

Parameters

struct efi_unicode_collation_protocol * this unicode collation protocol instance

const u16 * string Unicode string to convert

efi_uintn_t fat_size size of the target buffer

char * fat converted string

Description

This function implements the `StrToFat()` service of the EFI_UNICODE_COLLATION_PROTOCOL.

Return

true if an illegal character was substituted by '_'.

Unit testing

The following library functions are provided to support writing UEFI unit tests. The should not be used elsewhere.

efi_st_printf(*...*)

print a message

Parameters

... format string followed by fields to print

efi_st_error(*...*)

prints an error message

Parameters

... format string followed by fields to print

efi_st_todo(*...*)

prints a TODO message

Parameters

... format string followed by fields to print

enum **efi_test_phase**

phase when test will be executed

Constants

EFI_EXECUTE_BEFORE_BOOTTIME_EXIT

- execute before ExitBootServices

Setup, execute, and teardown are executed before `ExitBootServices()`.

EFI_SETUP_BEFORE_BOOTTIME_EXIT

- setup before ExitBootServices

Setup is executed before `ExitBootServices()` while execute, and teardown are executed after `ExitBootServices()`.

EFI_SETUP_AFTER_BOOTTIME_EXIT

- setup after ExitBootServices

  Setup, execute, and teardown are executed after `ExitBootServices()`.

Description

A test may be setup and executed at boottime, it may be setup at boottime and executed at runtime, or it may be setup and executed at runtime.

void **efi_st_exit_boot_services**(void)
  exit the boot services

Parameters

void no arguments

Description

- The size of the memory map is determined.
- Pool memory is allocated to copy the memory map.
- The memory map is copied and the map key is obtained.
- The map key is used to exit the boot services.

void **efi_st_printc**(int *color*, const char * *fmt*, ...)
  print a colored message

Parameters

int color color, see constants in efi_api.h, use -1 for no color

const char * fmt printf style format string

... arguments to be printed

u16 * **efi_st_translate_char**(u16 *code*)
  translate a Unicode character to a string

Parameters

u16 code Unicode character

Return

string

u16 * **efi_st_translate_code**(u16 *code*)
  translate a scan code to a human readable string

Parameters

u16 code scan code

Description

This function translates the scan code returned by the simple text input protocol to a human readable string, e.g. 0x04 is translated to L"Left".

Return

Unicode string

int **efi_st_strcmp_16_8**(const u16 * *buf1*, const char * *buf2*)
  compare an u16 string to a char string

Parameters

const u16 * buf1 u16 string

const char * buf2 char string

Description

This function compares each u16 value to the char value at the same position. This function is only useful for ANSI strings.

Return

0 if both buffers contain equivalent strings

u16 **efi_st_get_key**(void)
reads an Unicode character from the input device

Parameters

void no arguments

Return

Unicode character

struct **efi_unit_test**
EFI unit test

Definition

```
struct efi_unit_test {
  const char *name;
  const enum efi_test_phase phase;
  int (*setup)(const efi_handle_t handle, const struct efi_system_table *systable);
  int (*execute)(void);
  int (*teardown)(void);
  bool on_request;
};
```

Members

name name of the unit test used in the user interface

phase specifies when setup and execute are executed

setup set up function of the unit test

execute execute function of the unit test

teardown tear down function of the unit test

on_request flag indicating that the test shall only be executed on request

Description

The *struct efi_unit_test* structure provides a interface to an EFI unit test.

EFI_UNIT_TEST(*__name*)
macro to declare a new EFI unit test

Parameters

__name string identifying the unit test in the linker generated list

Description

The macro *EFI_UNIT_TEST()* declares an EFI unit test using the *struct efi_unit_test* structure. The test is added to a linker generated list which is evaluated by the 'bootefi selftest' command.

5.1.3 Option Parsing

struct **getopt_state**
Saved state across *getopt()* calls

Definition

```
struct getopt_state {
  int index;
  int opt;
  char *arg;
};
```

Members

index Index of the next unparsed argument of **argv**. If *getopt()* has parsed all of **argv**, then **index** will equal **argc**.

opt Option being parsed when an error occurs. **opt** is only valid when *getopt()* returns ? or :.

arg The argument to an option, NULL if there is none. **arg** is only valid when *getopt()* returns an option character.

void **getopt_init_state**(struct *getopt_state* * *gs*)
Initialize a *struct getopt_state*

Parameters

struct getopt_state * gs The state to initialize

Description

This must be called before using **gs** with *getopt()*.

int **getopt**(struct *getopt_state* * *gs*, int *argc*, char *const *argv*, const char * *optstring*)
Parse short command-line options

Parameters

struct getopt_state * gs Internal state and out-of-band return arguments. This must be initialized with getopt_init_context() beforehand.

int argc Number of arguments, not including the NULL terminator

char *const argv Argument list, terminated by NULL

const char * optstring Option specification, as described below

Description

getopt() parses short options. Short options are single characters. They may be followed by a required argument or an optional argument. Arguments to options may occur in the same argument as an option (like -larg), or in the following argument (like -l arg). An argument containing options begins with a -. If an option expects no arguments, then it may be immediately followed by another option (like ls -alR).

optstring is a list of accepted options. If an option is followed by : in **optstring**, then it expects a mandatory argument. If an option is followed by :: in **optstring**, it expects an optional argument. **gs.arg** points to the argument, if one is parsed.

getopt() stops parsing options when it encounters the first non-option argument, when it encounters the argument --, or when it runs out of arguments. For example, in ls -l foo -R, option parsing will stop when *getopt()* encounters foo, if l does not expect an argument. However, the whole list of arguments would be parsed if l expects an argument.

An example invocation of *getopt()* might look like:

```
char *argv[] = { "program", "-cbx", "-a", "foo", "bar", 0 };
int opt, argc = ARRAY_SIZE(argv) - 1;
struct getopt_state gs;

getopt_init_state(&gs);
while ((opt = getopt(&gs, argc, argv, "a::b:c")) != -1)
```

(continues on next page)

(continued from previous page)

```
    printf("opt = %c, index = %d, arg = \"%s\"\n", opt, gs.index, gs.arg);
printf("%d argument(s) left\n", argc - gs.index);
```

and would produce an output of:

```
opt = c, index = 1, arg = "<NULL>"
opt = b, index = 2, arg = "x"
opt = a, index = 4, arg = "foo"
1 argument(s) left
```

For further information, refer to the getopt(3) man page.

Return

- An option character if an option is found. **gs.arg** is set to the argument if there is one, otherwise it is set to `NULL`.
- `-1` if there are no more options, if a non-option argument is encountered, or if an `--` argument is encountered.
- `'?'` if we encounter an option not in **optstring**. **gs.opt** is set to the unknown option.
- `':'` if an argument is required, but no argument follows the option. **gs.opt** is set to the option missing its argument.

gs.index is always set to the index of the next unparsed argument in **argv**.

int **getopt_silent**(struct *getopt_state * gs*, int *argc*, char *const *argv*, const char * *optstring*)
Parse short command-line options silently

Parameters

`struct getopt_state * gs` State

`int argc` Argument count

`char *const argv` Argument list

`const char * optstring` Option specification

Description

Same as *`getopt()`*, except no error messages are printed.

5.1.4 Linker-Generated Arrays

A linker list is constructed by grouping together linker input sections, each containing one entry of the list. Each input section contains a constant initialized variable which holds the entry's content. Linker list input sections are constructed from the list and entry names, plus a prefix which allows grouping all lists together. Assuming _list and _entry are the list and entry names, then the corresponding input section name is

```
.u_boot_list_ + 2_ + @_list + _2_ + @_entry
```

and the C variable name is

```
_u_boot_list + _2_ + @_list + _2_ + @_entry
```

This ensures uniqueness for both input section and C variable name.

Note that the names differ only in the first character, "." for the section and "_" for the variable, so that the linker cannot confuse section and symbol names. From now on, both names will be referred to as

```
%u_boot_list_ + 2_ + @_list + _2_ + @_entry
```

Entry variables need never be referred to directly.

The naming scheme for input sections allows grouping all linker lists into a single linker output section and grouping all entries for a single list.

Note the two '_2_' constant components in the names: their presence allows putting a start and end symbols around a list, by mapping these symbols to sections names with components "1" (before) and "3" (after) instead of "2" (within). Start and end symbols for a list can generally be defined as

```
%u_boot_list_2_ + @_list + _1_...
%u_boot_list_2_ + @_list + _3_...
```

Start and end symbols for the whole of the linker lists area can be defined as

```
%u_boot_list_1_...
%u_boot_list_3_...
```

Here is an example of the sorted sections which result from a list "array" made up of three entries : "first", "second" and "third", iterated at least once.

```
.u_boot_list_2_array_1
.u_boot_list_2_array_2_first
.u_boot_list_2_array_2_second
.u_boot_list_2_array_2_third
.u_boot_list_2_array_3
```

If lists must be divided into sublists (e.g. for iterating only on part of a list), one can simply give the list a name of the form 'outer_2_inner', where 'outer' is the global list name and 'inner' is the sub-list name. Iterators for the whole list should use the global list name ("outer"); iterators for only a sub-list should use the full sub-list name ("outer_2_inner").

Here is an example of the sections generated from a global list named "drivers", two sub-lists named "i2c" and "pci", and iterators defined for the whole list and each sub-list:

```
%u_boot_list_2_drivers_1
%u_boot_list_2_drivers_2_i2c_1
%u_boot_list_2_drivers_2_i2c_2_first
%u_boot_list_2_drivers_2_i2c_2_first
%u_boot_list_2_drivers_2_i2c_2_second
%u_boot_list_2_drivers_2_i2c_2_third
%u_boot_list_2_drivers_2_i2c_3
%u_boot_list_2_drivers_2_pci_1
%u_boot_list_2_drivers_2_pci_2_first
%u_boot_list_2_drivers_2_pci_2_second
%u_boot_list_2_drivers_2_pci_2_third
%u_boot_list_2_drivers_2_pci_3
%u_boot_list_2_drivers_3
```

llsym(*_type*, *_name*, *_list*)
    Access a linker-generated array entry

Parameters

_type Data type of the entry

_name Name of the entry

_list name of the list. Should contain only characters allowed in a C variable name!

ll_entry_declare(*_type*, *_name*, *_list*)
Declare linker-generated array entry

Parameters

_type Data type of the entry

_name Name of the entry

_list name of the list. Should contain only characters allowed in a C variable name!

Description

This macro declares a variable that is placed into a linker-generated array. This is a basic building block for more advanced use of linker- generated arrays. The user is expected to build their own macro wrapper around this one.

A variable declared using this macro must be compile-time initialized.

Special precaution must be made when using this macro:

1) The _type must not contain the "static" keyword, otherwise the entry is generated and can be iterated but is listed in the map file and cannot be retrieved by name.
2) In case a section is declared that contains some array elements AND a subsection of this section is declared and contains some elements, it is imperative that the elements are of the same type.
3) In case an outer section is declared that contains some array elements AND an inner subsection of this section is declared and contains some elements, then when traversing the outer section, even the elements of the inner sections are present in the array.

Example

```
ll_entry_declare(struct my_sub_cmd, my_sub_cmd, cmd_sub) = {
        .x = 3,
        .y = 4,
};
```

ll_entry_declare_list(*_type*, *_name*, *_list*)
Declare a list of link-generated array entries

Parameters

_type Data type of each entry

_name Name of the entry

_list name of the list. Should contain only characters allowed in a C variable name!

Description

This is like *ll_entry_declare()* but creates multiple entries. It should be assigned to an array.

```
ll_entry_declare_list(struct my_sub_cmd, my_sub_cmd, cmd_sub) = {
     { .x = 3, .y = 4 },
     { .x = 8, .y = 2 },
     { .x = 1, .y = 7 }
};
```

ll_entry_start(*_type*, *_list*)
Point to first entry of linker-generated array

Parameters

_type Data type of the entry

_list Name of the list in which this entry is placed

Description

This function returns (_type *) pointer to the very first entry of a linker-generated array placed into subsection of .u_boot_list section specified by _list argument.

Since this macro defines an array start symbol, its leftmost index must be 2 and its rightmost index must be 1.

Example

```
struct my_sub_cmd *msc = ll_entry_start(struct my_sub_cmd, cmd_sub);
```

ll_entry_end(*_type*, *_list*)
Point after last entry of linker-generated array

Parameters

_type Data type of the entry

_list Name of the list in which this entry is placed (with underscores instead of dots)

Description

This function returns (_type *) pointer after the very last entry of a linker-generated array placed into subsection of .u_boot_list section specified by _list argument.

Since this macro defines an array end symbol, its leftmost index must be 2 and its rightmost index must be 3.

Example

```
struct my_sub_cmd *msc = ll_entry_end(struct my_sub_cmd, cmd_sub);
```

ll_entry_count(*_type*, *_list*)
Return the number of elements in linker-generated array

Parameters

_type Data type of the entry

_list Name of the list of which the number of elements is computed

Description

This function returns the number of elements of a linker-generated array placed into subsection of .u_boot_list section specified by _list argument. The result is of an unsigned int type.

Example

```
int i;
const unsigned int count = ll_entry_count(struct my_sub_cmd, cmd_sub);
struct my_sub_cmd *msc = ll_entry_start(struct my_sub_cmd, cmd_sub);
for (i = 0; i < count; i++, msc++)
        printf("Entry ``i``, x=``i`` y=``i``\n", i, msc->x, msc->y);
```

ll_entry_get(*_type*, *_name*, *_list*)
Retrieve entry from linker-generated array by name

Parameters

_type Data type of the entry

_name Name of the entry

_list Name of the list in which this entry is placed

Description

This function returns a pointer to a particular entry in linker-generated array identified by the subsection of u_boot_list where the entry resides and it's name.

Example

```
ll_entry_declare(struct my_sub_cmd, my_sub_cmd, cmd_sub) = {
        .x = 3,
        .y = 4,
};
...
struct my_sub_cmd *c = ll_entry_get(struct my_sub_cmd, my_sub_cmd, cmd_sub);
```

ll_start(*_type*)
Point to first entry of first linker-generated array

Parameters

_type Data type of the entry

Description

This function returns (`_type *`) pointer to the very first entry of the very first linker-generated array.

Since this macro defines the start of the linker-generated arrays, its leftmost index must be 1.

Example

```
struct my_sub_cmd *msc = ll_start(struct my_sub_cmd);
```

ll_end(*_type*)
Point after last entry of last linker-generated array

Parameters

_type Data type of the entry

Description

This function returns (`_type *`) pointer after the very last entry of the very last linker-generated array.

Since this macro defines the end of the linker-generated arrays, its leftmost index must be 3.

Example

```
struct my_sub_cmd *msc = ll_end(struct my_sub_cmd);
```

5.1.5 Pinctrl and Pinmux

struct **pinconf_param**
pin config parameters

Definition

```
struct pinconf_param {
  const char * const property;
  unsigned int param;
  u32 default_value;
};
```

Members

property Property name in DT nodes

param ID for this config parameter

default_value default value for this config parameter used in case no value is specified in DT nodes

struct **pinctrl_ops**
pin control operations, to be implemented by pin controller drivers.

Definition

```
struct pinctrl_ops {
  int (*get_pins_count)(struct udevice *dev);
  const char *(*get_pin_name)(struct udevice *dev, unsigned selector);
  int (*get_groups_count)(struct udevice *dev);
  const char *(*get_group_name)(struct udevice *dev, unsigned selector);
  int (*get_functions_count)(struct udevice *dev);
  const char *(*get_function_name)(struct udevice *dev, unsigned selector);
  int (*pinmux_set)(struct udevice *dev, unsigned pin_selector, unsigned func_
→selector);
  int (*pinmux_group_set)(struct udevice *dev, unsigned group_selector, unsigned func_
→selector);
  int (*pinmux_property_set)(struct udevice *dev, u32 pinmux_group);
  unsigned int pinconf_num_params;
  const struct pinconf_param *pinconf_params;
  int (*pinconf_set)(struct udevice *dev, unsigned pin_selector, unsigned param,␣
→unsigned argument);
  int (*pinconf_group_set)(struct udevice *dev, unsigned group_selector, unsigned␣
→param, unsigned argument);
  int (*set_state)(struct udevice *dev, struct udevice *config);
  int (*set_state_simple)(struct udevice *dev, struct udevice *periph);
  int (*request)(struct udevice *dev, int func, int flags);
  int (*get_periph_id)(struct udevice *dev, struct udevice *periph);
  int (*get_gpio_mux)(struct udevice *dev, int banknum, int index);
  int (*get_pin_muxing)(struct udevice *dev, unsigned int selector, char *buf, int␣
→size);
  int (*gpio_request_enable)(struct udevice *dev, unsigned int selector);
  int (*gpio_disable_free)(struct udevice *dev, unsigned int selector);
};
```

Members

get_pins_count Get the number of selectable pins

> **dev**: Pinctrl device to use
>
> This function is necessary to parse the "pins" property in DTS.
>
> **Return:** number of selectable named pins available in this driver

get_pin_name Get the name of a pin

> **dev**: Pinctrl device of the pin
>
> **selector**: The pin selector
>
> This function is called by the core to figure out which pin it will do operations to. This function is necessary to parse the "pins" property in DTS.
>
> **Return**: const pointer to the name of the pin

get_groups_count Get the number of selectable groups

> **dev**: Pinctrl device to use
>
> This function is necessary to parse the "groups" property in DTS.
>
> **Return:** number of selectable named groups available in the driver

get_group_name Get the name of a group

> **dev**: Pinctrl device of the group
>
> **selector**: The group selector

This function is called by the core to figure out which group it will do operations to. This function is necessary to parse the “groups” property in DTS.

Return: Pointer to the name of the group

get_functions_count Get the number of selectable functions

dev: Pinctrl device to use

This function is necessary for pin-muxing.

Return: number of selectable named functions available in this driver

get_function_name Get the name of a function

dev: Pinmux device of the function

selector: The function selector

This function is called by the core to figure out which mux setting it will map a certain device to. This function is necessary for pin-muxing.

Return: Pointer to the function name of the muxing selector

pinmux_set Mux a pin to a function

dev: Pinctrl device to use

pin_selector: The pin selector

func_selector: The func selector

On simple controllers one of **pin_selector** or **func_selector** may be ignored. This function is necessary for pin-muxing against a single pin.

Return: 0 if OK, or negative error code on failure

pinmux_group_set Mux a group of pins to a function

dev: Pinctrl device to use

group_selector: The group selector

func_selector: The func selector

On simple controllers one of **group_selector** or **func_selector** may be ignored. This function is necessary for pin-muxing against a group of pins.

Return: 0 if OK, or negative error code on failure

pinmux_property_set Enable a pinmux group

dev: Pinctrl device to use

pinmux_group: A u32 representing the pin identifier and mux settings. The exact format of a pinmux group is left up to the driver.

Mux a single pin to a single function based on a driver-specific pinmux group. This function is necessary for parsing the “pinmux” property in DTS, and for pin-muxing against a pinmux group.

Return: Pin selector for the muxed pin if OK, or negative error code on failure

pinconf_num_params Number of driver-specific parameters to be parsed from device trees. This member is necessary for pin configuration.

pinconf_params List of driver-specific parameters to be parsed from the device tree. This member is necessary for pin configuration.

pinconf_set Configure an individual pin with a parameter

dev: Pinctrl device to use

pin_selector: The pin selector

param: An *enum pin_config_param* from **pinconf_params**

argument: The argument to this param from the device tree, or pinconf_params.default_value

This function is necessary for pin configuration against a single pin.

Return: 0 if OK, or negative error code on failure

`pinconf_group_set` Configure all pins in a group with a parameter

dev: Pinctrl device to use

pin_selector: The group selector

param: A *`enum pin_config_param`* from pinconf_params

argument: The argument to this param from the device tree, or pinconf_params.default_value

This function is necessary for pin configuration against a group of pins.

Return: 0 if OK, or negative error code on failure

`set_state` Configure a pinctrl device

dev: Pinctrl device to use

config: Pseudo device pointing a config node

This function is required to be implemented by all pinctrl drivers. Drivers may set this member to *`pinctrl_generic_set_state()`*, which will call other functions in *`struct pinctrl_ops`* to parse **config**.

Return: 0 if OK, or negative error code on failure

`set_state_simple` Configure a pinctrl device

dev: Pinctrl device to use

config: Pseudo-device pointing a config node

This function is usually a simpler version of `set_state()`. Only the first pinctrl device on the system is supported by this function.

Return: 0 if OK, or negative error code on failure

`request` Request a particular pinctrl function

dev: Device to adjust (`UCLASS_PINCTRL`)

func: Function number (driver-specific)

This activates the selected function.

Return: 0 if OK, or negative error code on failure

`get_periph_id` Get the peripheral ID for a device

dev: Pinctrl device to use for decoding

periph: Device to check

This generally looks at the peripheral's device tree node to work out the peripheral ID. The return value is normally interpreted as `enum periph_id`. so long as this is defined by the platform (which it should be).

Return: Peripheral ID of **periph**, or `-ENOENT` on error

`get_gpio_mux` Get the mux value for a particular GPIO

dev: Pinctrl device to use

banknum: GPIO bank number

index: GPIO index within the bank

This allows the raw mux value for a GPIO to be obtained. It is useful for displaying the function being used by that GPIO, such as with the 'gpio' command. This function is internal to the GPIO subsystem and should not be used by generic code. Typically it is used by a GPIO driver with knowledge of the SoC pinctrl setup.

Return: Mux value (SoC-specific, e.g. 0 for input, 1 for output)

get_pin_muxing Show pin muxing

dev: Pinctrl device to use

selector: Pin selector

buf: Buffer to fill with pin muxing description

size: Size of **buf**

This allows to display the muxing of a given pin. It's useful for debug purposes to know if a pin is configured as GPIO or as an alternate function and which one. Typically it is used by a PINCTRL driver with knowledge of the SoC pinctrl setup.

Return: 0 if OK, or negative error code on failure

gpio_request_enable Request and enable GPIO on a certain pin.

dev: Pinctrl device to use

selector: Pin selector

Implement this only if you can mux every pin individually as GPIO. The affected GPIO range is passed along with an offset(pin number) into that specific GPIO range - function selectors and pin groups are orthogonal to this, the core will however make sure the pins do not collide.

Return: 0 if OK, or negative error code on failure

gpio_disable_free Free up GPIO muxing on a certain pin.

dev: Pinctrl device to use

selector: Pin selector

This function is the reverse of **gpio_request_enable**.

Return: 0 if OK, or negative error code on failure

Description

`set_state()` is the only mandatory operation. You can implement your pinctrl driver with its own **set_state**. In this case, the other callbacks are not required. Otherwise, generic pinctrl framework is also available; use pinctrl_generic_set_state for **set_state**, and implement other operations depending on your necessity.

enum **pin_config_param**
Generic pin configuration parameters

Constants

PIN_CONFIG_BIAS_BUS_HOLD The pin will be set to weakly latch so that it weakly drives the last value on a tristate bus, also known as a "bus holder", "bus keeper" or "repeater". This allows another device on the bus to change the value by driving the bus high or low and switching to tristate. The argument is ignored.

PIN_CONFIG_BIAS_DISABLE Disable any pin bias on the pin, a transition from say pull-up to pull-down implies that you disable pull-up in the process, this setting disables all biasing.

PIN_CONFIG_BIAS_HIGH_IMPEDANCE The pin will be set to a high impedance mode, also know as "third-state" (tristate) or "high-Z" or "floating". On output pins this effectively disconnects the pin, which is useful if for example some other pin is going to drive the signal connected to it for a while. Pins used for input are usually always high impedance.

PIN_CONFIG_BIAS_PULL_DOWN The pin will be pulled down (usually with high impedance to GROUND). If the argument is != 0 pull-down is enabled, if it is 0, pull-down is total, i.e. the pin is connected to GROUND.

PIN_CONFIG_BIAS_PULL_PIN_DEFAULT The pin will be pulled up or down based on embedded knowledge of the controller hardware, like current mux function. The pull direction and possibly strength too will normally be decided completely inside the hardware block and not be readable from the kernel side. If the argument is != 0 pull up/down is enabled, if it is 0, the configuration is ignored. The proper way to disable it is to use **PIN_CONFIG_BIAS_DISABLE**.

PIN_CONFIG_BIAS_PULL_UP The pin will be pulled up (usually with high impedance to VDD). If the argument is != 0 pull-up is enabled, if it is 0, pull-up is total, i.e. the pin is connected to VDD.

PIN_CONFIG_DRIVE_OPEN_DRAIN The pin will be driven with open drain (open collector) which means it is usually wired with other output ports which are then pulled up with an external resistor. Setting this config will enable open drain mode, the argument is ignored.

PIN_CONFIG_DRIVE_OPEN_SOURCE The pin will be driven with open source (open emitter). Setting this config will enable open source mode, the argument is ignored.

PIN_CONFIG_DRIVE_PUSH_PULL The pin will be driven actively high and low, this is the most typical case and is typically achieved with two active transistors on the output. Setting this config will enable push-pull mode, the argument is ignored.

PIN_CONFIG_DRIVE_STRENGTH The pin will sink or source at most the current passed as argument. The argument is in mA.

PIN_CONFIG_DRIVE_STRENGTH_UA The pin will sink or source at most the current passed as argument. The argument is in uA.

PIN_CONFIG_INPUT_DEBOUNCE This will configure the pin to debounce mode, which means it will wait for signals to settle when reading inputs. The argument gives the debounce time in usecs. Setting the argument to zero turns debouncing off.

PIN_CONFIG_INPUT_ENABLE Enable the pin's input. Note that this does not affect the pin's ability to drive output. 1 enables input, 0 disables input.

PIN_CONFIG_INPUT_SCHMITT This will configure an input pin to run in schmitt-trigger mode. If the schmitt-trigger has adjustable hysteresis, the threshold value is given on a custom format as argument when setting pins to this mode.

PIN_CONFIG_INPUT_SCHMITT_ENABLE Control schmitt-trigger mode on the pin. If the argument != 0, schmitt-trigger mode is enabled. If it's 0, schmitt-trigger mode is disabled.

PIN_CONFIG_LOW_POWER_MODE This will configure the pin for low power operation, if several modes of operation are supported these can be passed in the argument on a custom form, else just use argument 1 to indicate low power mode, argument 0 turns low power mode off.

PIN_CONFIG_OUTPUT_ENABLE This will enable the pin's output mode without driving a value there. For most platforms this reduces to enable the output buffers and then let the pin controller current configuration (eg. the currently selected mux function) drive values on the line. Use argument 1 to enable output mode, argument 0 to disable it.

PIN_CONFIG_OUTPUT This will configure the pin as an output and drive a value on the line. Use argument 1 to indicate high level, argument 0 to indicate low level. (Please see Documentation/driver-api/pinctl.rst, section "GPIO mode pitfalls" for a discussion around this parameter.)

PIN_CONFIG_POWER_SOURCE If the pin can select between different power supplies, the argument to this parameter (on a custom format) tells the driver which alternative power source to use.

PIN_CONFIG_SLEEP_HARDWARE_STATE Indicate this is sleep related state.

PIN_CONFIG_SLEW_RATE If the pin can select slew rate, the argument to this parameter (on a custom format) tells the driver which alternative slew rate to use.

PIN_CONFIG_SKEW_DELAY If the pin has programmable skew rate (on inputs) or latch delay (on outputs) this parameter (in a custom format) specifies the clock skew or latch delay. It typically controls how many double inverters are put in front of the line.

PIN_CONFIG_END This is the last enumerator for pin configurations, if you need to pass in custom configurations to the pin controller, use PIN_CONFIG_END+1 as the base offset.

PIN_CONFIG_MAX This is the maximum configuration value that can be presented using the packed format.

int **pinctrl_generic_set_state**(struct udevice * *pctldev*, struct udevice * *config*)
Generic set_state operation

Parameters

struct udevice * pctldev Pinctrl device to use

struct udevice * config Config device (pseudo device), pointing a config node in DTS

Description

Parse the DT node of **config** and its children and handle generic properties such as "pins", "groups", "functions", and pin configuration parameters.

Return

0 on success, or negative error code on failure

int **pinctrl_select_state**(struct udevice * *dev*, const char * *statename*)
Set a device to a given state

Parameters

struct udevice * dev Peripheral device

const char * statename State name, like "default"

Return

0 on success, or negative error code on failure

int **pinctrl_request**(struct udevice * *dev*, int *func*, int *flags*)
Request a particular pinctrl function

Parameters

struct udevice * dev Pinctrl device to use

int func Function number (driver-specific)

int flags Flags (driver-specific)

Return

0 if OK, or negative error code on failure

int **pinctrl_request_noflags**(struct udevice * *dev*, int *func*)
Request a particular pinctrl function

Parameters

struct udevice * dev Pinctrl device to use

int func Function number (driver-specific)

Description

This is similar to *pinctrl_request()* but uses 0 for **flags**.

Return

0 if OK, or negative error code on failure

int **pinctrl_get_periph_id**(struct udevice * *dev*, struct udevice * *periph*)
Get the peripheral ID for a device

Parameters

struct udevice * dev Pinctrl device to use for decoding

struct udevice * periph Device to check

Description

This generally looks at the peripheral's device tree node to work out the peripheral ID. The return value is normally interpreted as enum periph_id. so long as this is defined by the platform (which it should be).

Return

Peripheral ID of **periph**, or -ENOENT on error

int **pinctrl_get_gpio_mux**(struct udevice * *dev*, int *banknum*, int *index*)
get the mux value for a particular GPIO

Parameters

struct udevice * dev Pinctrl device to use

int banknum GPIO bank number

int index GPIO index within the bank

Description

This allows the raw mux value for a GPIO to be obtained. It is useful for displaying the function being used by that GPIO, such as with the 'gpio' command. This function is internal to the GPIO subsystem and should not be used by generic code. Typically it is used by a GPIO driver with knowledge of the SoC pinctrl setup.

Return

Mux value (SoC-specific, e.g. 0 for input, 1 for output)

int **pinctrl_get_pin_muxing**(struct udevice * *dev*, int *selector*, char * *buf*, int *size*)
Returns the muxing description

Parameters

struct udevice * dev Pinctrl device to use

int selector Pin index within pin-controller

char * buf Pin's muxing description

int size Pin's muxing description length

Description

This allows to display the muxing description of the given pin for debug purpose

Return

0 if OK, or negative error code on failure

int **pinctrl_get_pins_count**(struct udevice * *dev*)
Display pin-controller pins number

Parameters

struct udevice * dev Pinctrl device to use

Description

This allows to know the number of pins owned by a given pin-controller

Return

Number of pins if OK, or negative error code on failure

int **pinctrl_get_pin_name**(struct udevice * *dev*, int *selector*, char * *buf*, int *size*)
Returns the pin's name

Parameters

struct udevice * dev Pinctrl device to use

int selector Pin index within pin-controller

char * buf Buffer to fill with the name of the pin

int size Size of **buf**

Description

This allows to display the pin's name for debug purpose

Return

0 if OK, or negative error code on failure

int **pinctrl_gpio_request**(struct udevice * *dev*, unsigned *offset*)
Request a single pin to be used as GPIO

Parameters

struct udevice * dev GPIO peripheral device

unsigned offset GPIO pin offset from the GPIO controller

Return

0 on success, or negative error code on failure

int **pinctrl_gpio_free**(struct udevice * *dev*, unsigned *offset*)
Free a single pin used as GPIO

Parameters

struct udevice * dev GPIO peripheral device

unsigned offset GPIO pin offset from the GPIO controller

Return

0 on success, or negative error code on failure

5.1.6 Random number generation

Hardware random number generation

int **dm_rng_read**(struct udevice * *dev*, void * *buffer*, size_t *size*)
read a random number seed from the rng device

Parameters

struct udevice * dev random number generator device

void * buffer input buffer to put the read random seed into

size_t size number of random bytes to read

Description

The function blocks until the requested number of bytes is read.

Return

0 if OK, -ve on error

struct **dm_rng_ops**
operations for the hwrng uclass

Definition

```
struct dm_rng_ops {
  int (*read)(struct udevice *dev, void *data, size_t max);
};
```

Members

read read a random bytes

The function blocks until the requested number of bytes is read.

read.dev: random number generator device **read.data**: input buffer to read the random seed into **read.max**: number of random bytes to read **read.Return**: 0 if OK, -ve on error

Description

This structures contains the function implemented by a hardware random number generation device.

Pseudo random number generation

void **srand**(unsigned int *seed*)
Set the random-number seed value

Parameters

unsigned int seed New seed

Description

This can be used to restart the pseudo-random-number sequence from a known point. This affects future calls to *rand()* to start from that point

unsigned int **rand**(void)
Get a 32-bit pseudo-random number

Parameters

void no arguments

Return

next random number in the sequence

unsigned int **rand_r**(unsigned int * *seedp*)
Get a 32-bit pseudo-random number

Parameters

unsigned int * seedp seed value to use, updated on exit

Description

This version of the function allows multiple sequences to be used at the same time, since it requires the caller to store the seed value.

Return

next random number in the sequence

5.1.7 Sandbox

The following API routines are used to implement the U-Boot sandbox.

ssize_t **os_read**(int *fd*, void * *buf*, size_t *count*)

Parameters

int fd File descriptor as returned by *os_open()*

void * buf Buffer to place data

size_t count Number of bytes to read

Return

number of bytes read, or -1 on error

ssize_t **os_write**(int *fd*, const void * *buf*, size_t *count*)

Parameters

int fd File descriptor as returned by *os_open()*

const void * buf Buffer containing data to write

size_t count Number of bytes to write

Return

number of bytes written, or -1 on error

off_t **os_lseek**(int *fd*, off_t *offset*, int *whence*)

Parameters

int fd File descriptor as returned by *os_open()*

off_t offset File offset (based on whence)

int whence Position offset is relative to (see below)

Return

new file offset

int **os_open**(const char * *pathname*, int *flags*)

Parameters

const char * pathname Pathname of file to open

int flags Flags, like OS_O_RDONLY, OS_O_RDWR

Return

file descriptor, or -1 on error

int **os_close**(int *fd*)
access to the OS `close()` system call

Parameters

int fd File descriptor to close

Return

0 on success, -1 on error

int **os_unlink**(const char * *pathname*)
access to the OS `unlink()` system call

Parameters

const char * pathname Path of file to delete

Return

0 for success, other for error

void **os_exit**(int *exit_code*)
access to the OS `exit()` system call

Parameters

int exit_code exit code for U-Boot

Description

This exits with the supplied return code, which should be 0 to indicate success.

void **os_tty_raw**(int *fd*, bool *allow_sigs*)
put tty into raw mode to mimic serial console better

Parameters

int fd File descriptor of stdin (normally 0)

bool allow_sigs Allow Ctrl-C, Ctrl-Z to generate signals rather than be handled by U-Boot

void **os_fd_restore**(void)
restore the tty to its original mode

Parameters

void no arguments

Description

Call this to restore the original terminal mode, after it has been changed by *os_tty_raw()*. This is an internal function.

void * **os_malloc**(size_t *length*)
aquires some memory from the underlying os.

Parameters

size_t length Number of bytes to be allocated

Return

Pointer to length bytes or NULL on error

void **os_free**(void * *ptr*)
free memory previous allocated with *os_malloc()*

Parameters

void * ptr Pointer to memory block to free

Description

This returns the memory to the OS.

void **os_usleep**(unsigned long *usec*)
access to the usleep function of the os

Parameters

unsigned long usec time to sleep in micro seconds

uint64_t **os_get_nsec**(void)

Parameters

void no arguments

Return

a monotonic increasing time scaled in nano seconds

int **os_parse_args**(struct sandbox_state * *state*, int *argc*, char * *argv*)

Parameters

struct sandbox_state * state sandbox state to update

int argc argument count

char * argv argument vector

Return

- 0 if ok, and program should continue
- 1 if ok, but program should stop
- -1 on error: program should terminate

struct **os_dirent_node**
directory node

Definition

```
struct os_dirent_node {
  struct os_dirent_node *next;
  ulong size;
  enum os_dirent_t type;
  char name[0];
};
```

Members

next pointer to next node, or NULL

size size of file in bytes

type type of entry

name name of entry

Description

A directory entry node, containing information about a single dirent

int **os_dirent_ls**(const char * *dirname*, struct *os_dirent_node* ** *headp*)
get a directory listing

Parameters

const char * dirname directory to examine

struct os_dirent_node ** headp on return pointer to head of linked list, or NULL if none

Description

This allocates and returns a linked list containing the directory listing.

Return

0 if ok, -ve on error

void **os_dirent_free**(struct *os_dirent_node* * *node*)
free directory list

Parameters

struct os_dirent_node * node pointer to head of linked list

Description

This frees a linked list containing a directory listing.

const char * **os_dirent_get_typename**(enum os_dirent_t *type*)
get the name of a directory entry type

Parameters

enum os_dirent_t type type to check

Return

string containing the name of that type, or "???" if none/invalid

int **os_get_filesize**(const char * *fname*, loff_t * *size*)
get the size of a file

Parameters

const char * fname filename to check

loff_t * size size of file is returned if no error

Return

0 on success or -1 if an error ocurred

void **os_putc**(int *ch*)
write a character to the controlling OS terminal

Parameters

int ch haracter to write

Description

This bypasses the U-Boot console support and writes directly to the OS stdout file descriptor.

void **os_puts**(const char * *str*)
write a string to the controlling OS terminal

Parameters

const char * str string to write (note that n is not appended)

Description

This bypasses the U-Boot console support and writes directly to the OS stdout file descriptor.

int **os_write_ram_buf**(const char * *fname*)
write the sandbox RAM buffer to a existing file

Parameters

const char * fname filename to write memory to (simple binary format)

Return

0 if OK, -ve on error

int **os_read_ram_buf**(const char * *fname*)
read the sandbox RAM buffer from an existing file

Parameters

const char * fname filename containing memory (simple binary format)

Return

0 if OK, -ve on error

int **os_jump_to_image**(const void * *dest*, int *size*)
jump to a new executable image

Parameters

const void * dest buffer containing executable image

int size size of buffer

Description

This uses exec() to run a new executable image, after putting it in a temporary file. The same arguments and environment are passed to this new image, with the addition of:

-j <filename> Specifies the filename the image was written to. The calling image may want to delete this at some point.

-m <filename> Specifies the file containing the sandbox memory (ram_buf) from this image, so that the new image can have access to this. It also means that the original memory filename passed to U-Boot will be left intact.

Return

0 if OK, -ve on error

int **os_find_u_boot**(char * *fname*, int *maxlen*)
determine the path to U-Boot proper

Parameters

char * fname place to put full path to U-Boot

int maxlen maximum size of **fname**

Description

This function is intended to be called from within sandbox SPL. It uses a few heuristics to find U-Boot proper. Normally it is either in the same directory, or the directory above (since u-boot-spl is normally in an spl/ subdirectory when built).

Return

0 if OK, -NOSPC if the filename is too large, -ENOENT if not found

int **os_spl_to_uboot**(const char * *fname*)
Run U-Boot proper

Parameters

const char * fname full pathname to U-Boot executable

Description

When called from SPL, this runs U-Boot proper. The filename is obtained by calling *os_find_u_boot()*.

Return

0 if OK, -ve on error

void **os_localtime**(struct rtc_time * *rt*)
read the current system time

Parameters

struct rtc_time * rt place to put system time

Description

This reads the current Local Time and places it into the provided structure.

void **os_abort**(void)
raise SIGABRT to exit sandbox (e.g. to debugger)

Parameters

void no arguments

int **os_mprotect_allow**(void * *start*, size_t *len*)
Remove write-protection on a region of memory

Parameters

void * start Region start

size_t len Region length in bytes

Description

The start and length will be page-aligned before use.

Return

0 if OK, -1 on error from `mprotect()`

int **os_write_file**(const char * *name*, const void * *buf*, int *size*)
write a file to the host filesystem

Parameters

const char * name File path to write to

const void * buf Data to write

int size Size of data to write

Description

This can be useful when debugging for writing data out of sandbox for inspection by external tools.

Return

0 if OK, -ve on error

int **os_read_file**(const char * *name*, void ** *bufp*, int * *sizep*)
Read a file from the host filesystem

Parameters

const char * name File path to read from

void ** bufp Returns buffer containing data read

int * sizep Returns size of data

Description

This can be useful when reading test data into sandbox for use by test routines. The data is allocated using *os_malloc()* and should be freed by the caller.

Return

0 if OK, -ve on error

void **os_relaunch**(char * *argv*)
restart the sandbox

Parameters

char * argv NULL terminated list of command line parameters

Description

This functions is used to implement the cold reboot of the sand box. **argv**[0] specifies the binary that is started while the calling process stops immediately. If the new binary cannot be started, the process is terminated and 1 is set as shell return code.

The PID of the process stays the same. All file descriptors that have not been opened with O_CLOEXEC stay open including stdin, stdout, stderr.

5.1.8 Serial system

void **serial_null**(void)
Void registration routine of a serial driver

Parameters

void no arguments

Description

This routine implements a void registration routine of a serial driver. The registration routine of a particular driver is aliased to this empty function in case the driver is not compiled into U-Boot.

int **on_baudrate**(const char * *name*, const char * *value*, enum env_op *op*, int *flags*)
Update the actual baudrate when the env var changes

Parameters

const char * name changed environment variable

const char * value new value of the environment variable

enum env_op op operation (create, overwrite, or delete)

int flags attributes of environment variable change, see flags H_* in include/search.h

Description

This will check for a valid baudrate and only apply it if valid.

Return

0 on success, 1 on error

serial_initfunc(*name*)
Forward declare of driver registration routine

Parameters

name Name of the real driver registration routine.

Description

This macro expands onto forward declaration of a driver registration routine, which is then used below in *serial_initialize()* function. The declaration is made weak and aliases to *serial_null()* so in case the driver is not compiled in, the function is still declared and can be used, but aliases to *serial_null()* and thus is optimized away.

void **serial_register**(struct serial_device * *dev*)
Register serial driver with serial driver core

Parameters

struct serial_device * dev Pointer to the serial driver structure

Description

This function registers the serial driver supplied via **dev** with serial driver core, thus making U-Boot aware of it and making it available for U-Boot to use. On platforms that still require manual relocation of constant variables, relocation of the supplied structure is performed.

int **serial_initialize**(void)
Register all compiled-in serial port drivers

Parameters

void no arguments

Description

This function registers all serial port drivers that are compiled into the U-Boot binary with the serial core, thus making them available to U-Boot to use. Lastly, this function assigns a default serial port to the serial core. That serial port is then used as a default output.

void **serial_stdio_init**(void)
Register serial ports with STDIO core

Parameters

void no arguments

Description

This function generates a proxy driver for each serial port driver. These proxy drivers then register with the STDIO core, making the serial drivers available as STDIO devices.

int **serial_assign**(const char * *name*)
Select the serial output device by name

Parameters

const char * name Name of the serial driver to be used as default output

Description

This function configures the serial output multiplexing by selecting which serial device will be used as default. In case the STDIO "serial" device is selected as stdin/stdout/stderr, the serial device previously configured by this function will be used for the particular operation.

Returns 0 on success, negative on error.

void **`serial_reinit_all`**(void)
    Reinitialize all compiled-in serial ports

Parameters

`void` no arguments

Description

This function reinitializes all serial ports that are compiled into U-Boot by calling their `serial_start()` functions.

struct serial_device * **`get_current`**(void)
    Return pointer to currently selected serial port

Parameters

`void` no arguments

Description

This function returns a pointer to currently selected serial port. The currently selected serial port is altered by *`serial_assign()`* function.

In case this function is called before relocation or before any serial port is configured, this function calls `default_serial_console()` to determine the serial port. Otherwise, the configured serial port is returned.

Returns pointer to the currently selected serial port on success, NULL on error.

int **`serial_init`**(void)
    Initialize currently selected serial port

Parameters

`void` no arguments

Description

This function initializes the currently selected serial port. This usually involves setting up the registers of that particular port, enabling clock and such. This function uses the *`get_current()`* call to determine which port is selected.

Returns 0 on success, negative on error.

void **`serial_setbrg`**(void)
    Configure baud-rate of currently selected serial port

Parameters

`void` no arguments

Description

This function configures the baud-rate of the currently selected serial port. The baud-rate is retrieved from global data within the serial port driver. This function uses the *`get_current()`* call to determine which port is selected.

Returns 0 on success, negative on error.

int **`serial_getc`**(void)
    Read character from currently selected serial port

Parameters

`void` no arguments

Description

This function retrieves a character from currently selected serial port. In case there is no character waiting on the serial port, this function will block and wait for the character to appear. This function uses the *get_current()* call to determine which port is selected.

Returns the character on success, negative on error.

int **serial_tstc**(void)
    Test if data is available on currently selected serial port

Parameters

void no arguments

Description

This function tests if one or more characters are available on currently selected serial port. This function never blocks. This function uses the *get_current()* call to determine which port is selected.

Returns positive if character is available, zero otherwise.

void **serial_putc**(const char *c*)
    Output character via currently selected serial port

Parameters

const char c Single character to be output from the serial port.

Description

This function outputs a character via currently selected serial port. This character is passed to the serial port driver responsible for controlling the hardware. The hardware may still be in process of transmitting another character, therefore this function may block for a short amount of time. This function uses the *get_current()* call to determine which port is selected.

void **serial_puts**(const char * *s*)
    Output string via currently selected serial port

Parameters

const char * s Zero-terminated string to be output from the serial port.

Description

This function outputs a zero-terminated string via currently selected serial port. This function behaves as an accelerator in case the hardware can queue multiple characters for transfer. The whole string that is to be output is available to the function implementing the hardware manipulation. Transmitting the whole string may take some time, thus this function may block for some amount of time. This function uses the *get_current()* call to determine which port is selected.

void **default_serial_puts**(const char * *s*)
    Output string by calling *serial_putc()* in loop

Parameters

const char * s Zero-terminated string to be output from the serial port.

Description

This function outputs a zero-terminated string by calling *serial_putc()* in a loop. Most drivers do not support queueing more than one byte for transfer, thus this function precisely implements their *serial_puts()*.

To optimize the number of *get_current()* calls, this function only calls *get_current()* once and then directly accesses the `putc()` call of the `struct serial_device` .

int **uart_post_test**(int *flags*)
    Test the currently selected serial port using POST

Parameters

int flags POST framework flags

Description

Do a loopback test of the currently selected serial port. This function is only useful in the context of the POST testing framwork. The serial port is first configured into loopback mode and then characters are sent through it.

Returns 0 on success, value otherwise.

5.1.9 Timer Subsystem

int **dm_timer_init**(void)
initialize a timer for time keeping. On success initializes gd->timer so that lib/timer can use it for future referrence.

Parameters

void no arguments

Return

0 on success or error number

int **timer_timebase_fallback**(struct udevice * *dev*)
Helper for timers using timebase fallback

Parameters

struct udevice * dev A timer partially-probed timer device

Description

This is a helper function designed for timers which need to fall back on the cpu's timebase. This function is designed to be called during the driver's `probe()`. If there is a clocks or clock-frequency property in the timer's binding, then it will be used. Otherwise, the timebase of the current cpu will be used. This is initialized by the cpu driver, and usually gotten from `/cpus/timebase-frequency` or `/cpus/cpu**X**/timebase-frequency`.

Return

0 if OK, or negative error code on failure

u64 **timer_conv_64**(u32 *count*)
convert 32-bit counter value to 64-bit

Parameters

u32 count 32-bit counter value

Return

64-bit counter value

int **timer_get_count**(struct udevice * *dev*, u64 * *count*)
Get the current timer count

Parameters

struct udevice * dev The timer device

u64 * count pointer that returns the current timer count

Return

0 if OK, -ve on error

unsigned long **timer_get_rate**(struct udevice * *dev*)
Get the timer input clock frequency

Parameters

struct udevice * dev The timer device

Return

the timer input clock frequency

struct **timer_ops**
Driver model timer operations

Definition

```
struct timer_ops {
  u64 (*get_count)(struct udevice *dev);
};
```

Members

get_count Get the current timer count

dev: The timer device

This function may be called at any time after the driver is probed. All necessary initialization must be completed by the time probe() returns. The count returned by this functions should be monotonic. This function must succeed.

Return: The current 64-bit timer count

Description

The uclass interface is implemented by all timer devices which use driver model.

struct **timer_dev_priv**
information about a device used by the uclass

Definition

```
struct timer_dev_priv {
  unsigned long clock_rate;
};
```

Members

clock_rate the timer input clock frequency

u64 **timer_early_get_count**(void)
Implement *timer_get_count()* before driver model

Parameters

void no arguments

Description

If CONFIG_TIMER_EARLY is enabled, this function wil be called to return the current timer value before the proper driver model timer is ready. It should be implemented by one of the timer values. This is mostly useful for tracing.

unsigned long **timer_early_get_rate**(void)
Get the timer rate before driver model

Parameters

void no arguments

Description

If CONFIG_TIMER_EARLY is enabled, this function wil be called to return the current timer rate in Hz before the proper driver model timer is ready. It should be implemented by one of the timer values. This is mostly useful for tracing. This corresponds to the clock_rate value in struct timer_dev_priv.

5.1.10 Unicode support

int **console_read_unicode**(s32 * *code*)
read Unicode code point from console

Parameters

s32 * code pointer to store Unicode code point

Return

0 = success

s32 **utf8_get**(const char ** *src*)
get next UTF-8 code point from buffer

Parameters

const char ** src pointer to current byte, updated to point to next byte

Return

code point, or 0 for end of string, or -1 if no legal code point is found. In case of an error src points to the incorrect byte.

int **utf8_put**(s32 *code*, char ** *dst*)
write UTF-8 code point to buffer

Parameters

s32 code code point

char ** dst pointer to destination buffer, updated to next position

Return

-1 if the input parameters are invalid

size_t **utf8_utf16_strnlen**(const char * *src*, size_t *count*)
length of a truncated utf-8 string after conversion to utf-16

Parameters

const char * src utf-8 string

size_t count maximum number of code points to convert

Return

length in u16 after conversion to utf-16 without the trailing 0. If an invalid UTF-8 sequence is hit one u16 will be reserved for a replacement character.

utf8_utf16_strlen(*a*)
length of a utf-8 string after conversion to utf-16

Parameters

a utf-8 string

Return

length in u16 after conversion to utf-16 without the trailing 0. If an invalid UTF-8 sequence is hit one u16 will be reserved for a replacement character.

int **utf8_utf16_strncpy**(u16 ** *dst*, const char * *src*, size_t *count*)
copy utf-8 string to utf-16 string

Parameters

u16 ** dst destination buffer

const char * src source buffer

size_t count maximum number of code points to copy

Return

-1 if the input parameters are invalid

utf8_utf16_strcpy(*d*, *s*)
copy utf-8 string to utf-16 string

Parameters

d destination buffer

s source buffer

Return

-1 if the input parameters are invalid

s32 **utf16_get**(const u16 ** *src*)
get next UTF-16 code point from buffer

Parameters

const u16 ** src pointer to current word, updated to point to next word

Return

code point, or 0 for end of string, or -1 if no legal code point is found. In case of an error src points to the incorrect word.

int **utf16_put**(s32 *code*, u16 ** *dst*)
write UTF-16 code point to buffer

Parameters

s32 code code point

u16 ** dst pointer to destination buffer, updated to next position

Return

-1 if the input parameters are invalid

size_t **utf16_strnlen**(const u16 * *src*, size_t *count*)
length of a truncated utf-16 string

Parameters

const u16 * src utf-16 string

size_t count maximum number of code points to convert

Return

length in code points. If an invalid UTF-16 sequence is hit one position will be reserved for a replacement character.

size_t **utf16_utf8_strnlen**(const u16 * *src*, size_t *count*)
length of a truncated utf-16 string after conversion to utf-8

Parameters

const u16 * src utf-16 string

size_t count maximum number of code points to convert

Return

length in bytes after conversion to utf-8 without the trailing 0. If an invalid UTF-16 sequence is hit one byte will be reserved for a replacement character.

utf16_utf8_strlen(*a*)
length of a utf-16 string after conversion to utf-8

Parameters

a utf-16 string

Return

length in bytes after conversion to utf-8 without the trailing 0. If an invalid UTF-16 sequence is hit one byte will be reserved for a replacement character.

int **utf16_utf8_strncpy**(char ** *dst*, const u16 * *src*, size_t *count*)
copy utf-16 string to utf-8 string

Parameters

char ** dst destination buffer

const u16 * src source buffer

size_t count maximum number of code points to copy

Return

-1 if the input parameters are invalid

utf16_utf8_strcpy(*d*, *s*)
copy utf-16 string to utf-8 string

Parameters

d destination buffer

s source buffer

Return

-1 if the input parameters are invalid

s32 **utf_to_lower**(const s32 *code*)
convert a Unicode letter to lower case

Parameters

const s32 code letter to convert

Return

lower case letter or unchanged letter

s32 **utf_to_upper**(const s32 *code*)
convert a Unicode letter to upper case

Parameters

const s32 code letter to convert

Return

upper case letter or unchanged letter

int **u16_strncmp**(const u16 * *s1*, const u16 * *s2*, size_t *n*)
compare two u16 string

Parameters

const u16 * s1 first string to compare

const u16 * s2 second string to compare

size_t n maximum number of u16 to compare

Return

0 if the first n u16 are the same in s1 and s2 < 0 if the first different u16 in s1 is less than the corresponding u16 in s2 > 0 if the first different u16 in s1 is greater than the corresponding u16 in s2

u16_strcmp(*s1*, *s2*)
compare two u16 string

Parameters

s1 first string to compare

s2 second string to compare

Return

0 if the first n u16 are the same in s1 and s2 < 0 if the first different u16 in s1 is less than the corresponding u16 in s2 > 0 if the first different u16 in s1 is greater than the corresponding u16 in s2

size_t **u16_strlen**(const void * *in*)
count non-zero words

Parameters

const void * in null terminated u16 string

Description

This function matches `wsclen()` if the -fshort-wchar compiler flag is set. In the EFI context we explicitly need a function handling u16 strings.

Return

number of non-zero words. This is not the number of utf-16 letters!

size_t **u16_strsize**(const void * *in*)
count size of u16 string in bytes including the null character

Parameters

const void * in null terminated u16 string

Description

Counts the number of bytes occupied by a u16 string

Return

bytes in a u16 string

size_t **u16_strnlen**(const u16 * *in*, size_t *count*)
count non-zero words

Parameters

const u16 * in null terminated u16 string

size_t count maximum number of words to count

Description

This function matches `wscnlen_s()` if the -fshort-wchar compiler flag is set. In the EFI context we explicitly need a function handling u16 strings.

Return

number of non-zero words. This is not the number of utf-16 letters!

u16 * **u16_strcpy**(u16 * *dest*, const u16 * *src*)
copy u16 string

Parameters

u16 * dest destination buffer

const u16 * src source buffer (null terminated)

Description

Copy u16 string pointed to by src, including terminating null word, to the buffer pointed to by dest.

Return

'dest' address

u16 * **u16_strdup**(const void * *src*)
  duplicate u16 string

Parameters

const void * src source buffer (null terminated)

Description

Copy u16 string pointed to by src, including terminating null word, to a newly allocated buffer.

Return

allocated new buffer on success, NULL on failure

uint8_t * **utf16_to_utf8**(uint8_t * *dest*, const uint16_t * *src*, size_t *size*)
  Convert an utf16 string to utf8

Parameters

uint8_t * dest the destination buffer to write the utf8 characters

const uint16_t * src the source utf16 string

size_t size the number of utf16 characters to convert

Description

Converts 'size' characters of the utf16 string 'src' to utf8 written to the 'dest' buffer.

NOTE that a single utf16 character can generate up to 3 utf8 characters. See MAX_UTF8_PER_UTF16.

Return

the pointer to the first unwritten byte in 'dest'

CHAPTER

SIX

ARCHITECTURE-SPECIFIC DOC

These books provide programming details about architecture-specific implementation.

6.1 Architecture-specific doc

6.1.1 ARC

Synopsys' DesignWare(r) ARC(r) Processors are a family of 32-bit CPUs that SoC designers can optimize for a wide range of uses, from deeply embedded to high-performance host applications.

More information on ARC cores avaialble here: http://www.synopsys.com/IP/ProcessorIP/ARCProcessors/Pages/default.aspx

Designers can differentiate their products by using patented configuration technology to tailor each ARC processor instance to meet specific performance, power and area requirements.

The DesignWare ARC processors are also extendable, allowing designers to add their own custom instructions that dramatically increase performance.

Synopsys' ARC processors have been used by over 170 customers worldwide who collectively ship more than 1 billion ARC-based chips annually.

All DesignWare ARC processors utilize a 16-/32-bit ISA that provides excellent performance and code density for embedded and host SoC applications.

The RISC microprocessors are synthesizable and can be implemented in any foundry or process, and are supported by a complete suite of development tools.

The ARC GNU toolchain with support for all ARC Processors can be downloaded from here (available prebuilt toolchains as well):

https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/releases

6.1.2 ARM64

Summary

The initial arm64 U-Boot port was developed before hardware was available, so the first supported platforms were the Foundation and Fast Model for ARMv8. These days U-Boot runs on a variety of 64-bit capable ARM hardware, from embedded development boards to servers.

Notes

1. U-Boot can run at any exception level it is entered in, it is recommened to enter it in EL3 if U-Boot takes some responsibilities of a classical firmware (like initial hardware setup, CPU errata workarounds or SMP bringup). U-Boot can be entered in EL2 when its main purpose is that of a boot loader. It can drop to lower exception levels before entering the OS.

2. U-Boot for arm64 is compiled with AArch64-gcc. AArch64-gcc use rela relocation format, a tool(tools/relocate-rela) by Scott Wood is used to encode the initial addend of rela to u-boot.bin. After running, the U-Boot will be relocated to destination again.
3. Earlier Linux kernel versions required the FDT to be placed at a 2 MB boundary and within the same 512 MB section as the kernel image, resulting in fdt_high to be defined specially. Since kernel version 4.2 Linux is more relaxed about the DT location, so it can be placed anywhere in memory. Please reference linux/Documentation/arm64/booting.txt for detail.
4. Spin-table is used to wake up secondary processors. One location (or per processor location) is defined to hold the kernel entry point for secondary processors. It must be ensured that the location is accessible and zero immediately after secondary processor enter slave_cpu branch execution in start.S. The location address is encoded in cpu node of DTS. Linux kernel store the entry point of secondary processors to it and send event to wakeup secondary processors. Please reference linux/Documentation/arm64/booting.txt for detail.
5. Generic board is supported.
6. CONFIG_ARM64 instead of CONFIG_ARMV8 is used to distinguish aarch64 and aarch32 specific codes.

Contributors

- Tom Rini <trini@ti.com>
- Scott Wood <scottwood@freescale.com>
- York Sun <yorksun@freescale.com>
- Simon Glass <sjg@chromium.org>
- Sharma Bhupesh <bhupesh.sharma@freescale.com>
- Rob Herring <robherring2@gmail.com>
- Sergey Temerkhanov <s.temerkhanov@gmail.com>

6.1.3 M68K / ColdFire

History

- November 02, 2017 Angelo Dureghello <angelo@sysam.it>
- August 08, 2005 Jens Scharsig <esw@bus-elektronik.de> MCF5282 implementation without preloader
- January 12, 2004 <josef.baumgartner@telex.de>

This file contains status information for the port of U-Boot to the Motorola ColdFire series of CPUs.

Overview

The ColdFire instruction set is "assembly source" compatible but an evolution of the original 68000 instruction set. Some not much used instructions has been removed. The instructions are only 16, 32, or 48 bits long, a simplification compared to the 68000 series.

Bernhard Kuhn ported U-Boot 0.4.0 to the Motorola ColdFire architecture. The patches of Bernhard support the MCF5272 and MCF5282. A great disadvantage of these patches was that they needed a pre-bootloader to start U-Boot. Because of this, a new port was created which no longer needs a first stage booter.

Thanks mainly to Freescale but also to several other contributors, U-Boot now supports nearly the entire range of ColdFire processors and their related development boards.

Supported CPU families

Please “make menuconfig” and select “m68k” or check arch/m68k/cpu to see the currently supported processor and families.

Supported boards

U-Boot supports actually more than 40 ColdFire based boards. Board configuration can be done trough include/configs/<boardname>.h but the current recommended method is to use the new and more friendly approach as the “make menuconfig” way, very similar to the Linux way.

To know details as memory map, build targets, default setup, etc, of a specific board please check:

- include/configs/<boardname>.h

and/or

- configs/<boardname>_defconfig

It is possible to build all ColdFire boards in a single command-line command, from u-boot root directory, as:

```
./tools/buildman/buildman m68k
```

Build U-Boot for a specific board

A bash script similar to the one below may be used:

```
#!/bin/bash

export CROSS_COMPILE=/opt/toolchains/m68k/gcc-4.9.0-nolibc/bin/m68k-linux-

board=M5475DFE

make distclean
make ${board}_defconfig
make KBUILD_VERBOSE=1
```

Adopted toolchains

Please check: https://www.denx.de/wiki/U-Boot/ColdFireNotes

ColdFire specific configuration options/settings

Configuration to use a pre-loader

If U-Boot should be loaded to RAM and started by a pre-loader CONFIG_MONITOR_IS_IN_RAM must be defined. If it is defined the initial vector table and basic processor initialization will not be compiled in. The start address of U-Boot must be adjusted in the boards config header file (CONFIG_SYS_MONITOR_BASE) and Makefile (CONFIG_SYS_TEXT_BASE) to the load address.

ColdFire CPU specific options/settings

To specify a CPU model, some defines shoudl be used, i.e.:

CONFIG_MCF52x2: defined for all MCF52x2 CPUs

CONFIG_M5272: defined for all Motorola MCF5272 CPUs

Other options, generally set inside include/configs/<boardname>.h, they may apply to one or more cpu for the ColdFire family:

CONFIG_SYS_MBAR: defines the base address of the MCF5272 configuration registers

CONFIG_SYS_ENET_BD_BASE: defines the base address of the FEC buffer descriptors

CONFIG_SYS_SCR: defines the contents of the System Configuration Register

CONFIG_SYS_SPR: defines the contents of the System Protection Register

CONFIG_SYS_MFD: defines the PLL Multiplication Factor Divider (see table 9-4 of MCF user manual)

CONFIG_SYS_RFD: defines the PLL Reduce Frequency Devider (see table 9-4 of MCF user manual)

CONFIG_SYS_CSx_BASE: defines the base address of chip select x

CONFIG_SYS_CSx_SIZE: defines the memory size (address range) of chip select x

CONFIG_SYS_CSx_WIDTH: defines the bus with of chip select x

CONFIG_SYS_CSx_MASK: defines the mask for the related chip select x

CONFIG_SYS_CSx_RO: if set to 0 chip select x is read/write else chip select is read only

CONFIG_SYS_CSx_WS: defines the number of wait states of chip select x

CONFIG_SYS_CACHE_ICACR: cache-related registers config

CONFIG_SYS_CACHE_DCACR: cache-related registers config

CONFIG_SYS_CACHE_ACRX: cache-related registers config

CONFIG_SYS_SDRAM_BASE: SDRAM config for SDRAM controller-specific registers

CONFIG_SYS_SDRAM_SIZE: SDRAM config for SDRAM controller-specific registers

CONFIG_SYS_SDRAM_BASEX: SDRAM config for SDRAM controller-specific registers

CONFIG_SYS_SDRAM_CFG1: SDRAM config for SDRAM controller-specific registers

CONFIG_SYS_SDRAM_CFG2: SDRAM config for SDRAM controller-specific registers

CONFIG_SYS_SDRAM_CTRL: SDRAM config for SDRAM controller-specific registers

CONFIG_SYS_SDRAM_MODE: SDRAM config for SDRAM controller-specific registers

CONFIG_SYS_SDRAM_EMOD: SDRAM config for SDRAM controller-specific registers, please see arch/m68k/cpu/<specific_cpu>/start.S files to see how these options are used.

CONFIG_MCFUART: defines enabling of ColdFire UART driver

CONFIG_SYS_UART_PORT: defines the UART port to be used (only a single UART can be actually enabled)

CONFIG_SYS_SBFHDR_SIZE: size of the prepended SBF header, if any

6.1.4 MIPS

Notes for the MIPS architecture port of U-Boot

Toolchains

- ELDK < DULG < DENX
- Embedded Debian – Cross-development toolchains
- Buildroot

Known Issues

- Cache incoherency issue caused by do_bootelf_exec() at cmd_elf.c

  Cache will be disabled before entering the loaded ELF image without writing back and invalidating cache lines. This leads to cache incoherency in most cases, unless the code gets loaded after U-Boot re-initializes the cache. The more common uImage 'bootm' command does not suffer this problem.

  [workaround] To avoid this cache incoherency:

  - insert flush_cache(all) before calling dcache_disable(), or
  - fix dcache_disable() to do both flushing and disabling cache.

- Note that Linux users need to kill dcache_disable() in do_bootelf_exec() or override do_bootelf_exec() not to disable I-/D-caches, because most Linux/MIPS ports don't re-enable caches after entering kernel_entry.

TODOs

- Probe CPU types, I-/D-cache and TLB size etc. automatically
- Secondary cache support missing
- Initialize TLB entries redardless of their use
- R2000/R3000 class parts are not supported
- Limited testing across different MIPS variants
- Due to cache initialization issues, the DRAM on board must be initialized in board specific assembler language before the cache init code is run – that is, initialize the DRAM in lowlevel_init().
- centralize/share more CPU code of MIPS32, MIPS64 and XBurst
- support Qemu Malta

6.1.5 NDS32

NDS32 is a new high-performance 32-bit RISC microprocessor core.

http://www.andestech.com/

AndeStar ISA

AndeStar is a patent-pending 16-bit/32-bit mixed-length instruction set to achieve optimal system performance, code density, and power efficiency.

It contains the following features:

- Intermixable 32-bit and 16-bit instruction sets without the need for mode switch.
- 16-bit instructions as a frequently used subset of 32-bit instructions.
- RISC-style register-based instruction set.
- 32 32-bit General Purpose Registers (GPR).
- Upto 1024 User Special Registers (USR) for existing and extension instructions.
- **Rich load/store instructions for...**
  - Single memory access with base address update.
  - Multiple aligned and unaligned memory accesses for memory copy and stack operations.
  - Data prefetch to improve data cache performance.

  - Non-bus locking synchronization instructions.
- PC relative jump and PC read instructions for efficient position independent code.
- Multiply-add and multiple-sub with 64-bit accumulator.
- Instruction for efficient power management.
- Bi-endian support.
- **Three instruction extension space for application acceleration:**
  - Performance extension.
  - Andes future extensions (for floating-point, multimedia, etc.)
  - Customer extensions.

AndesCore CPU

Andes Technology has 4 families of CPU cores: N12, N10, N9, N8.

For details about N12 CPU family, please check below N1213 features. N1213 is a configurable hard/soft core of NDS32's N12 CPU family.

N1213 Features

CPU Core

- 16-/32-bit mixable instruction format.
- 32 general-purpose 32-bit registers.
- 8-stage pipeline.
- Dynamic branch prediction.
- 32/64/128/256 BTB.
- Return address stack (RAS).
- Vector interrupts for internal/external. interrupt controller with 6 hardware interrupt signals.
- 3 HW-level nested interruptions.
- User and super-user mode support.
- Memory-mapped I/O.
- Address space up to 4GB.

Memory Management Unit

- **TLB**
  - 4/8-entry fully associative iTLB/dTLB.
  - 32/64/128-entry 4-way set-associati.ve main TLB.
  - TLB locking support
- Optional hardware page table walker.
- **Two groups of page size support.**
  - 4KB & 1MB.
  - 8KB & 1MB.

Memory Subsystem

- **I & D cache.**

- Virtually indexed and physically tagged.
- Cache size: 8KB/16KB/32KB/64KB.
- Cache line size: 16B/32B.
- Set associativity: 2-way, 4-way or direct-mapped.
- Cache locking support.

- **I & D local memory (LM).**
  - Size: 4KB to 1MB.
  - Bank numbers: 1 or 2.
  - Optional 1D/2D DMA engine.
  - Internal or external to CPU core.

Bus Interface

- Synchronous/Asynchronous AHB bus: 0, 1 or 2 ports.
- Synchronous High speed memory port. (HSMP): 0, 1 or 2 ports.

Debug

- JTAG debug interface.
- Embedded debug module (EDM).
- Optional embedded program tracer interface.

Miscellaneous

- Programmable data endian control.
- Performance monitoring mechanism.

The NDS32 ports of u-boot, the Linux kernel, the GNU toolchain and other associated software are actively supported by Andes Technology Corporation.

6.1.6 Nios II

Nios II is a 32-bit embedded-processor architecture designed specifically for the Altera family of FPGAs.

Please refer to the link for more information on Nios II: https://www.altera.com/products/processors/overview.html

Please refer to the link for Linux port and toolchains: http://rocketboards.org/foswiki/view/Documentation/NiosIILinuxUserManual

The Nios II port of u-boot is controlled by device tree. Please check out doc/README.fdt-control.

To add a new board/configuration (eg, mysystem) to u-boot, you will need three files.

1. The device tree source which describes the hardware, dts file: arch/nios2/dts/mysystem.dts
2. Default configuration of Kconfig, defconfig file: configs/mysystem_defconfig
3. The legacy board header file: include/configs/mysystem.h

The device tree source must be generated from your qsys/sopc design using the sopc2dts tool. Then modified to fit your configuration.

Please find the sopc2dts download and usage at the wiki: http://www.alterawiki.com/wiki/Sopc2dts

```
$ java -jar sopc2dts.jar --force-altr -i mysystem.sopcinfo -o mysystem.dts
```

You will need to add additional properties to the dts. Please find an example at, arch/nios2/dts/10m50_devboard.dts.

1. Add “stdout-path=…” property with your serial path to the chosen node, like this:

```
chosen {
        stdout-path = &uart_0;
};
```

2. If you use SPI/EPCS or I2C, you will need to add aliases to number the sequence of these devices, like this:

```
aliases {
        spi0 = &epcs_controller;
};
```

Next, you will need a default config file. You may start with 10m50_defconfig, modify the options and save it.

```
$ make 10m50_defconfig
$ make menuconfig
$ make savedefconfig
$ cp defconfig configs/mysystem_defconfig
```

You will need to change the names of board header file and device tree, and select the drivers with menuconfig.

```
Nios II architecture  --->
  (mysystem) Board header file
Device Tree Control  --->
  (mysystem) Default Device Tree for DT control
```

There is a selection of “Provider of DTB for DT control” in the Device Tree Control menu.

- Separate DTB for DT control, will cat the dtb to end of u-boot binary, output u-boot-dtb.bin. This should be used for production. If you use boot copier, like EPCS boot copier, make sure the copier copies all the u-boot-dtb.bin, not just u-boot.bin.
- Embedded DTB for DT control, will include the dtb inside the u-boot binary. This is handy for development, eg, using gdb or nios2-download.

The last thing, legacy board header file describes those config options not covered in Kconfig yet. You may copy it from 10m50_devboard.h:

```
$ cp include/configs/10m50_devboard.h include/configs/mysystem.h
```

Please change the SDRAM base and size to match your board. The base should be cached virtual address, for Nios II with MMU it is 0xCxxx_xxxx to 0xDxxx_xxxx.

```
#define CONFIG_SYS_SDRAM_BASE                0xc8000000
#define CONFIG_SYS_SDRAM_SIZE                0x08000000
```

You will need to change the environment variables location and setting, too. You may change other configs to fit your board.

After all these changes, you may build and test:

```
$ export CROSS_COMPILE=nios2-elf-  (or nios2-linux-gnu-)
$ make mysystem_defconfig
$ make
```

Enjoy!

6.1.7 Sandbox

Native Execution of U-Boot

The 'sandbox' architecture is designed to allow U-Boot to run under Linux on almost any hardware. To achieve this it builds U-Boot (so far as possible) as a normal C application with a main() and normal C libraries.

All of U-Boot's architecture-specific code therefore cannot be built as part of the sandbox U-Boot. The purpose of running U-Boot under Linux is to test all the generic code, not specific to any one architecture. The idea is to create unit tests which we can run to test this upper level code.

CONFIG_SANDBOX is defined when building a native board.

The board name is 'sandbox' but the vendor name is unset, so there is a single board in board/sandbox.

CONFIG_SANDBOX_BIG_ENDIAN should be defined when running on big-endian machines.

There are two versions of the sandbox: One using 32-bit-wide integers, and one using 64-bit-wide integers. The 32-bit version can be build and run on either 32 or 64-bit hosts by either selecting or deselecting CONFIG_SANDBOX_32BIT; by default, the sandbox it built for a 32-bit host. The sandbox using 64-bit-wide integers can only be built on 64-bit hosts.

Note that standalone/API support is not available at present.

Prerequisites

Here are some packages that are worth installing if you are doing sandbox or tools development in U-Boot:

> python3-pytest lzma lzma-alone lz4 python3 python3-virtualenv libssl1.0-dev

Basic Operation

To run sandbox U-Boot use something like:

```
make sandbox_defconfig all
./u-boot
```

Note: If you get errors about 'sdl-config: Command not found' you may need to install libsdl2.0-dev or similar to get SDL support. Alternatively you can build sandbox without SDL (i.e. no display/keyboard support) by removing the CONFIG_SANDBOX_SDL line in include/configs/sandbox.h or using:

```
make sandbox_defconfig all NO_SDL=1
./u-boot
```

U-Boot will start on your computer, showing a sandbox emulation of the serial console:

```
U-Boot 2014.04 (Mar 20 2014 - 19:06:00)

DRAM:  128 MiB
Using default environment

In:    serial
Out:   lcd
Err:   lcd
=>
```

You can issue commands as your would normally. If the command you want is not supported you can add it to include/configs/sandbox.h.

To exit, type 'reset' or press Ctrl-C.

Console / LCD support

Assuming that CONFIG_SANDBOX_SDL is defined when building, you can run the sandbox with LCD and keyboard emulation, using something like:

```
./u-boot -d u-boot.dtb -l
```

This will start U-Boot with a window showing the contents of the LCD. If that window has the focus then you will be able to type commands as you would on the console. You can adjust the display settings in the device tree file - see arch/sandbox/dts/sandbox.dts.

Command-line Options

Various options are available, mostly for test purposes. Use -h to see available options. Some of these are described below:

- -t, –terminal <arg> - The terminal is normally in what is called 'raw-with-sigs' mode. This means that you can use arrow keys for command editing and history, but if you press Ctrl-C, U-Boot will exit instead of handling this as a keypress. Other options are 'raw' (so Ctrl-C is handled within U-Boot) and 'cooked' (where the terminal is in cooked mode and cursor keys will not work, Ctrl-C will exit).
- -l - Show the LCD emulation window.
- -d <device_tree> - A device tree binary file can be provided with -d. If you edit the source (it is stored at arch/sandbox/dts/sandbox.dts) you must rebuild U-Boot to recreate the binary file.
- -D - To use the default device tree, use -D.
- -T - To use the test device tree, use -T.
- -c [<cmd>;]<cmd> - To execute commands directly, use the -c option. You can specify a single command, or multiple commands separated by a semicolon, as is normal in U-Boot. Be careful with quoting as the shell will normally process and swallow quotes. When -c is used, U-Boot exits after the command is complete, but you can force it to go to interactive mode instead with -i.
- -i - Go to interactive mode after executing the commands specified by -c.

Memory Emulation

Memory emulation is supported, with the size set by CONFIG_SYS_SDRAM_SIZE. The -m option can be used to read memory from a file on start-up and write it when shutting down. This allows preserving of memory contents across test runs. You can tell U-Boot to remove the memory file after it is read (on start-up) with the –rm_memory option.

To access U-Boot's emulated memory within the code, use map_sysmem(). This function is used throughout U-Boot to ensure that emulated memory is used rather than the U-Boot application memory. This provides memory starting at 0 and extending to the size of the emulation.

Storing State

With sandbox you can write drivers which emulate the operation of drivers on real devices. Some of these drivers may want to record state which is preserved across U-Boot runs. This is particularly useful for testing. For example, the contents of a SPI flash chip should not disappear just because U-Boot exits.

State is stored in a device tree file in a simple format which is driver- specific. You then use the -s option to specify the state file. Use -r to make U-Boot read the state on start-up (otherwise it starts empty) and -w to write it on exit (otherwise the stored state is left unchanged and any changes U-Boot made will be lost). You can also use -n to tell U-Boot to ignore any problems with missing state. This is useful when first running since the state file will be empty.

The device tree file has one node for each driver - the driver can store whatever properties it likes in there. See 'Writing Sandbox Drivers' below for more details on how to get drivers to read and write their state.

Running and Booting

Since there is no machine architecture, sandbox U-Boot cannot actually boot a kernel, but it does support the bootm command. Filesystems, memory commands, hashing, FIT images, verified boot and many other features are supported.

When 'bootm' runs a kernel, sandbox will exit, as U-Boot does on a real machine. Of course in this case, no kernel is run.

It is also possible to tell U-Boot that it has jumped from a temporary previous U-Boot binary, with the -j option. That binary is automatically removed by the U-Boot that gets the -j option. This allows you to write tests which emulate the action of chain-loading U-Boot, typically used in a situation where a second 'updatable' U-Boot is stored on your board. It is very risky to overwrite or upgrade the only U-Boot on a board, since a power or other failure will brick the board and require return to the manufacturer in the case of a consumer device.

Supported Drivers

U-Boot sandbox supports these emulations:

- Block devices
- Chrome OS EC
- GPIO
- Host filesystem (access files on the host from within U-Boot)
- I2C
- Keyboard (Chrome OS)
- LCD
- Network
- Serial (for console only)
- Sound (incomplete - see sandbox_sdl_sound_init() for details)
- SPI
- SPI flash
- TPM (Trusted Platform Module)

A wide range of commands are implemented. Filesystems which use a block device are supported.

Also sandbox supports driver model (CONFIG_DM) and associated commands.

Sandbox Variants

There are unfortunately quite a few variants at present:

sandbox: should be used for most tests

sandbox64: special build that forces a 64-bit host

sandbox_flattree: builds with dev_read_...() functions defined as inline. We need this build so that we can test those inline functions, and we cannot build with both the inline functions and the non-inline functions since they are named the same.

sandbox_spl: builds sandbox with SPL support, so you can run spl/u-boot-spl and it will start up and then load ./u-boot. It is also possible to run ./u-boot directly.

Of these sandbox_spl can probably be removed since it is a superset of sandbox.

Most of the config options should be identical between these variants.

Linux RAW Networking Bridge

The sandbox_eth_raw driver bridges traffic between the bottom of the network stack and the RAW sockets API in Linux. This allows much of the U-Boot network functionality to be tested in sandbox against real network traffic.

For Ethernet network adapters, the bridge utilizes the RAW AF_PACKET API. This is needed to get access to the lowest level of the network stack in Linux. This means that all of the Ethernet frame is included. This allows the U-Boot network stack to be fully used. In other words, nothing about the Linux network stack is involved in forming the packets that end up on the wire. To receive the responses to packets sent from U-Boot the network interface has to be set to promiscuous mode so that the network card won't filter out packets not destined for its configured (on Linux) MAC address.

The RAW sockets Ethernet API requires elevated privileges in Linux. You can either run as root, or you can add the capability needed like so:

```
sudo /sbin/setcap "CAP_NET_RAW+ep" /path/to/u-boot
```

The default device tree for sandbox includes an entry for eth0 on the sandbox host machine whose alias is "eth1". The following are a few examples of network operations being tested on the eth0 interface.

```
sudo /path/to/u-boot -D

DHCP
....

setenv autoload no
setenv ethrotate no
setenv ethact eth1
dhcp

PING
....

setenv autoload no
setenv ethrotate no
setenv ethact eth1
dhcp
ping $gatewayip

TFTP
....

setenv autoload no
setenv ethrotate no
setenv ethact eth1
dhcp
setenv serverip WWW.XXX.YYY.ZZZ
tftpboot u-boot.bin
```

The bridge also supports (to a lesser extent) the localhost interface, 'lo'.

The 'lo' interface cannot use the RAW AF_PACKET API because the lo interface doesn't support Ethernet-level traffic. It is a higher-level interface that is expected only to be used at the AF_INET level of the API. As such, the most raw we can get on that interface is the RAW AF_INET API on UDP. This allows us to set the IP_HDRINCL option to include everything except the Ethernet header in the packets we send and receive.

Because only UDP is supported, ICMP traffic will not work, so expect that ping commands will time out.

The default device tree for sandbox includes an entry for lo on the sandbox host machine whose alias is “eth5”. The following is an example of a network operation being tested on the lo interface.

```
TFTP
....

setenv ethrotate no
setenv ethact eth5
tftpboot u-boot.bin
```

SPI Emulation

Sandbox supports SPI and SPI flash emulation.

The device can be enabled via a device tree, for example:

```
spi@0 {
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <0 1>;
        compatible = "sandbox,spi";
        cs-gpios = <0>, <&gpio_a 0>;
        spi.bin@0 {
                reg = <0>;
                compatible = "spansion,m25p16", "jedec,spi-nor";
                spi-max-frequency = <40000000>;
                sandbox,filename = "spi.bin";
        };
};
```

The file must be created in advance:

```
$ dd if=/dev/zero of=spi.bin bs=1M count=2
$ u-boot -T
```

Here, you can use “-T” or “-D” option to specify test.dtb or u-boot.dtb, respectively, or “-d <file>” for your own dtb.

With this setup you can issue SPI flash commands as normal:

```
=>sf probe
SF: Detected M25P16 with page size 64 KiB, total 2 MiB
=>sf read 0 0 10000
SF: 65536 bytes @ 0x0 Read: OK
```

Since this is a full SPI emulation (rather than just flash), you can also use low-level SPI commands:

```
=>sspi 0:0 32 9f
FF202015
```

This is issuing a READ_ID command and getting back 20 (ST Micro) part 0x2015 (the M25P16).

Block Device Emulation

U-Boot can use raw disk images for block device emulation. To e.g. list the contents of the root directory on the second partion of the image “disk.raw”, you can use the following commands:

```
=>host bind 0 ./disk.raw
=>ls host 0:2
```

A disk image can be created using the following commands:

```
$> truncate -s 1200M ./disk.raw
$> echo -e "label: gpt\n,64M,U\n,,L" | /usr/sbin/sgdisk  ./disk.raw
$> lodev=`sudo losetup -P -f --show ./disk.raw`
$> sudo mkfs.vfat -n EFI -v ${lodev}p1
$> sudo mkfs.ext4 -L ROOT -v ${lodev}p2
```

or utilize the device described in test/py/make_test_disk.py:

```
#!/usr/bin/python
import make_test_disk
make_test_disk.makeDisk()
```

Writing Sandbox Drivers

Generally you should put your driver in a file containing the word 'sandbox' and put it in the same directory as other drivers of its type. You can then implement the same hooks as the other drivers.

To access U-Boot's emulated memory, use map_sysmem() as mentioned above.

If your driver needs to store configuration or state (such as SPI flash contents or emulated chip registers), you can use the device tree as described above. Define handlers for this with the SANDBOX_STATE_IO macro. See arch/sandbox/include/asm/state.h for documentation. In short you provide a node name, compatible string and functions to read and write the state. Since writing the state can expand the device tree, you may need to use state_setprop() which does this automatically and avoids running out of space. See existing code for examples.

Debugging the init sequence

If you get a failure in the initcall sequence, like this:

```
initcall sequence 0000560775957c80 failed at call 0000000000048134 (err=-96)
```

Then you use can use grep to see which init call failed, e.g.:

```
$ grep 0000000000048134 u-boot.map
stdio_add_devices
```

Of course another option is to run it with a debugger such as gdb:

```
$ gdb u-boot
...
(gdb) br initcall.h:41
Breakpoint 1 at 0x4db9d: initcall.h:41. (2 locations)
```

Note that two locations are reported, since this function is used in both board_init_f() and board_init_r().

```
(gdb) r
Starting program: /tmp/b/sandbox/u-boot
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

U-Boot 2018.09-00264-ge0c2ba9814-dirty (Sep 22 2018 - 12:21:46 -0600)
```

(continues on next page)

(continued from previous page)

```
DRAM:  128 MiB
MMC:

Breakpoint 1, initcall_run_list (init_sequence=0x5555559619e0 <init_sequence_f>)
    at /scratch/sglass/cosarm/src/third_party/u-boot/files/include/initcall.h:41
41                              printf("initcall sequence %p failed at call %p (err=
↪%d)\n",
(gdb) print *init_fnc_ptr
$1 = (const init_fnc_t) 0x55555559c114 <stdio_add_devices>
(gdb)
```

This approach can be used on normal boards as well as sandbox.

SDL_CONFIG

If sdl-config is on a different path from the default, set the SDL_CONFIG environment variable to the correct pathname before building U-Boot.

Using valgrind / memcheck

It is possible to run U-Boot under valgrind to check memory allocations:

```
valgrind u-boot
```

If you are running sandbox SPL or TPL, then valgrind will not by default notice when U-Boot jumps from TPL to SPL, or from SPL to U-Boot proper. To fix this, use:

```
valgrind --trace-children=yes u-boot
```

Testing

U-Boot sandbox can be used to run various tests, mostly in the test/ directory. These include:

command_ut: Unit tests for command parsing and handling

compression: Unit tests for U-Boot's compression algorithms, useful for security checking. It supports gzip, bzip2, lzma and lzo.

driver model: Run this pytest:

```
./test/py/test.py --bd sandbox --build -k ut_dm -v
```

image: Unit tests for images: test/image/test-imagetools.sh - multi-file images test/image/test-fit.py - FIT images

tracing: test/trace/test-trace.sh tests the tracing system (see README.trace)

verified boot: See test/vboot/vboot_test.sh for this

If you change or enhance any of the above subsystems, you shold write or expand a test and include it with your patch series submission. Test coverage in U-Boot is limited, as we need to work to improve it.

Note that many of these tests are implemented as commands which you can run natively on your board if desired (and enabled).

To run all tests use "make check".

To run a single test in an existing sandbox build, you can use -T to use the test device tree, and -c to select the test:

/tmp/b/sandbox/u-boot -T -c “ut dm pci_busdev”

This runs dm_test_pci_busdev() which is in test/dm/pci.c

Memory Map

Sandbox has its own emulated memory starting at 0. Here are some of the things that are mapped into that memory:

Addr	Config	Usage
0	CONFIG_SYS_FDT_LOAD_ADDR	Device tree
e000	CONFIG_BLOBLIST_ADDR	Blob list
10000	CONFIG_MALLOC_F_ADDR	Early memory allocation
f0000	CONFIG_PRE_CON_BUF_ADDR	Pre-console buffer
100000	CONFIG_TRACE_EARLY_ADDR	Early trace buffer (if enabled)

6.1.8 SuperH

What’s this?

This file contains status information for the port of U-Boot to the Renesas SuperH series of CPUs.

Overview

SuperH has an original boot loader. However, source code is dirty, and maintenance is not done. To improve sharing and the maintenance of the code, Nobuhiro Iwamatsu started the porting to U-Boot in 2007.

Supported CPUs

Renesas SH7750/SH7750R

This CPU has the SH4 core.

Renesas SH7722

This CPU has the SH4AL-DSP core.

Renesas SH7780

This CPU has the SH4A core.

Supported Boards

Hitachi UL MS7750SE01/MS7750RSE01

Board specific code is in board/ms7750se To use this board, type “make ms7750se_config”. Support devices are:

- SCIF
- SDRAM

- NOR Flash
- Marubun PCMCIA

Hitachi UL MS7722SE01

Board specific code is in board/ms7722se To use this board, type "make ms7722se_config". Support devices are:

- SCIF
- SDRAM
- NOR Flash
- Marubun PCMCIA
- SMC91x ethernet

Hitachi UL MS7720ERP01

Board specific code is in board/ms7720se To use this board, type "make ms7720se_config". Support devices are:

- SCIF
- SDRAM
- NOR Flash
- Marubun PCMCIA

Renesas R7780MP

Board specific code is in board/r7780mp To use this board, type "make r7780mp_config". Support devices are:

- SCIF
- DDR-SDRAM
- NOR Flash
- Compact Flash
- ASIX ethernet
- SH7780 PCI bridge
- RTL8110 ethernet

In SuperH, S-record and binary of made u-boot work on the memory. When u-boot is written in the flash, it is necessary to change the address by using 'objcopy':

```
ex) shX-linux-objcopy -Ibinary -Osrec u-boot.bin u-boot.flash.srec
```

Compiler

You can use the following of u-boot to compile.

- SuperH Linux Open site
- KPIT GNU tools

Future

I plan to support the following CPUs and boards.

CPUs

- SH7751R(SH4)

Boards

Many boards ;-)

6.1.9 x86

This document describes the information about U-Boot running on x86 targets, including supported boards, build instructions, todo list, etc.

Status

U-Boot supports running as a coreboot payload on x86. So far only Link (Chromebook Pixel) and QEMU x86 targets have been tested, but it should work with minimal adjustments on other x86 boards since coreboot deals with most of the low-level details.

U-Boot is a main bootloader on Intel Edison board.

U-Boot also supports booting directly from x86 reset vector, without coreboot. In this case, known as bare mode, from the fact that it runs on the 'bare metal', U-Boot acts like a BIOS replacement. The following platforms are supported:

- Bayley Bay CRB
- Cherry Hill CRB
- Congatec QEVAL 2.0 & conga-QA3/E3845
- Cougar Canyon 2 CRB
- Crown Bay CRB
- Galileo
- Link (Chromebook Pixel)
- Minnowboard MAX
- Samus (Chromebook Pixel 2015)
- QEMU x86 (32-bit & 64-bit)

As for loading an OS, U-Boot supports directly booting a 32-bit or 64-bit Linux kernel as part of a FIT image. It also supports a compressed zImage. U-Boot supports loading an x86 VxWorks kernel. Please check README.vxworks for more details.

Build Instructions for U-Boot as BIOS replacement (bare mode)

Building a ROM version of U-Boot (hereafter referred to as u-boot.rom) is a little bit tricky, as generally it requires several binary blobs which are not shipped in the U-Boot source tree. Due to this reason, the u-boot.rom build is not turned on by default in the U-Boot source tree. Firstly, you need turn it on by enabling the ROM build either via an environment variable:

```
$ export BUILD_ROM=y
```

or via configuration:

```
CONFIG_BUILD_ROM=y
```

Both tell the Makefile to build u-boot.rom as a target.

CPU Microcode

Modern CPUs usually require a special bit stream called microcode to be loaded on the processor after power up in order to function properly. U-Boot has already integrated these as hex dumps in the source tree.

SMP Support

On a multicore system, U-Boot is executed on the bootstrap processor (BSP). Additional application processors (AP) can be brought up by U-Boot. In order to have an SMP kernel to discover all of the available processors, U-Boot needs to prepare configuration tables which contain the multi-CPUs information before loading the OS kernel. Currently U-Boot supports generating two types of tables for SMP, called Simple Firmware Interface (SFI) and Multi-Processor (MP) tables. The writing of these two tables are controlled by two Kconfig options GENERATE_SFI_TABLE and GENERATE_MP_TABLE.

Driver Model

x86 has been converted to use driver model for serial, GPIO, SPI, SPI flash, keyboard, real-time clock, USB. Video is in progress.

Device Tree

x86 uses device tree to configure the board thus requires CONFIG_OF_CONTROL to be turned on. Not every device on the board is configured via device tree, but more and more devices will be added as time goes by. Check out the directory arch/x86/dts/ for these device tree source files.

Useful Commands

In keeping with the U-Boot philosophy of providing functions to check and adjust internal settings, there are several x86-specific commands that may be useful:

fsp Display information about Intel Firmware Support Package (FSP). This is only available on platforms which use FSP, mostly Atom.

iod Display I/O memory

iow Write I/O memory

mtrr List and set the Memory Type Range Registers (MTRR). These are used to tell the CPU whether memory is cacheable and if so the cache write mode to use. U-Boot sets up some reasonable values but you can adjust then with this command.

Booting Ubuntu

As an example of how to set up your boot flow with U-Boot, here are instructions for starting Ubuntu from U-Boot. These instructions have been tested on Minnowboard MAX with a SATA drive but are equally applicable on other platforms and other media. There are really only four steps and it's a very simple script, but a more detailed explanation is provided here for completeness.

Note: It is possible to set up U-Boot to boot automatically using syslinux. It could also use the grub.cfg file (/efi/ubuntu/grub.cfg) to obtain the GUID. If you figure these out, please post patches to this README.

Firstly, you will need Ubuntu installed on an available disk. It should be possible to make U-Boot start a USB start-up disk but for now let's assume that you used another boot loader to install Ubuntu.

Use the U-Boot command line to find the UUID of the partition you want to boot. For example our disk is SCSI device 0:

```
=> part list scsi 0

Partition Map for SCSI device 0  --   Partition Type: EFI

   Part      Start LBA       End LBA         Name
     Attributes
     Type GUID
     Partition GUID
   1 0x00000800      0x001007ff      ""
     attrs:  0x0000000000000000
     type:   c12a7328-f81f-11d2-ba4b-00a0c93ec93b
     guid:   9d02e8e4-4d59-408f-a9b0-fd497bc9291c
   2 0x00100800      0x037d8fff      ""
     attrs:  0x0000000000000000
     type:   0fc63daf-8483-4772-8e79-3d69d8477de4
     guid:   965c59ee-1822-4326-90d2-b02446050059
   3 0x037d9000      0x03ba27ff      ""
     attrs:  0x0000000000000000
     type:   0657fd6d-a4ab-43c4-84e5-0933c84b4f4f
     guid:   2c4282bd-1e82-4bcf-a5ff-51dedbf39f17
   =>
```

This shows that your SCSI disk has three partitions. The really long hex strings are called Globally Unique Identifiers (GUIDs). You can look up the 'type' ones here. On this disk the first partition is for EFI and is in VFAT format (DOS/Windows):

```
=> fatls scsi 0:1
            efi/

0 file(s), 1 dir(s)
```

Partition 2 is 'Linux filesystem data' so that will be our root disk. It is in ext2 format:

```
=> ext2ls scsi 0:2
<DIR>       4096 .
<DIR>       4096 ..
<DIR>      16384 lost+found
<DIR>       4096 boot
<DIR>      12288 etc
<DIR>       4096 media
<DIR>       4096 bin
<DIR>       4096 dev
<DIR>       4096 home
<DIR>       4096 lib
<DIR>       4096 lib64
<DIR>       4096 mnt
<DIR>       4096 opt
<DIR>       4096 proc
<DIR>       4096 root
<DIR>       4096 run
```

(continues on next page)

(continued from previous page)

```
<DIR>      12288 sbin
<DIR>       4096 srv
<DIR>       4096 sys
<DIR>       4096 tmp
<DIR>       4096 usr
<DIR>       4096 var
<SYM>         33 initrd.img
<SYM>         30 vmlinuz
<DIR>       4096 cdrom
<SYM>         33 initrd.img.old
=>
```

and if you look in the /boot directory you will see the kernel:

```
=> ext2ls scsi 0:2 /boot
<DIR>       4096 .
<DIR>       4096 ..
<DIR>       4096 efi
<DIR>       4096 grub
         3381262 System.map-3.13.0-32-generic
         1162712 abi-3.13.0-32-generic
          165611 config-3.13.0-32-generic
          176500 memtest86+.bin
          178176 memtest86+.elf
          178680 memtest86+_multiboot.bin
         5798112 vmlinuz-3.13.0-32-generic
          165762 config-3.13.0-58-generic
         1165129 abi-3.13.0-58-generic
         5823136 vmlinuz-3.13.0-58-generic
        19215259 initrd.img-3.13.0-58-generic
         3391763 System.map-3.13.0-58-generic
         5825048 vmlinuz-3.13.0-58-generic.efi.signed
        28304443 initrd.img-3.13.0-32-generic
=>
```

The ‘vmlinuz’ files contain a packaged Linux kernel. The format is a kind of self-extracting compressed file mixed with some ‘setup’ configuration data. Despite its size (uncompressed it is >10MB) this only includes a basic set of device drivers, enough to boot on most hardware types.

The ‘initrd’ files contain a RAM disk. This is something that can be loaded into RAM and will appear to Linux like a disk. Ubuntu uses this to hold lots of drivers for whatever hardware you might have. It is loaded before the real root disk is accessed.

The numbers after the end of each file are the version. Here it is Linux version 3.13. You can find the source code for this in the Linux tree with the tag v3.13. The ‘.0’ allows for additional Linux releases to fix problems, but normally this is not needed. The ‘-58’ is used by Ubuntu. Each time they release a new kernel they increment this number. New Ubuntu versions might include kernel patches to fix reported bugs. Stable kernels can exist for some years so this number can get quite high.

The ‘.efi.signed’ kernel is signed for EFI’s secure boot. U-Boot has its own secure boot mechanism - see this & that. It cannot read .efi files at present.

To boot Ubuntu from U-Boot the steps are as follows:

1. Set up the boot arguments. Use the GUID for the partition you want to boot:

```
=> setenv bootargs root=/dev/disk/by-partuuid/965c59ee-1822-4326-90d2-b02446050059␣
↪ro
```

Here root= tells Linux the location of its root disk. The disk is specified by its GUID, using '/dev/disk/by-partuuid/', a Linux path to a 'directory' containing all the GUIDs Linux has found. When it starts up, there will be a file in that directory with this name in it. It is also possible to use a device name here, see later.

2. Load the kernel. Since it is an ext2/4 filesystem we can do:

```
=> ext2load scsi 0:2 03000000 /boot/vmlinuz-3.13.0-58-generic
```

The address 30000000 is arbitrary, but there seem to be problems with using small addresses (sometimes Linux cannot find the ramdisk). This is 48MB into the start of RAM (which is at 0 on x86).

3. Load the ramdisk (to 64MB):

```
=> ext2load scsi 0:2 04000000 /boot/initrd.img-3.13.0-58-generic
```

4. Start up the kernel. We need to know the size of the ramdisk, but can use a variable for that. U-Boot sets 'filesize' to the size of the last file it loaded:

```
=> zboot 03000000 0 04000000 ${filesize}
```

Type 'help zboot' if you want to see what the arguments are. U-Boot on x86 is quite verbose when it boots a kernel. You should see these messages from U-Boot:

```
Valid Boot Flag
Setup Size = 0x00004400
Magic signature found
Using boot protocol version 2.0c
Linux kernel version 3.13.0-58-generic (buildd@allspice) #97-Ubuntu SMP Wed Jul 8
↪02:56:15 UTC 2015
Building boot_params at 0x00090000
Loading bzImage at address 100000 (5805728 bytes)
Magic signature found
Initial RAM disk at linear address 0x04000000, size 19215259 bytes
Kernel command line: "root=/dev/disk/by-partuuid/965c59ee-1822-4326-90d2-b02446050059
↪ro"

Starting kernel ...
```

U-Boot prints out some bootstage timing. This is more useful if you put the above commands into a script since then it will be faster:

```
Timer summary in microseconds:
       Mark    Elapsed  Stage
          0          0  reset
    241,535    241,535  board_init_r
  2,421,611  2,180,076  id=64
  2,421,790        179  id=65
  2,428,215      6,425  main_loop
 48,860,584 46,432,369  start_kernel

Accumulated time:
               240,329  ahci
             1,422,704  vesa display
```

Now the kernel actually starts (if you want to examine kernel boot up message on the serial console, append "console=ttyS0,115200" to the kernel command line):

```
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
```

(continues on next page)

(continued from previous page)

```
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.13.0-58-generic (buildd@allspice) (gcc version 4.8.2␣
↪(Ubuntu 4.8.2-19ubuntu1) ) #97-Ubuntu SMP Wed Jul 8 02:56:15 UTC 2015 (Ubuntu 3.13.0-
↪58.97-generic 3.13.11-ckt22)
[    0.000000] Command line: root=/dev/disk/by-partuuid/965c59ee-1822-4326-90d2-
↪b02446050059 ro console=ttyS0,115200
```

It continues for a long time. Along the way you will see it pick up your ramdisk:

```
[    0.000000] RAMDISK: [mem 0x04000000-0x05253fff]
...
[    0.788540] Trying to unpack rootfs image as initramfs...
[    1.540111] Freeing initrd memory: 18768K (ffff880004000000 - ffff880005254000)
...
```

Later it actually starts using it:

```
Begin: Running /scripts/local-premount ... done.
```

You should also see your boot disk turn up:

```
[    4.357243] scsi 1:0:0:0: Direct-Access     ATA      ADATA SP310      5.2  PQ: 0␣
↪ANSI: 5
[    4.366860] sd 1:0:0:0: [sda] 62533296 512-byte logical blocks: (32.0 GB/29.8 GiB)
[    4.375677] sd 1:0:0:0: Attached scsi generic sg0 type 0
[    4.381859] sd 1:0:0:0: [sda] Write Protect is off
[    4.387452] sd 1:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't␣
↪support DPO or FUA
[    4.399535]  sda: sda1 sda2 sda3
```

Linux has found the three partitions (sda1-3). Mercifully it doesn't print out the GUIDs. In step 1 above we could have used:

```
setenv bootargs root=/dev/sda2 ro
```

instead of the GUID. However if you add another drive to your board the numbering may change whereas the GUIDs will not. So if your boot partition becomes sdb2, it will still boot. For embedded systems where you just want to boot the first disk, you have that option.

The last thing you will see on the console is mention of plymouth (which displays the Ubuntu start-up screen) and a lot of 'Starting' messages:

```
* Starting Mount filesystems on boot                                    [ OK ]
```

After a pause you should see a login screen on your display and you are done.

If you want to put this in a script you can use something like this:

```
setenv bootargs root=UUID=b2aaf743-0418-4d90-94cc-3e6108d7d968 ro
setenv boot zboot 03000000 0 04000000 \${filesize}
setenv bootcmd "ext2load scsi 0:2 03000000 /boot/vmlinuz-3.13.0-58-generic; ext2load␣
↪scsi 0:2 04000000 /boot/initrd.img-3.13.0-58-generic; run boot"
saveenv
```

The is to tell the shell not to evaluate ${filesize} as part of the setenv command.

You can also bake this behaviour into your build by hard-coding the environment variables if you add this to minnowmax.h:

```
#undef CONFIG_BOOTCOMMAND
#define CONFIG_BOOTCOMMAND      \
        "ext2load scsi 0:2 03000000 /boot/vmlinuz-3.13.0-58-generic; " \
        "ext2load scsi 0:2 04000000 /boot/initrd.img-3.13.0-58-generic; " \
        "run boot"

#undef CONFIG_EXTRA_ENV_SETTINGS
#define CONFIG_EXTRA_ENV_SETTINGS "boot=zboot 03000000 0 04000000 ${filesize}"
```

and change CONFIG_BOOTARGS value in configs/minnowmax_defconfig to:

```
CONFIG_BOOTARGS="root=/dev/sda2 ro"
```

Test with SeaBIOS

SeaBIOS is an open source implementation of a 16-bit x86 BIOS. It can run in an emulator or natively on x86 hardware with the use of U-Boot. With its help, we can boot some OSes that require 16-bit BIOS services like Windows/DOS.

As U-Boot, we have to manually create a table where SeaBIOS gets various system information (eg: E820) from. The table unfortunately has to follow the coreboot table format as SeaBIOS currently supports booting as a coreboot payload.

To support loading SeaBIOS, U-Boot should be built with CONFIG_SEABIOS on. Booting SeaBIOS is done via U-Boot's bootelf command, like below:

```
=> tftp bios.bin.elf;bootelf
Using e1000#0 device
TFTP from server 10.10.0.100; our IP address is 10.10.0.108
...
Bytes transferred = 122124 (1dd0c hex)
## Starting application at 0x000ff06e ...
SeaBIOS (version rel-1.9.0)
...
```

bios.bin.elf is the SeaBIOS image built from SeaBIOS source tree. Make sure it is built as follows:

```
$ make menuconfig
```

Inside the "General Features" menu, select "Build for coreboot" as the "Build Target". Inside the "Debugging" menu, turn on "Serial port debugging" so that we can see something as soon as SeaBIOS boots. Leave other options as in their default state. Then:

```
$ make
...
Total size: 121888  Fixed: 66496  Free: 9184 (used 93.0% of 128KiB rom)
Creating out/bios.bin.elf
```

Currently this is tested on QEMU x86 target with U-Boot chain-loading SeaBIOS to install/boot a Windows XP OS (below for example command to install Windows).

```
# Create a 10G disk.img as the virtual hard disk
$ qemu-img create -f qcow2 disk.img 10G

# Install a Windows XP OS from an ISO image 'winxp.iso'
$ qemu-system-i386 -serial stdio -bios u-boot.rom -hda disk.img -cdrom winxp.iso -smp
 2 -m 512
```

(continues on next page)

(continued from previous page)

```
# Boot a Windows XP OS installed on the virutal hard disk
$ qemu-system-i386 -serial stdio -bios u-boot.rom -hda disk.img -smp 2 -m 512
```

This is also tested on Intel Crown Bay board with a PCIe graphics card, booting SeaBIOS then chain-loading a GRUB on a USB drive, then Linux kernel finally.

If you are using Intel Integrated Graphics Device (IGD) as the primary display device on your board, SeaBIOS needs to be patched manually to get its VGA ROM loaded and run by SeaBIOS. SeaBIOS locates VGA ROM via the PCI expansion ROM register, but IGD device does not have its VGA ROM mapped by this register. Its VGA ROM is packaged as part of u-boot.rom at a configurable flash address which is unknown to SeaBIOS. An example patch is needed for SeaBIOS below:

```
diff --git a/src/optionroms.c b/src/optionroms.c
index 65f7fe0..c7b6f5e 100644
--- a/src/optionroms.c
+++ b/src/optionroms.c
@@ -324,6 +324,8 @@ init_pcirom(struct pci_device *pci, int isvga, u64 *sources)
         rom = deploy_romfile(file);
     else if (RunPCIroms > 1 || (RunPCIroms == 1 && isvga))
         rom = map_pcirom(pci);
+    if (pci->bdf == pci_to_bdf(0, 2, 0))
+        rom = (struct rom_header *)0xfff90000;
     if (! rom)
         // No ROM present.
         return;
```

Note: the patch above expects IGD device is at PCI b.d.f 0.2.0 and its VGA ROM is at 0xfff90000 which corresponds to CONFIG_VGA_BIOS_ADDR on Minnowboard MAX. Change these two accordingly if this is not the case on your board.

Development Flow

These notes are for those who want to port U-Boot to a new x86 platform.

Since x86 CPUs boot from SPI flash, a SPI flash emulator is a good investment. The Dediprog em100 can be used on Linux.

The em100 tool is available here: http://review.coreboot.org/p/em100.git

On Minnowboard Max the following command line can be used:

```
sudo em100 -s -p LOW -d u-boot.rom -c W25Q64DW -r
```

A suitable clip for connecting over the SPI flash chip is here: http://www.dediprog.com/pd/programmer-accessories/EM-TC-8.

This allows you to override the SPI flash contents for development purposes. Typically you can write to the em100 in around 1200ms, considerably faster than programming the real flash device each time. The only important limitation of the em100 is that it only supports SPI bus speeds up to 20MHz. This means that images must be set to boot with that speed. This is an Intel-specific feature - e.g. tools/ifttool has an option to set the SPI speed in the SPI descriptor region.

If your chip/board uses an Intel Firmware Support Package (FSP) it is fairly easy to fit it in. You can follow the Minnowboard Max implementation, for example. Hopefully you will just need to create new files similar to those in arch/x86/cpu/baytrail which provide Bay Trail support.

If you are not using an FSP you have more freedom and more responsibility. The ivybridge support works this way, although it still uses a ROM for graphics and still has binary blobs containing Intel code. You should aim to support all important peripherals on your platform including video and storage. Use the device tree for configuration where possible.

For the microcode you can create a suitable device tree file using the microcode tool:

```
./tools/microcode-tool -d microcode.dat -m <model> create
```

or if you only have header files and not the full Intel microcode.dat database:

```
./tools/microcode-tool -H BAY_TRAIL_FSP_KIT/Microcode/M0130673322.h \
 -H BAY_TRAIL_FSP_KIT/Microcode/M0130679901.h -m all create
```

These are written to arch/x86/dts/microcode/ by default.

Note that it is possible to just add the micrcode for your CPU if you know its model. U-Boot prints this information when it starts:

```
CPU: x86_64, vendor Intel, device 30673h
```

so here we can use the M0130673322 file.

If you platform can display POST codes on two little 7-segment displays on the board, then you can use post_code() calls from C or assembler to monitor boot progress. This can be good for debugging.

If not, you can try to get serial working as early as possible. The early debug serial port may be useful here. See setup_internal_uart() for an example.

During the U-Boot porting, one of the important steps is to write correct PIRQ routing information in the board device tree. Without it, device drivers in the Linux kernel won't function correctly due to interrupt is not working. Please refer to U-Boot doc for the device tree bindings of Intel interrupt router. Here we have more details on the intel,pirq-routing property below.

```
intel,pirq-routing = <
        PCI_BDF(0, 2, 0) INTA PIRQA
        ...
>;
```

As you see each entry has 3 cells. For the first one, we need describe all pci devices mounted on the board. For SoC devices, normally there is a chapter on the chipset datasheet which lists all the available PCI devices. For example on Bay Trail, this is chapter 4.3 (PCI configuration space). For the second one, we can get the interrupt pin either from datasheet or hardware via U-Boot shell. The reliable source is the hardware as sometimes chipset datasheet is not 100% up-to-date. Type 'pci header' plus the device's pci bus/device/function number from U-Boot shell below:

```
=> pci header 0.1e.1
  vendor ID =                 0x8086
  device ID =                 0x0f08
  ...
  interrupt line =            0x09
  interrupt pin =             0x04
  ...
```

It shows this PCI device is using INTD pin as it reports 4 in the interrupt pin register. Repeat this until you get interrupt pins for all the devices. The last cell is the PIRQ line which a particular interrupt pin is mapped to. On Intel chipset, the power-up default mapping is INTA/B/C/D maps to PIRQA/B/C/D. This can be changed by registers in LPC bridge. So far Intel FSP does not touch those registers so we can write down the PIRQ according to the default mapping rule.

Once we get the PIRQ routing information in the device tree, the interrupt allocation and assignment will be done by U-Boot automatically. Now you can enable CONFIG_GENERATE_PIRQ_TABLE for testing Linux kernel using i8259 PIC and CONFIG_GENERATE_MP_TABLE for testing Linux kernel using local APIC and I/O APIC.

This script might be useful. If you feed it the output of 'pci long' from U-Boot then it will generate a device tree fragment with the interrupt configuration for each device (note it needs gawk 4.0.0):

```
$ cat console_output |awk '/PCI/ {device=$4} /interrupt line/ {line=$4} \
     /interrupt pin/ {pin = $4; if (pin != "0x00" && pin != "0xff") \
     {patsplit(device, bdf, "[0-9a-f]+"); \
     printf "PCI_BDF(%d, %d, %d) INT%c PIRQ%c\n", strtonum("0x" bdf[1]), \
     strtonum("0x" bdf[2]), bdf[3], strtonum(pin) + 64, 64 + strtonum(pin)}}'
```

Example output:

```
PCI_BDF(0, 2, 0) INTA PIRQA
PCI_BDF(0, 3, 0) INTA PIRQA
...
```

Porting Hints

Quark-specific considerations

To port U-Boot to other boards based on the Intel Quark SoC, a few things need to be taken care of. The first important part is the Memory Reference Code (MRC) parameters. Quark MRC supports memory-down configuration only. All these MRC parameters are supplied via the board device tree. To get started, first copy the MRC section of arch/x86/dts/galileo.dts to your board's device tree, then change these values by consulting board manuals or your hardware vendor. Available MRC parameter values are listed in include/dt-bindings/mrc/quark.h. The other tricky part is with PCIe. Quark SoC integrates two PCIe root ports, but by default they are held in reset after power on. In U-Boot, PCIe initialization is properly handled as per Quark's firmware writer guide. In your board support codes, you need provide two routines to aid PCIe initialization, which are board_assert_perst() and board_deassert_perst(). The two routines need implement a board-specific mechanism to assert/deassert PCIe PERST# pin. Care must be taken that in those routines that any APIs that may trigger PCI enumeration process are strictly forbidden, as any access to PCIe root port's configuration registers will cause system hang while it is held in reset. For more details, check how they are implemented by the Intel Galileo board support codes in board/intel/galileo/galileo.c.

coreboot

See scripts/coreboot.sed which can assist with porting coreboot code into U-Boot drivers. It will not resolve all build errors, but will perform common transformations. Remember to add attribution to coreboot for new files added to U-Boot. This should go at the top of each file and list the coreboot filename where the code originated.

Debugging ACPI issues with Windows

Windows might cache system information and only detect ACPI changes if you modify the ACPI table versions. So tweak them liberally when debugging ACPI issues with Windows.

ACPI Support Status

Advanced Configuration and Power Interface (ACPI) aims to establish industry-standard interfaces enabling OS-directed configuration, power management, and thermal management of mobile, desktop, and server platforms.

Linux can boot without ACPI with "acpi=off" command line parameter, but with ACPI the kernel gains the capabilities to handle power management. For Windows, ACPI is a must-have firmware feature since Windows Vista. CONFIG_GENERATE_ACPI_TABLE is the config option to turn on ACPI support in U-Boot. This requires Intel ACPI compiler to be installed on your host to compile ACPI DSDT table written in ASL format to AML format. You can get the compiler via "apt-get install iasl" if you are on Ubuntu or download the source from https://www.acpica.org/downloads to compile one by yourself.

Current ACPI support in U-Boot is basically complete. More optional features can be added in the future. The status as of today is:

- Support generating RSDT, XSDT, FACS, FADT, MADT, MCFG tables.
- Support one static DSDT table only, compiled by Intel ACPI compiler.
- Support S0/S3/S4/S5, reboot and shutdown from OS.
- Support booting a pre-installed Ubuntu distribution via 'zboot' command.
- Support installing and booting Ubuntu 14.04 (or above) from U-Boot with the help of SeaBIOS using legacy interface (non-UEFI mode).
- Support installing and booting Windows 8.1/10 from U-Boot with the help of SeaBIOS using legacy interface (non-UEFI mode).
- Support ACPI interrupts with SCI only.

Features that are optional:

- Dynamic AML bytecodes insertion at run-time. We may need this to support SSDT table generation and DSDT fix up.
- SMI support. Since U-Boot is a modern bootloader, we don't want to bring those legacy stuff into U-Boot. ACPI spec allows a system that does not support SMI (a legacy-free system).

ACPI was initially enabled on BayTrail based boards. Testing was done by booting a pre-installed Ubuntu 14.04 from a SATA drive. Installing Ubuntu 14.04 and Windows 8.1/10 to a SATA drive and booting from there is also tested. Most devices seem to work correctly and the board can respond a reboot/shutdown command from the OS.

For other platform boards, ACPI support status can be checked by examining their board defconfig files to see if CONFIG_GENERATE_ACPI_TABLE is set to y.

The S3 sleeping state is a low wake latency sleeping state defined by ACPI spec where all system context is lost except system memory. To test S3 resume with a Linux kernel, simply run "echo mem > /sys/power/state" and kernel will put the board to S3 state where the power is off. So when the power button is pressed again, U-Boot runs as it does in cold boot and detects the sleeping state via ACPI register to see if it is S3, if yes it means we are waking up. U-Boot is responsible for restoring the machine state as it is before sleep. When everything is done, U-Boot finds out the wakeup vector provided by OSes and jump there. To determine whether ACPI S3 resume is supported, check to see if CONFIG_HAVE_ACPI_RESUME is set for that specific board.

Note for testing S3 resume with Windows, correct graphics driver must be installed for your platform, otherwise you won't find "Sleep" option in the "Power" submenu from the Windows start menu.

EFI Support

U-Boot supports booting as a 32-bit or 64-bit EFI payload, e.g. with UEFI. This is enabled with CONFIG_EFI_STUB to boot from both 32-bit and 64-bit UEFI BIOS. U-Boot can also run as an EFI application, with CONFIG_EFI_APP. The CONFIG_EFI_LOADER option, where U-Boot provides an EFI environment to the kernel (i.e. replaces UEFI completely but provides the same EFI run-time services) is supported too. For example, we can even use 'bootefi' command to load a 'u-boot-payload.efi', see below test logs on QEMU.

```
=> load ide 0 3000000 u-boot-payload.efi
489787 bytes read in 138 ms (3.4 MiB/s)
=> bootefi 3000000
Scanning disk ide.blk#0...
Found 2 disks
WARNING: booting without device tree
## Starting EFI application at 03000000 ...
U-Boot EFI Payload
```

(continues on next page)

(continued from previous page)

```
U-Boot 2018.07-rc2 (Jun 23 2018 - 17:12:58 +0800)

CPU: x86_64, vendor AMD, device 663h
DRAM:  2 GiB
MMC:
Video: 1024x768x32
Model: EFI x86 Payload
Net:   e1000: 52:54:00:12:34:56

Warning: e1000#0 using MAC address from ROM
eth0: e1000#0
No controllers found
Hit any key to stop autoboot:  0
```

See *U-Boot on EFI* and *UEFI on U-Boot* for details of EFI support in U-Boot.

Chain-loading

U-Boot can be chain-loaded from another bootloader, such as coreboot or Slim Bootloader. Typically this is done by building for targets 'coreboot' or 'slimbootloader'.

For example, at present we have a 'coreboot' target but this runs very different code from the bare-metal targets, such as coral. There is very little in common between them.

It is useful to be able to boot the same U-Boot on a device, with or without a first-stage bootloader. For example, with chromebook_coral, it is helpful for testing to be able to boot the same U-Boot (complete with FSP) on bare metal and from coreboot. It allows checking of things like CPU speed, comparing registers, ACPI tables and the like.

To do this you can use ll_boot_init() in appropriate places to skip init that has already been done by the previous stage. This works by setting a GD_FLG_NO_LL_INIT flag when U-Boot detects that it is running from another bootloader.

With this feature, you can build a bare-metal target and boot it from coreboot, for example.

Note that this is a development feature only. It is not intended for use in production environments. Also it is not currently part of the automated tests so may break in the future.

SMBIOS tables

To generate SMBIOS tables in U-Boot, for use by the OS, enable the CONFIG_GENERATE_SMBIOS_TABLE option. The easiest way to provide the values to use is via the device tree. For details see device-tree-bindings/sysinfo/smbios.txt

TODO List

- Audio
- Chrome OS verified boot

6.1.10 Xtensa

Xtensa Architecture and Diamond Cores

Xtensa is a configurable processor architecture from Tensilica, Inc. Diamond Cores are pre-configured instances available for license and SoC cores in the same manner as ARM, MIPS, etc.

Xtensa licensees create their own Xtensa cores with selected features and custom instructions, registers and co-processors. The custom core is configured with Tensilica tools and built with Tensilica's Xtensa Processor Generator.

There are an effectively infinite number of CPUs in the Xtensa architecture family. It is, however, not feasible to support individual Xtensa CPUs in U-Boot. Therefore, there is only a single 'xtensa' CPU in the cpu tree of U-Boot.

In the same manner as the Linux port to Xtensa, U-Boot adapts to an individual Xtensa core configuration using a set of macros provided with the particular core. This is part of what is known as the hardware abstraction layer (HAL). For the purpose of U-Boot, the HAL consists only of a few header files. These provide CPP macros that customize sources, Makefiles, and the linker script.

Adding support for an additional processor configuration

The header files for one particular processor configuration are inside a variant-specific directory located in the arch/xtensa/include/asm directory. The name of that directory starts with 'arch-' followed by the name for the processor configuration, for example, arch-dc233c for the Diamond DC233 processor.

core.h: Definitions for the core itself.

The following files are part of the overlay but not used by U-Boot.

tie.h: Co-processors and custom extensions defined in the Tensilica Instruction Extension (TIE) language.

tie-asm.h: Assembly macros to access custom-defined registers and states.

Global Data Pointer, Exported Function Stubs, and the ABI

To support standalone applications launched with the "go" command, U-Boot provides a jump table of entrypoints to exported functions (grep for EXPORT_FUNC). The implementation for Xtensa depends on which ABI (or function calling convention) is used.

Windowed ABI presents unique difficulties with the approach based on keeping global data pointer in dedicated register. Because the register window rotates during a call, there is no register that is constantly available for the gd pointer. Therefore, on xtensa gd is a simple global variable. Another difficulty arises from the requirement to have an 'entry' at the beginning of a function, which rotates the register file and reserves a stack frame. This is an integral part of the windowed ABI implemented in hardware. It makes using a jump table to an arbitrary (separately compiled) function a bit tricky. Use of a simple wrapper is also very tedious due to the need to move all possible register arguments and adjust the stack to handle arguments that cannot be passed in registers. The most efficient approach is to have the jump table perform the 'entry' so as to pretend it's the start of the real function. This requires decoding the target function's 'entry' instruction to determine the stack frame size, and adjusting the stack pointer accordingly, then jumping into the target function just after the 'entry'. Decoding depends on the processor's endianness so uses the HAL. The implementation (12 instructions) is in examples/stubs.c.

Access to Invalid Memory Addresses

U-Boot does not check if memory addresses given as arguments to commands such as "md" are valid. There are two possible types of invalid addresses: an area of physical address space may not be mapped to RAM or peripherals, or in the presence of MMU an area of virtual address space may not be mapped to physical addresses.

Accessing first type of invalid addresses may result in hardware lockup, reading of meaningless data, written data being ignored or an exception, depending on the CPU wiring to the system. Accessing second type of invalid addresses always ends with an exception.

U-Boot for Xtensa provides a special memory exception handler that reports such access attempts and resets the board.

CHAPTER

SEVEN

BOARD-SPECIFIC DOC

These books provide details about board-specific information. They are organized in a vendor subdirectory.

7.1 Board-specific doc

7.1.1 Actions

CUBIEBOARD7

About this

This document describes build and flash steps for Actions S700 SoC based Cubieboard7 board.

Cubieboard7 initial configuration

Default Cubieboard7 comes with pre-installed Android where U-Boot is configured with a bootdelay of 0, entering a prompt by pressing keys does not seem to work.

Though, one can enter ADFU mode and flash debian image(from host machine) where getting into u-boot prompt is easy.

Enter ADFU Mode

Before write the firmware, let the development board entering the ADFU mode: insert one end of the USB cable to the PC, press and hold the ADFU button, and then connect the other end of the USB cable to the Mini USB port of the development board, release the ADFU button, after connecting it will enter the ADFU mode.

Check whether entered ADFU Mode

The user needs to run the following command on the PC side to check if the ADFU device is detected. ID realted to "Actions Semiconductor Co., Ltd" means that the PC side has been correctly detected ADFU device, the development board also enter into the ADFU mode.

```
$ lsusb
Bus 001 Device 005: ID 04f2:b2eb Chicony Electronics Co., Ltd
Bus 001 Device 004: ID 0a5c:21e6 Broadcom Corp. BCM20702 Bluetooth 4.0 [ThinkPad]
Bus 001 Device 003: ID 046d:c534 Logitech, Inc. Unifying Receiver
Bus 001 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
```

(continues on next page)

(continued from previous page)

```
Bus 003 Device 013: ID 10d6:10d6 Actions Semiconductor Co., Ltd
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Flashing debian image

```
$ sudo ./ActionsFWU.py --fw=debian-stretch-desktop-cb7-emmc-v2.0.fw
ActionsFWU.py        : 1.0.150828.0830
libScript.so    : 2.3.150825.0951
libFileSystem.so: 2.3.150825.0952
libProduction.so: 2.3.150915.1527
=====burn all partition====
FW_VER: 3.10.37.180608
3% DOWNLOAD ADFUDEC ...
5% DOWNLOAD BOOT PARA ...
7% SWITCH ADFUDEC ...
12% DOWNLOAD BL31 ...
13% DOWNLOAD BL32 ...
15% DOWNLOAD VMLINUX ...
20% DOWNLOAD INITRD ...
24% DOWNLOAD FDT ...
27% DOWNLOAD ADFUS ...
30% SWITCH ADFUS ...
32% DOWNLOAD MBR ...
35% DOWNLOAD PARTITIONS ...
WRITE_MBRC_PARTITION
35% write p0 size = 2048 : ok
WRITE_BOOT_PARTITION
35% write p1 size = 2048 : ok
WRITE_MISC_PARTITION
36% write p2 size = 98304 : ok
WRITE_SYSTEM_PARTITION
94% write p3 size = 4608000 : ok
FORMAT_SWAP_PARTITION
94% write p4 size = 20480 : ok
95% TRANSFER OVER ...
Firmware upgrade successfully!
```

Debian image can be downloaded from here[1].

Once debian image is flashed, one can get into u-boot prompt by pressing any key and from there run ums command(make sure, usb cable is connected between host and target):

```
owl> ums 0 mmc 1
```

Above command would mount debian image partition on host machine.

Building U-BOOT proper image

```
$ make clean
$ export CROSS_COMPILE=aarch64-linux-gnu-
$ make cubieboard7_defconfig
$ make u-boot-dtb.img -j16
```

u-boot-dtb.img can now be flashed to debian image partition mounted on host machine.

```
$ sudo dd if=u-boot-dtb.img of=/dev/sdb bs=1024 seek=3072
```

[1]: https://pan.baidu.com/s/1uawPr0Jao2HgWFLZCLzHAg#list/path=%2FCubieBoard_Download%2FBoard%2FCubieBoard7%2F%E6%96%B9%E7%B3%96%E6%96%B9%E6%A1%88%E5%BC%80%E5%8F%91%E8%B5%84%E6%96%99%2FImage%2FDebian%2FV2.1-test&parentPath=%2F

7.1.2 Andes Tech

ADP-AG101P

ADP-AG101P is the SoC with AG101 hardcore CPU.

AG101P SoC

AG101P is the mainline SoC produced by Andes Technology using N1213 CPU core with FPU and DDR contoller support. AG101P has integrated both AHB and APB bus and many periphals for application and product development.

Configurations

CONFIG_MEM_REMAP: Doing memory remap is essential for preparing some non-OS or RTOS applications.

CONFIG_SKIP_LOWLEVEL_INIT: If you want to boot this system from SPI ROM and bypass e-bios (the other boot loader on ROM). You should undefine CONFIG_SKIP_LOWLEVEL_INIT in "include/configs/adp-ag101p.h".

Build and boot steps

Build:

1. Prepare the toolchains and make sure the $PATH to toolchains is correct.
2. Use *make adp-ag101p_defconfig* in u-boot root to build the image.

Burn U-Boot to SPI ROM

This section will be added later.

AX25-AE350

AE350 is the mainline SoC produced by Andes Technology using AX25 CPU core base on RISC-V architecture.

AE350 has integrated both AHB and APB bus and many periphals for application and product development.

AX25-AE350 is the SoC with AE350 hardcore CPU.

AX25 is Andes CPU IP to adopt RISC-V architecture.

AX25 Features

CPU Core

- 5-stage in-order execution pipeline
- **Hardware Multiplier**
  - radix-2/radix-4/radix-16/radix-256/fast
- Hardware Divider
- Optional branch prediction
- Machine mode and optional user mode
- Optional performance monitoring

ISA

- RV64I base integer instructions
- RVC for 16-bit compressed instructions
- RVM for multiplication and division instructions

Memory subsystem

- **I & D local memory**
  - Size: 4KB to 16MB
- **Memory subsyetem soft-error protection**
  - Protection scheme: parity-checking or error-checking-and-correction (ECC)
  - Automatic hardware error correction

Bus

- **Interface Protocol**
  - Synchronous AHB (32-bit/64-bit data-width), or
  - Synchronous AXI4 (64-bit data-width)

Power management

- Wait for interrupt (WFI) mode

Debug

- Configurable number of breakpoints: 2/4/8
- **External Debug Module**
  - AHB slave port
- External JTAG debug transport module

Platform Level Interrupt Controller (PLIC)

- AHB slave port
- Configurable number of interrupts: 1-1023
- Configurable number of interrupt priorities: 3/7/15/63/127/255
- Configurable number of targets: 1-16
- Preempted interrupt priority stack

Configurations

CONFIG_SKIP_LOWLEVEL_INIT:

If you want to boot this system from SPI ROM and bypass e-bios (the other boot loader on ROM). You should undefine CONFIG_SKIP_LOWLEVEL_INIT in "include/configs/ax25-ae350.h".

Build and boot steps

Build:

1. Prepare the toolchains and make sure the $PATH to toolchains is correct.
2. Use *make ae350_rv[32|64]_defconfig* in u-boot root to build the image for 32 or 64 bit.

Verification:

1. startup
2. relocation
3. timer driver
4. uart driver
5. mac driver
6. mmc driver
7. spi driver

Steps

1. Define CONFIG_SKIP_LOWLEVEL_INIT to build u-boot which is loaded via gdb from ram.
2. Undefine CONFIG_SKIP_LOWLEVEL_INIT to build u-boot which is booted from spi rom.
3. Ping a server by mac driver
4. Scan sd card and copy u-boot image which is booted from flash to ram by sd driver.
5. Burn this u-boot image to spi rom by spi driver
6. Re-boot u-boot from spi flash with power off and power on.

Messages of U-Boot boot on AE350 board

```
U-Boot 2018.01-rc2-00033-g824f89a (Dec 21 2017 - 16:51:26 +0800)

DRAM:  1 GiB
MMC:   mmc@f0e00000: 0
SF: Detected mx25u1635e with page size 256 Bytes, erase size 4 KiB, total 2 MiB
In:    serial@f0300000
Out:   serial@f0300000
Err:   serial@f0300000
Net:
Warning: mac@e0100000 (eth0) using random MAC address - be:dd:d7:e4:e8:10
eth0: mac@e0100000

RISC-V # version
U-Boot 2018.01-rc2-00033-gb265b91-dirty (Dec 22 2017 - 13:54:21 +0800)
```

(continues on next page)

(continued from previous page)

```
riscv32-unknown-linux-gnu-gcc (GCC) 7.2.0
GNU ld (GNU Binutils) 2.29

RISC-V # setenv ipaddr 10.0.4.200 ;
RISC-V # setenv serverip 10.0.4.97 ;
RISC-V # ping 10.0.4.97 ;
Using mac@e0100000 device
host 10.0.4.97 is alive

RISC-V # mmc rescan
RISC-V # fatls mmc 0:1
   318907   u-boot-ae350-64.bin
     1252   hello_world_ae350_32.bin
   328787   u-boot-ae350-32.bin

3 file(s), 0 dir(s)

RISC-V # sf probe 0:0 50000000 0
SF: Detected mx25u1635e with page size 256 Bytes, erase size 4 KiB, total 2 MiB

RISC-V # sf test 0x100000 0x1000
SPI flash test:
0 erase: 36 ticks, 111 KiB/s 0.888 Mbps
1 check: 29 ticks, 137 KiB/s 1.096 Mbps
2 write: 40 ticks, 100 KiB/s 0.800 Mbps
3 read: 20 ticks, 200 KiB/s 1.600 Mbps
Test passed
0 erase: 36 ticks, 111 KiB/s 0.888 Mbps
1 check: 29 ticks, 137 KiB/s 1.096 Mbps
2 write: 40 ticks, 100 KiB/s 0.800 Mbps
3 read: 20 ticks, 200 KiB/s 1.600 Mbps

RISC-V # fatload mmc 0:1 0x600000 u-boot-ae350-32.bin
reading u-boot-ae350-32.bin
328787 bytes read in 324 ms (990.2 KiB/s)

RISC-V # sf erase 0x0 0x51000
SF: 331776 bytes @ 0x0 Erased: OK

RISC-V # sf write 0x600000 0x0 0x50453
device 0 offset 0x0, size 0x50453
SF: 328787 bytes @ 0x0 Written: OK

RISC-V # crc32 0x600000 0x50453
crc32 for 00600000 ... 00650452 ==> 692dc44a

RISC-V # crc32 0x80000000 0x50453
crc32 for 80000000 ... 80050452 ==> 692dc44a
RISC-V #

*** power-off and power-on, this U-Boot is booted from spi flash ***

U-Boot 2018.01-rc2-00032-gf67dd47-dirty (Dec 21 2017 - 13:56:03 +0800)

DRAM:  1 GiB
MMC:   mmc@f0e00000: 0
```

(continues on next page)

(continued from previous page)

```
SF: Detected mx25u1635e with page size 256 Bytes, erase size 4 KiB, total 2 MiB
In:    serial@f0300000
Out:   serial@f0300000
Err:   serial@f0300000
Net:
Warning: mac@e0100000 (eth0) using random MAC address - ee:4c:58:29:32:f5
eth0: mac@e0100000
RISC-V #
```

Boot bbl and riscv-linux via U-Boot on QEMU

1. Build riscv-linux
2. Build bbl and riscv-linux with –with-payload
3. Prepare ae350.dtb
4. Creating OS-kernel images

```
./mkimage -A riscv -O linux -T kernel -C none -a 0x0000 -e 0x0000 -d bbl.bin␣
↪bootmImage-bbl.bin
Image Name:
Created:      Tue Mar 13 10:06:42 2018
Image Type:   RISC-V Linux Kernel Image (uncompressed)
Data Size:    17901204 Bytes = 17481.64 KiB = 17.07 MiB
Load Address: 00000000
Entry Point:  00000000
```

5. Copy bootmImage-bbl.bin and ae350.dtb to qemu sd card image
6. Message of booting riscv-linux from bbl via u-boot on qemu

```
U-Boot 2018.03-rc4-00031-g2631273 (Mar 13 2018 - 15:02:55 +0800)

DRAM:  1 GiB
main-loop: WARNING: I/O thread spun for 1000 iterations
MMC:   mmc@f0e00000: 0
Loading Environment from SPI Flash... *** Warning - spi_flash_probe_bus_cs() failed,␣
↪using default environment

Failed (-22)
In:    serial@f0300000
Out:   serial@f0300000
Err:   serial@f0300000
Net:
Warning: mac@e0100000 (eth0) using random MAC address - 02:00:00:00:00:00
eth0: mac@e0100000
RISC-V # mmc rescan
RISC-V # mmc part

Partition Map for MMC device 0  --   Partition Type: DOS

Part    Start Sector    Num Sectors     UUID            Type
RISC-V # fatls mmc 0:0
 17901268   bootmImage-bbl.bin
     1954   ae2xx.dtb

```

(continues on next page)

(continued from previous page)

```
2 file(s), 0 dir(s)

RISC-V # fatload mmc 0:0 0x00600000 bootmImage-bbl.bin
17901268 bytes read in 4642 ms (3.7 MiB/s)
RISC-V # fatload mmc 0:0 0x2000000 ae350.dtb
1954 bytes read in 1 ms (1.9 MiB/s)
RISC-V # setenv bootm_size 0x2000000
RISC-V # setenv fdt_high 0x1f00000
RISC-V # bootm 0x00600000 - 0x2000000
## Booting kernel from Legacy Image at 00600000 ...
   Image Name:
   Image Type:   RISC-V Linux Kernel Image (uncompressed)
   Data Size:    17901204 Bytes = 17.1 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 02000000
   Booting using the fdt blob at 0x2000000
   Loading Kernel Image ... OK
   Loading Device Tree to 0000000001efc000, end 0000000001eff7a1 ... OK
[    0.000000] OF: fdt: Ignoring memory range 0x0 - 0x200000
[    0.000000] Linux version 4.14.0-00046-gf3e439f-dirty (rick@atcsqa06) (gcc version␣
↪7.1.1 20170509 (GCC)) #1 Tue Jan 9 16:34:25 CST 2018
[    0.000000] bootconsole [early0] enabled
[    0.000000] Initial ramdisk at: 0xffffffe000016a98 (12267008 bytes)
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000000200000-0x000000007fffffff]
[    0.000000]   Normal   empty
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000000200000-0x000000007fffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000000200000-0x000000007fffffff]
[    0.000000] elf_hwcap is 0x112d
[    0.000000] random: fast init done
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 516615
[    0.000000] Kernel command line: console=ttyS0,38400n8 earlyprintk=uart8250-32bit,
↪0xf0300000 debug loglevel=7
[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
[    0.000000] Dentry cache hash table entries: 262144 (order: 9, 2097152 bytes)
[    0.000000] Inode-cache hash table entries: 131072 (order: 8, 1048576 bytes)
[    0.000000] Sorting __ex_table...
[    0.000000] Memory: 2047832K/2095104K available (1856K kernel code, 204K rwdata,␣
↪532K rodata, 12076K init, 756K bss, 47272K reserved, 0K cma-reserved)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
[    0.000000] NR_IRQS: 0, nr_irqs: 0, preallocated irqs: 0
[    0.000000] riscv,cpu_intc,0: 64 local interrupts mapped
[    0.000000] riscv,plic0,e4000000: mapped 31 interrupts to 1/2 handlers
[    0.000000] clocksource: riscv_clocksource: mask: 0xffffffffffffffff max_cycles:␣
↪0x24e6a1710, max_idle_ns: 440795202120 ns
[    0.000000] Calibrating delay loop (skipped), value calculated using timer␣
↪frequency.. 20.00 BogoMIPS (lpj=40000)
[    0.000000] pid_max: default: 32768 minimum: 301
[    0.004000] Mount-cache hash table entries: 4096 (order: 3, 32768 bytes)
[    0.004000] Mountpoint-cache hash table entries: 4096 (order: 3, 32768 bytes)
[    0.056000] devtmpfs: initialized
[    0.060000] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_
↪ns: 7645041785100000 ns
```

(continues on next page)

(continued from previous page)

```
[    0.064000] futex hash table entries: 256 (order: 0, 6144 bytes)
[    0.068000] NET: Registered protocol family 16
[    0.080000] vgaarb: loaded
[    0.084000] clocksource: Switched to clocksource riscv_clocksource
[    0.088000] NET: Registered protocol family 2
[    0.092000] TCP established hash table entries: 16384 (order: 5, 131072 bytes)
[    0.096000] TCP bind hash table entries: 16384 (order: 5, 131072 bytes)
[    0.096000] TCP: Hash tables configured (established 16384 bind 16384)
[    0.100000] UDP hash table entries: 1024 (order: 3, 32768 bytes)
[    0.100000] UDP-Lite hash table entries: 1024 (order: 3, 32768 bytes)
[    0.104000] NET: Registered protocol family 1
[    0.616000] Unpacking initramfs...
[    1.220000] workingset: timestamp_bits=62 max_order=19 bucket_order=0
[    1.244000] io scheduler noop registered
[    1.244000] io scheduler cfq registered (default)
[    1.244000] io scheduler mq-deadline registered
[    1.248000] io scheduler kyber registered
[    1.360000] Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
[    1.368000] console [ttyS0] disabled
[    1.372000] f0300000.serial: ttyS0 at MMIO 0xf0300020 (irq = 10, base_baud =
↪1228800) is a 16550A
[    1.392000] console [ttyS0] enabled
[    1.392000] ftmac100: Loading version 0.2 ...
[    1.396000] ftmac100 e0100000.mac eth0: irq 8, mapped at ffffffd002005000
[    1.400000] ftmac100 e0100000.mac eth0: generated random MAC address
↪6e:ac:c3:92:36:c0
[    1.404000] IR NEC protocol handler initialized
[    1.404000] IR RC5(x/sz) protocol handler initialized
[    1.404000] IR RC6 protocol handler initialized
[    1.404000] IR JVC protocol handler initialized
[    1.408000] IR Sony protocol handler initialized
[    1.408000] IR SANYO protocol handler initialized
[    1.408000] IR Sharp protocol handler initialized
[    1.408000] IR MCE Keyboard/mouse protocol handler initialized
[    1.412000] IR XMP protocol handler initialized
[    1.456000] ftsdc010 f0e00000.mmc: mmc0 - using hw SDIO IRQ
[    1.464000] bootconsole [early0] uses init memory and must be disabled even before
↪the real one is ready
[    1.464000] bootconsole [early0] disabled
[    1.508000] Freeing unused kernel memory: 12076K
[    1.512000] This architecture does not have kernel memory protection.
[    1.520000] mmc0: new SD card at address 4567
[    1.524000] mmcblk0: mmc0:4567 QEMU! 20.0 MiB
[    1.844000]  mmcblk0:
Wed Dec  1 10:00:00 CST 2010
/ #
```

Running U-Boot SPL

The U-Boot SPL will boot in M mode and load the FIT image which include OpenSBI and U-Boot proper images. After loading progress, it will jump to OpenSBI first and then U-Boot proper which will run in S mode.

How to build U-Boot SPL

Before building U-Boot SPL, OpenSBI must be build first. OpenSBI can be cloned and build for AE350 as below:

```
git clone https://github.com/riscv/opensbi.git
cd opensbi
make PLATFORM=andes/ae350
```

Copy OpenSBI FW_DYNAMIC image (buildplatformandesae350firmwarefw_dynamic.bin) into U-Boot root directory

How to build U-Boot SPL booting from RAM

With ae350_rv[32|64]_spl_defconfigs:

U-Boot SPL will be loaded by gdb or FSBL and runs in RAM in machine mode and then load FIT image from RAM device on AE350.

How to build U-Boot SPL booting from ROM

With ae350_rv[32|64]_spl_xip_defconfigs:

U-Boot SPL can be burned into SPI flash and run in flash in machine mode and then load FIT image from SPI flash or MMC device on AE350.

Messages of U-Boot SPL boots Kernel on AE350 board

```
U-Boot SPL 2020.01-rc1-00292-g67a3313-dirty (Nov 14 2019 - 11:26:21 +0800)
Trying to boot from RAM

OpenSBI v0.5-1-gdd8ef28 (Nov 14 2019 11:08:39)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : Andes AE350
Platform HART Features : RV64ACIMSUX
Platform Max HARTs     : 4
Current Hart           : 0
Firmware Base          : 0x0
Firmware Size          : 84 KB
Runtime SBI Version    : 0.2

PMP0: 0x0000000000000000-0x000000000001ffff (A)
PMP1: 0x0000000000000000-0x00000001ffffffff (A,R,W,X)


U-Boot 2020.01-rc1-00292-g67a3313-dirty (Nov 14 2019 - 11:26:21 +0800)

```

(continues on next page)

(continued from previous page)

```
DRAM:  1 GiB
Flash: 64 MiB
MMC:   mmc@f0e00000: 0
Loading Environment from SPI Flash... SF: Detected mx25u1635e with page size 256 Bytes,
↪ erase size 4 KiB, total 2 MiB
OK
In:    serial@f0300000
Out:   serial@f0300000
Err:   serial@f0300000
Net:   no alias for ethernet0

Warning: mac@e0100000 (eth0) using random MAC address - a2:ae:93:7b:cc:8f
eth0: mac@e0100000
Hit any key to stop autoboot:  0
6455 bytes read in 31 ms (203.1 KiB/s)
20421684 bytes read in 8647 ms (2.3 MiB/s)
## Booting kernel from Legacy Image at 00600000 ...
   Image Name:
   Image Type:   RISC-V Linux Kernel Image (uncompressed)
   Data Size:    20421620 Bytes = 19.5 MiB
   Load Address: 00200000
   Entry Point:  00200000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 20000000
   Booting using the fdt blob at 0x20000000
   Loading Kernel Image
   Loading Device Tree to 000000001effb000, end 000000001efff936 ... OK

Starting kernel ...

OF: fdt: Ignoring memory range 0x0 - 0x200000
Linux version 4.17.0-00253-g49136e10bcb2 (sqa@atcsqa07) (gcc version 7.3.0 (2019-04-06_
↪nds64le-linux-glibc-v5_experimental)) #1 SMP PREEMPT Sat Apr 6 23:41:49 CST 2019
bootconsole [early0] enabled
Initial ramdisk at: 0x        (ptrval) (13665712 bytes)
Zone ranges:
  DMA32    [mem 0x0000000000200000-0x000000003fffffff]
  Normal   empty
Movable zone start for each node
Early memory node ranges
  node   0: [mem 0x0000000000200000-0x000000003fffffff]
Initmem setup node 0 [mem 0x0000000000200000-0x000000003fffffff]
software IO TLB [mem 0x3b1f8000-0x3f1f8000] (64MB) mapped at [        (ptrval)-
↪(ptrval)]
elf_platform is rv64i2p0m2p0a2p0c2p0xv5-0p0
compatible privileged spec version 1.10
percpu: Embedded 16 pages/cpu @        (ptrval) s28184 r8192 d29160 u65536
Built 1 zonelists, mobility grouping on.  Total pages: 258055
Kernel command line: console=ttyS0,38400n8 debug loglevel=7
log_buf_len individual max cpu contribution: 4096 bytes
log_buf_len total cpu_extra contributions: 12288 bytes
log_buf_len min size: 16384 bytes
log_buf_len: 32768 bytes
early log buf free: 14608(89%)
Dentry cache hash table entries: 131072 (order: 8, 1048576 bytes)
Inode-cache hash table entries: 65536 (order: 7, 524288 bytes)
```

(continues on next page)

(continued from previous page)

```
Sorting __ex_table...
Memory: 944428K/1046528K available (3979K kernel code, 246K rwdata, 1490K rodata,␣
↪13523K init, 688K bss, 102100K reserved, 0K cma-reserved)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
Preemptible hierarchical RCU implementation.
        Tasks RCU enabled.
NR_IRQS: 72, nr_irqs: 72, preallocated irqs: 0
riscv,cpu_intc,0: 64 local interrupts mapped
riscv,cpu_intc,1: 64 local interrupts mapped
riscv,cpu_intc,2: 64 local interrupts mapped
riscv,cpu_intc,3: 64 local interrupts mapped
riscv,plic0,e4000000: mapped 71 interrupts to 8/8 handlers
clocksource: riscv_clocksource: mask: 0xffffffffffffffff max_cycles: 0x1bacf917bf, max_
↪idle_ns: 881590412290 ns
sched_clock: 64 bits at 60MHz, resolution 16ns, wraps every 4398046511098ns
Console: colour dummy device 40x30
Calibrating delay loop (skipped), value calculated using timer frequency.. 120.00␣
↪BogoMIPS (lpj=600000)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 2048 (order: 2, 16384 bytes)
Mountpoint-cache hash table entries: 2048 (order: 2, 16384 bytes)
Hierarchical SRCU implementation.
smp: Bringing up secondary CPUs ...
CPU0: online
CPU2: online
CPU3: online
smp: Brought up 1 node, 4 CPUs
devtmpfs: initialized
random: get_random_u32 called from bucket_table_alloc+0x198/0x1d8 with crng_init=0
clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns:␣
↪19112604462750000 ns
futex hash table entries: 1024 (order: 4, 65536 bytes)
NET: Registered protocol family 16
Advanced Linux Sound Architecture Driver Initialized.
clocksource: Switched to clocksource riscv_clocksource
NET: Registered protocol family 2
tcp_listen_portaddr_hash hash table entries: 512 (order: 1, 8192 bytes)
TCP established hash table entries: 8192 (order: 4, 65536 bytes)
TCP bind hash table entries: 8192 (order: 5, 131072 bytes)
TCP: Hash tables configured (established 8192 bind 8192)
UDP hash table entries: 512 (order: 2, 16384 bytes)
UDP-Lite hash table entries: 512 (order: 2, 16384 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
Unpacking initramfs...
workingset: timestamp_bits=62 max_order=18 bucket_order=0
NFS: Registering the id_resolver key type
Key type id_resolver registered
Key type id_legacy registered
nfs4filelayout_init: NFSv4 File Layout Driver Registering...
io scheduler noop registered
io scheduler cfq registered (default)
io scheduler mq-deadline registered
```

(continues on next page)

(continued from previous page)

```
io scheduler kyber registered
Console: switching to colour frame buffer device 40x30
Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
console [ttyS0] disabled
f0300000.serial: ttyS0 at MMIO 0xf0300020 (irq = 20, base_baud = 1228800) is a 16550A
console [ttyS0] enabled
console [ttyS0] enabled
bootconsole [early0] disabled
bootconsole [early0] disabled
loop: module loaded
tun: Universal TUN/TAP device driver, 1.6
ftmac100: Loading version 0.2 ...
ftmac100 e0100000.mac eth0: irq 21, mapped at         (ptrval)
ftmac100 e0100000.mac eth0: generated random MAC address 4e:fd:bd:f3:04:fc
ftsdc010 f0e00000.mmc: mmc0 - using hw SDIO IRQ
mmc0: new SDHC card at address d555
ftssp010 card registered!
mmcblk0: mmc0:d555 SD04G 3.79 GiB
NET: Registered protocol family 10
 mmcblk0: p1
Segment Routing with IPv6
sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
NET: Registered protocol family 17
NET: Registered protocol family 15
ALSA device list:
  #0: ftssp_ac97 controller
Freeing unused kernel memory: 13520K
This architecture does not have kernel memory protection.
Sysinit starting
Sat Apr  6 23:33:53 CST 2019
nfs4flexfilelayout_init: NFSv4 Flexfile Layout Driver Registering...
```

~ #

7.1.3 Amlogic

Hardware Support Matrix

An up-do-date matrix is also available on: http://linux-meson.com

This matrix concerns the actual source code version.

	S905	S905X S805X	S912 S905D	A113X	S905X2 S905D2 S905Y2	S922X A311D	S905X3 S905D3
Boards	Odroid-C2 Nanopi-K2 P200 P201	P212 Khadas-VIM LibreTech-CC LibreTech-AC	Khadas VIM2 Libretech-PC	S400	U200 SEI510	Odroid-N2 Khadas-VIM3	SEI610 Khadas-VIM3L Odroid-C4
UART	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Pinctrl/GPIO	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Clock Control	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
PWM	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Reset Control	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Infrared Decoder	No	No	No	No	No	No	No
Ethernet	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Multi-core	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Fuse access	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
SPI (FC)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No
SPI (CC)	No	No	No	No	No	No	No
I2C	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
USB	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
USB OTG	No	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
eMMC	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
SDCard	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
NAND	No	No	No	No	No	No	No
ADC	**Yes**	**Yes**	**Yes**	No	No	No	No
CVBS Output	**Yes**	**Yes**	**Yes**	*N/A*	**Yes**	**Yes**	**Yes**
HDMI Output	**Yes**	**Yes**	**Yes**	*N/A*	**Yes**	**Yes**	**Yes**
CEC	No	No	No	*N/A*	No	No	No
Thermal Sensor	No	No	No	No	No	No	No
LCD/LVDS Output	No	*N/A*	No	No	No	No	No
MIPI DSI Output	*N/A*	*N/A*	*N/A*	No	No	No	No
SoC (version) information	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Board Documentation

U-Boot for Khadas VIM2

Khadas VIM2 is an Open Source DIY Box manufactured by Shenzhen Wesion Technology Co., Ltd with the following specifications:

- Amlogic S912 ARM Cortex-A53 octo-core SoC @ 1.5GHz
- ARM Mali T860 GPU

- 2/3GB DDR4 SDRAM
- 10/100/1000 Ethernet
- HDMI 2.0 4K/60Hz display
- 40-pin GPIO header
- 2 x USB 2.0 Host, 1 x USB 2.0 Type-C OTG
- 16GB/32GB/64GB eMMC
- 2MB SPI Flash
- microSD
- SDIO Wifi Module, Bluetooth
- Two channels IR receiver

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make khadas-vim2_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/khadas/u-boot -b khadas-vim-v2015.01 vim-u-boot
$ cd vim-u-boot
$ make kvim2_defconfig
$ make
$ export FIPDIR=$PWD/fip
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/gxl/bl2.bin fip/
$ cp $FIPDIR/gxl/acs.bin fip/
$ cp $FIPDIR/gxl/bl21.bin fip/
$ cp $FIPDIR/gxl/bl30.bin fip/
$ cp $FIPDIR/gxl/bl301.bin fip/
$ cp $FIPDIR/gxl/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
    fip/bl30.bin \
```

(continues on next page)

(continued from previous page)

```
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ python $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0

$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl30_new.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl31.img
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl33.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl2sig --input fip/bl2_new.bin --output fip/bl2.n.bin.
↪sig
$ $FIPDIR/gxl/aml_encrypt_gxl --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Khadas VIM3L

Khadas VIM3L is a single board computer manufactured by Shenzhen Wesion Technology Co., Ltd. with the following specifications:

- Amlogic S905D3 Arm Cortex-A55 quad-core SoC
- 2GB LPDDR4 SDRAM
- Gigabit Ethernet
- HDMI 2.1 display
- 40-pin GPIO header
- 1 x USB 3.0 Host, 1 x USB 2.0 Host
- eMMC, microSD
- M.2
- Infrared receiver

Schematics are available on the manufacturer website.

PCIe Setup

The VIM3 on-board MCU can mux the PCIe/USB3.0 shared differential lines using a FUSB340TMX USB 3.1 SuperSpeed Data Switch between an USB3.0 Type A connector and a M.2 Key M slot. The PHY driving these differential lines is shared between the USB3.0 controller and the PCIe Controller, thus only a single controller can use it.

To setup for PCIe, run the following commands from U-Boot:

```
i2c dev i2c@5000
i2c mw 0x18 0x33 1
```

Then power-cycle the board.

To set back to USB3.0, run the following commands from U-Boot:

```
i2c dev i2c@5000
i2c mw 0x18 0x33 0
```

Then power-cycle the board.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make khadas-vim3l_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH

$ DIR=vim3l-u-boot
$ git clone --depth 1 \
   https://github.com/khadas/u-boot.git -b khadas-vims-v2015.01 \
   $DIR

$ cd vim3l-u-boot
$ make kvim3l_defconfig
$ make CROSS_COMPILE=aarch64-none-elf-
$ export UBOOTDIR=$PWD
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ wget https://github.com/BayLibre/u-boot/releases/download/v2017.11-libretech-cc/blx_
↪fix_g12a.sh -O fip/blx_fix.sh
```

(continues on next page)

(continued from previous page)

```
$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/khadas/kvim3l/firmware/acs.bin fip/
$ cp $UBOOTDIR/fip/g12a/bl2.bin fip/
$ cp $UBOOTDIR/fip/g12a/bl30.bin fip/
$ cp $UBOOTDIR/fip/g12a/bl31.img fip/
$ cp $UBOOTDIR/fip/g12a/ddr3_1d.fw fip/
$ cp $UBOOTDIR/fip/g12a/ddr4_1d.fw fip/
$ cp $UBOOTDIR/fip/g12a/ddr4_2d.fw fip/
$ cp $UBOOTDIR/fip/g12a/diag_lpddr4.fw fip/
$ cp $UBOOTDIR/fip/g12a/lpddr3_1d.fw fip/
$ cp $UBOOTDIR/fip/g12a/lpddr4_1d.fw fip/
$ cp $UBOOTDIR/fip/g12a/lpddr4_2d.fw fip/
$ cp $UBOOTDIR/fip/g12a/piei.fw fip/
$ cp $UBOOTDIR/fip/g12a/aml_ddr.fw fip/
$ cp u-boot.bin fip/bl33.bin

$ bash fip/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ bash fip/blx_fix.sh \
    fip/bl2.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/acs.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl30sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.g12a.enc \
                                    --level v3
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl3sig --input fip/bl30_new.bin.g12a.enc \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33 --compress lz4
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc \
            --ddrfw1 fip/ddr4_1d.fw \
            --ddrfw2 fip/ddr4_2d.fw \
```

(continues on next page)

(continued from previous page)

```
            --ddrfw3 fip/ddr3_1d.fw \
            --ddrfw4 fip/piei.fw \
            --ddrfw5 fip/lpddr4_1d.fw \
            --ddrfw6 fip/lpddr4_2d.fw \
            --ddrfw7 fip/diag_lpddr4.fw \
            --ddrfw8 fip/aml_ddr.fw \
            --ddrfw9 fip/lpddr3_1d.fw \
            --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Khadas VIM3

Khadas VIM3 is a single board computer manufactured by Shenzhen Wesion Technology Co., Ltd. with the following specifications:

- Amlogic A311D Arm Cortex-A53 dual-core + Cortex-A73 quad-core SoC
- 4GB LPDDR4 SDRAM
- Gigabit Ethernet
- HDMI 2.1 display
- 40-pin GPIO header
- 1 x USB 3.0 Host, 1 x USB 2.0 Host
- eMMC, microSD
- M.2
- Infrared receiver

Schematics are available on the manufacturer website.

PCIe Setup

The VIM3 on-board MCU can mux the PCIe/USB3.0 shared differential lines using a FUSB340TMX USB 3.1 SuperSpeed Data Switch between an USB3.0 Type A connector and a M.2 Key M slot. The PHY driving these differential lines is shared between the USB3.0 controller and the PCIe Controller, thus only a single controller can use it.

To setup for PCIe, run the following commands from U-Boot:

```
i2c dev i2c@5000
i2c mw 0x18 0x33 1
```

Then power-cycle the board.

To set back to USB3.0, run the following commands from U-Boot:

```
i2c dev i2c@5000
i2c mw 0x18 0x33 0
```

Then power-cycle the board.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make khadas-vim3_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH

$ DIR=vim3-u-boot
$ git clone --depth 1 \
   https://github.com/khadas/u-boot.git -b khadas-vims-v2015.01 \
   $DIR

$ cd vim3-u-boot
$ make kvim3_defconfig
$ make CROSS_COMPILE=aarch64-none-elf-
$ export UBOOTDIR=$PWD
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ wget https://github.com/BayLibre/u-boot/releases/download/v2017.11-libretech-cc/blx_
↪fix_g12a.sh -O fip/blx_fix.sh
$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/khadas/kvim3/firmware/acs.bin fip/
$ cp $UBOOTDIR/fip/g12b/bl2.bin fip/
$ cp $UBOOTDIR/fip/g12b/bl30.bin fip/
$ cp $UBOOTDIR/fip/g12b/bl31.img fip/
$ cp $UBOOTDIR/fip/g12b/ddr3_1d.fw fip/
$ cp $UBOOTDIR/fip/g12b/ddr4_1d.fw fip/
$ cp $UBOOTDIR/fip/g12b/ddr4_2d.fw fip/
$ cp $UBOOTDIR/fip/g12b/diag_lpddr4.fw fip/
$ cp $UBOOTDIR/fip/g12b/lpddr3_1d.fw fip/
$ cp $UBOOTDIR/fip/g12b/lpddr4_1d.fw fip/
$ cp $UBOOTDIR/fip/g12b/lpddr4_2d.fw fip/
$ cp $UBOOTDIR/fip/g12b/piei.fw fip/
$ cp $UBOOTDIR/fip/g12b/aml_ddr.fw fip/
$ cp u-boot.bin fip/bl33.bin

$ bash fip/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
```

(continues on next page)

(continued from previous page)

```
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ bash fip/blx_fix.sh \
    fip/bl2.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/acs.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl30sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.g12a.enc \
                                    --level v3
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl3sig --input fip/bl30_new.bin.g12a.enc \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33 --compress lz4
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc \
            --ddrfw1 fip/ddr4_1d.fw \
            --ddrfw2 fip/ddr4_2d.fw \
            --ddrfw3 fip/ddr3_1d.fw \
            --ddrfw4 fip/piei.fw \
            --ddrfw5 fip/lpddr4_1d.fw \
            --ddrfw6 fip/lpddr4_2d.fw \
            --ddrfw7 fip/diag_lpddr4.fw \
            --ddrfw8 fip/aml_ddr.fw \
            --ddrfw9 fip/lpddr3_1d.fw \
            --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Khadas VIM

Khadas VIM is an Open Source DIY Box manufactured by Shenzhen Wesion Technology Co., Ltd with the following specifications:

- Amlogic S905X ARM Cortex-A53 quad-core SoC @ 1.5GHz
- ARM Mali 450 GPU
- 2GB DDR3 SDRAM
- 10/100 Ethernet
- HDMI 2.0 4K/60Hz display
- 40-pin GPIO header
- 2 x USB 2.0 Host, 1 x USB 2.0 Type-C OTG
- 8GB/16GBeMMC
- microSD
- SDIO Wifi Module, Bluetooth
- Two channels IR receiver

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make khadas-vim_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/khadas/u-boot -b Vim vim-u-boot
$ cd vim-u-boot
$ make kvim_defconfig
$ make CROSS_COMPILE=aarch64-none-elf-
$ export FIPDIR=$PWD/fip
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/gxl/bl2.bin fip/
$ cp $FIPDIR/gxl/acs.bin fip/
$ cp $FIPDIR/gxl/bl21.bin fip/
$ cp $FIPDIR/gxl/bl30.bin fip/
$ cp $FIPDIR/gxl/bl301.bin fip/
$ cp $FIPDIR/gxl/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
```

(continues on next page)

(continued from previous page)

```
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ python $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0

$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl30_new.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl31.img
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl33.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl2sig --input fip/bl2_new.bin --output fip/bl2.n.bin.
↪sig
$ $FIPDIR/gxl/aml_encrypt_gxl --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for LibreTech AC

LibreTech AC is a single board computer manufactured by Libre Technology with the following specifications:

- Amlogic S805X ARM Cortex-A53 quad-core SoC @ 1.2GHz
- ARM Mali 450 GPU
- 512MiB DDR4 SDRAM
- 10/100 Ethernet
- HDMI 2.0 4K/60Hz display
- 40-pin GPIO header
- 4 x USB 2.0 Host
- eMMC, SPI NOR Flash
- Infrared receiver

Schematics are available on the manufacturer website.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make libretech-ac_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b libretech-ac amlogic-u-boot
$ cd amlogic-u-boot
$ wget https://raw.githubusercontent.com/BayLibre/u-boot/libretech-cc/fip/blx_fix.sh
$ make libretech_ac_defconfig
$ make
$ export UBOOTDIR=$PWD
```

Download the latest Amlogic Buildroot package, and extract it :

```
$ wget http://openlinux2.amlogic.com:8000/ARM/filesystem/Linux_BSP/buildroot_openlinux_
↪kernel_4.9_fbdev_20180418.tar.gz
$ tar xfz buildroot_openlinux_kernel_4.9_fbdev_20180418.tar.gz buildroot_openlinux_
↪kernel_4.9_fbdev_20180418/bootloader
$ export BRDIR=$PWD/buildroot_openlinux_kernel_4.9_fbdev_20180418
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/amlogic/libretech_ac/firmware/bl21.bin fip/
$ cp $UBOOTDIR/build/board/amlogic/libretech_ac/firmware/acs.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl2/bin/gxl/bl2.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl30/bin/gxl/bl30.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl31/bin/gxl/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ sh $UBOOTDIR/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ $BRDIR/bootloader/uboot-repo/fip/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.
↪bin 0
```

(continues on next page)

(continued from previous page)

```
$ sh $UBOOTDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $BRDIR/bootloader/uboot-repo/fip/gxl/aml_encrypt_gxl --bl3enc --input fip/bl30_new.
↪bin
$ $BRDIR/bootloader/uboot-repo/fip/gxl/aml_encrypt_gxl --bl3enc --input fip/bl31.img
$ $BRDIR/bootloader/uboot-repo/fip/gxl/aml_encrypt_gxl --bl3enc --input fip/bl33.bin
$ $BRDIR/bootloader/uboot-repo/fip/gxl/aml_encrypt_gxl --bl2sig --input fip/bl2_new.
↪bin --output fip/bl2.n.bin.sig
$ $BRDIR/bootloader/uboot-repo/fip/gxl/aml_encrypt_gxl --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for LibreTech CC

LibreTech CC is a single board computer manufactured by Libre Technology with the following specifications:

- Amlogic S905X ARM Cortex-A53 quad-core SoC @ 1.5GHz
- ARM Mali 450 GPU
- 2GB DDR3 SDRAM
- 10/100 Ethernet
- HDMI 2.0 4K/60Hz display
- 40-pin GPIO header
- 4 x USB 2.0 Host
- eMMC, microSD
- Infrared receiver

Schematics are available on the manufacturer website.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make libretech-cc_defconfig
$ make
```

Image creation

To boot the system, u-boot must be combined with several earlier stage bootloaders:

- bl2.bin: vendor-provided binary blob
- bl21.bin: built from vendor u-boot source
- bl30.bin: vendor-provided binary blob
- bl301.bin: built from vendor u-boot source
- bl31.bin: vendor-provided binary blob
- acs.bin: built from vendor u-boot source

These binaries and the tools required below have been collected and prebuilt for convenience at <https://github.com/BayLibre/u-boot/releases/>

Download and extract the libretech-cc release from there, and set FIPDIR to point to the *fip* subdirectory.

```
$ export FIPDIR=/path/to/extracted/fip
```

Alternatively, you can obtain the original vendor u-boot tree which contains the required blobs and sources, and build yourself. Note that old compilers are required for this to build. The compilers here are suggested by Amlogic, and they are 32-bit x86 binaries.

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b libretech-cc amlogic-u-boot
$ cd amlogic-u-boot
$ make libretech_cc_defconfig
$ make
$ export FIPDIR=$PWD/fip
```

Once you have the binaries available (either through the prebuilt download, or having built the vendor u-boot yourself), you can then proceed to glue everything together. Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/gxl/bl2.bin fip/
$ cp $FIPDIR/gxl/acs.bin fip/
$ cp $FIPDIR/gxl/bl21.bin fip/
$ cp $FIPDIR/gxl/bl30.bin fip/
$ cp $FIPDIR/gxl/bl301.bin fip/
$ cp $FIPDIR/gxl/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
```

(continues on next page)

(continued from previous page)

```
    bl30

$ $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0

$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl30_new.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl31.img
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl33.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl2sig --input fip/bl2_new.bin --output fip/bl2.n.bin.
↪sig
$ $FIPDIR/gxl/aml_encrypt_gxl --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

Note that Amlogic provides aml_encrypt_gxl as a 32-bit x86 binary with no source code. Should you prefer to avoid that, there are open source reverse engineered versions available:

1. gxlimg <https://github.com/repk/gxlimg>, which comes with a handy Makefile that automates the whole process.
2. meson-tools <https://github.com/afaerber/meson-tools>

However, these community-developed alternatives are not endorsed by or supported by Amlogic.

U-Boot for NanoPi-K2

NanoPi-K2 is a single board computer manufactured by FriendlyElec with the following specifications:

- Amlogic S905 ARM Cortex-A53 quad-core SoC @ 1.5GHz
- ARM Mali 450 GPU
- 2GB DDR3 SDRAM
- Gigabit Ethernet
- HDMI 2.0 4K/60Hz display
- 40-pin GPIO header
- 4 x USB 2.0 Host, 1 x USB OTG
- eMMC, microSD
- Infrared receiver

Schematics are available on the manufacturer website.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make nanopi-k2_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b libretech-cc amlogic-u-boot
$ git clone https://github.com/friendlyarm/u-boot.git -b nanopi-k2-v2015.01 amlogic-u-
↪boot
$ cd amlogic-u-boot
$ sed -i 's/aarch64-linux-gnu-/aarch64-none-elf-/' Makefile
$ sed -i 's/arm-linux-/arm-none-eabi-/' arch/arm/cpu/armv8/gxb/firmware/scp_task/
↪Makefile
$ make nanopi-k2_defconfig
$ make
$ export FIPDIR=$PWD/fip
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/gxb/bl2.bin fip/
$ cp $FIPDIR/gxb/acs.bin fip/
$ cp $FIPDIR/gxb/bl21.bin fip/
$ cp $FIPDIR/gxb/bl30.bin fip/
$ cp $FIPDIR/gxb/bl301.bin fip/
$ cp $FIPDIR/gxb/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ $FIPDIR/fip_create \
     --bl30 fip/bl30_new.bin \
```

(continues on next page)

(continued from previous page)

```
     --bl31 fip/bl31.img \
     --bl33 fip/bl33.bin \
     fip/fip.bin

$ python $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0

$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ cat fip/bl2_new.bin fip/fip.bin > fip/boot_new.bin

$ $FIPDIR/gxb/aml_encrypt_gxb --bootsig \
            --input fip/boot_new.bin
            --output fip/u-boot.bin
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin of=$DEV conv=fsync,notrunc bs=512 seek=1
```

U-Boot for ODROID-C2

ODROID-C2 is a single board computer manufactured by Hardkernel Co. Ltd with the following specifications:

- Amlogic S905 ARM Cortex-A53 quad-core SoC @ 2GHz
- ARM Mali 450 GPU
- 2GB DDR3 SDRAM
- Gigabit Ethernet
- HDMI 2.0 4K/60Hz display
- 40-pin GPIO header
- 4 x USB 2.0 Host, 1 x USB OTG
- eMMC, microSD
- Infrared receiver

Schematics are available on the manufacturer website.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make odroid-c2_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ DIR=odroid-c2
$ git clone --depth 1 \
   https://github.com/hardkernel/u-boot.git -b odroidc2-v2015.01 \
   $DIR
$ $DIR/fip/fip_create --bl30  $DIR/fip/gxb/bl30.bin \
                   --bl301 $DIR/fip/gxb/bl301.bin \
                   --bl31  $DIR/fip/gxb/bl31.bin \
                   --bl33  u-boot.bin \
                   $DIR/fip.bin
$ $DIR/fip/fip_create --dump $DIR/fip.bin
$ cat $DIR/fip/gxb/bl2.package $DIR/fip.bin > $DIR/boot_new.bin
$ $DIR/fip/gxb/aml_encrypt_gxb --bootsig \
                            --input $DIR/boot_new.bin \
                            --output $DIR/u-boot.img
$ dd if=$DIR/u-boot.img of=$DIR/u-boot.gxbb bs=512 skip=96
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ BL1=$DIR/sd_fuse/bl1.bin.hardkernel
$ dd if=$BL1 of=$DEV conv=fsync bs=1 count=442
$ dd if=$BL1 of=$DEV conv=fsync bs=512 skip=1 seek=1
$ dd if=$DIR/u-boot.gxbb of=$DEV conv=fsync bs=512 seek=97
```

U-Boot for ODROID-C4

ODROID-C4 is a single board computer manufactured by Hardkernel Co. Ltd with the following specifications:

- Amlogic S905X3 Arm Cortex-A55 quad-core SoC
- 4GB DDR4 SDRAM
- Gigabit Ethernet
- HDMI 2.1 display
- 40-pin GPIO header
- 4x USB 3.0 Host
- 1x USB 2.0 Host/OTG (micro)
- eMMC, microSD
- UART serial
- Infrared receiver

Schematics are available on the manufacturer website.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make odroid-c4_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH

$ DIR=odroid-c4
$ git clone --depth 1 \
   https://github.com/hardkernel/u-boot.git -b odroidg12-v2015.01 \
   $DIR

$ cd odroid-c4
$ make odroidc4_defconfig
$ make
$ export UBOOTDIR=$PWD
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ wget https://github.com/BayLibre/u-boot/releases/download/v2017.11-libretech-cc/blx_
↪fix_g12a.sh -O fip/blx_fix.sh
$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/hardkernel/odroidc4/firmware/acs.bin fip/
$ cp $UBOOTDIR/fip/g12a/bl2.bin fip/
$ cp $UBOOTDIR/fip/g12a/bl30.bin fip/
$ cp $UBOOTDIR/fip/g12a/bl31.img fip/
$ cp $UBOOTDIR/fip/g12a/ddr3_1d.fw fip/
$ cp $UBOOTDIR/fip/g12a/ddr4_1d.fw fip/
$ cp $UBOOTDIR/fip/g12a/ddr4_2d.fw fip/
$ cp $UBOOTDIR/fip/g12a/diag_lpddr4.fw fip/
$ cp $UBOOTDIR/fip/g12a/lpddr3_1d.fw fip/
$ cp $UBOOTDIR/fip/g12a/lpddr4_1d.fw fip/
$ cp $UBOOTDIR/fip/g12a/lpddr4_2d.fw fip/
$ cp $UBOOTDIR/fip/g12a/piei.fw fip/
$ cp $UBOOTDIR/fip/g12a/aml_ddr.fw fip/
$ cp u-boot.bin fip/bl33.bin

$ sh fip/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ sh fip/blx_fix.sh \
    fip/bl2.bin \
```

(continues on next page)

(continued from previous page)

```
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/acs.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl30sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.g12a.enc \
                                    --level v3
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl3sig --input fip/bl30_new.bin.g12a.enc \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33 --compress lz4
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $UBOOTDIR/fip/g12a/aml_encrypt_g12a --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc \
            --ddrfw1 fip/ddr4_1d.fw \
            --ddrfw2 fip/ddr4_2d.fw \
            --ddrfw3 fip/ddr3_1d.fw \
            --ddrfw4 fip/piei.fw \
            --ddrfw5 fip/lpddr4_1d.fw \
            --ddrfw6 fip/lpddr4_2d.fw \
            --ddrfw7 fip/diag_lpddr4.fw \
            --ddrfw8 fip/aml_ddr.fw \
            --ddrfw9 fip/lpddr3_1d.fw \
            --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for ODROID-N2

ODROID-N2 is a single board computer manufactured by Hardkernel Co. Ltd with the following specifications:

- Amlogic S922X ARM Cortex-A53 dual-core + Cortex-A73 quad-core SoC
- 4GB DDR4 SDRAM
- Gigabit Ethernet
- HDMI 2.1 4K/60Hz display
- 40-pin GPIO header

- 4 x USB 3.0 Host, 1 x USB OTG
- eMMC, microSD
- Infrared receiver

Schematics are available on the manufacturer website.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make odroid-n2_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
  $ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
  $ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
  $ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
  $ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
  $ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux/bin:$PATH

  $ DIR=odroid-n2
  $ git clone --depth 1 \
     https://github.com/hardkernel/u-boot.git -b odroidn2-v2015.01 \
     $DIR

  $ cd odroid-n2
  $ make odroidn2_defconfig
  $ make
  $ export UBOOTDIR=$PWD

Go back to mainline U-Boot source tree then :
```

```
$ mkdir fip

$ wget https://github.com/BayLibre/u-boot/releases/download/v2017.11-libretech-cc/blx_
↪fix_g12a.sh -O fip/blx_fix.sh
$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/hardkernel/odroidn2/firmware/acs.bin fip/
$ cp $UBOOTDIR/fip/g12b/bl2.bin fip/
$ cp $UBOOTDIR/fip/g12b/bl30.bin fip/
$ cp $UBOOTDIR/fip/g12b/bl31.img fip/
$ cp $UBOOTDIR/fip/g12b/ddr3_1d.fw fip/
$ cp $UBOOTDIR/fip/g12b/ddr4_1d.fw fip/
$ cp $UBOOTDIR/fip/g12b/ddr4_2d.fw fip/
$ cp $UBOOTDIR/fip/g12b/diag_lpddr4.fw fip/
$ cp $UBOOTDIR/fip/g12b/lpddr4_1d.fw fip/
$ cp $UBOOTDIR/fip/g12b/lpddr4_2d.fw fip/
$ cp $UBOOTDIR/fip/g12b/piei.fw fip/
```

(continues on next page)

(continued from previous page)

```
$ cp $UBOOTDIR/fip/g12b/aml_ddr.fw fip/
$ cp u-boot.bin fip/bl33.bin

$ sh fip/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ sh fip/blx_fix.sh \
    fip/bl2.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/acs.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl30sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.g12a.enc \
                                    --level v3
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl3sig --input fip/bl30_new.bin.g12a.enc \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33 --compress lz4
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $UBOOTDIR/fip/g12b/aml_encrypt_g12b --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc \
            --ddrfw1 fip/ddr4_1d.fw \
            --ddrfw2 fip/ddr4_2d.fw \
            --ddrfw3 fip/ddr3_1d.fw \
            --ddrfw4 fip/piei.fw \
            --ddrfw5 fip/lpddr4_1d.fw \
            --ddrfw6 fip/lpddr4_2d.fw \
            --ddrfw7 fip/diag_lpddr4.fw \
            --ddrfw8 fip/aml_ddr.fw \
            --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic P200

P200 is a reference board manufactured by Amlogic with the following specifications:

- Amlogic S905 ARM Cortex-A53 quad-core SoC @ 1.5GHz
- ARM Mali 450 GPU
- 2GB DDR3 SDRAM
- Gigabit Ethernet
- HDMI 2.0 4K/60Hz display
- 2 x USB 2.0 Host
- eMMC, microSD
- Infrared receiver
- SDIO WiFi Module
- CVBS+Stereo Audio Jack

Schematics are available from Amlogic on demand.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make p200_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b n-amlogic-openlinux-20170606␣
↪amlogic-u-boot
$ cd amlogic-u-boot
$ make gxb_p200_v1_defconfig
$ make
$ export FIPDIR=$PWD/fip
```

Go back to mainline U-boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/gxl/bl2.bin fip/
$ cp $FIPDIR/gxl/acs.bin fip/
$ cp $FIPDIR/gxl/bl21.bin fip/
$ cp $FIPDIR/gxl/bl30.bin fip/
```

(continues on next page)

(continued from previous page)

```
$ cp $FIPDIR/gxl/bl301.bin fip/
$ cp $FIPDIR/gxl/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0

$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl30_new.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl31.img
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl33.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl2sig --input fip/bl2_new.bin --output fip/bl2.n.bin.
↪sig
$ $FIPDIR/gxl/aml_encrypt_gxl --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic P201

P201 is a reference board manufactured by Amlogic with the following specifications:

- Amlogic S905 ARM Cortex-A53 quad-core SoC @ 1.5GHz
- ARM Mali 450 GPU
- 2GB DDR3 SDRAM
- 10/100 Ethernet
- HDMI 2.0 4K/60Hz display
- 2 x USB 2.0 Host
- eMMC, microSD

- Infrared receiver
- SDIO WiFi Module
- CVBS+Stereo Audio Jack

Schematics are available from Amlogic on demand.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make p201_defconfig
$ make
```

Image creation

Amlogic doesn’t provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b n-amlogic-openlinux-20170606␣
↪amlogic-u-boot
$ cd amlogic-u-boot
$ make gxb_p201_v1_defconfig
$ make
$ export FIPDIR=$PWD/fip
```

Go back to mainline U-boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/gxl/bl2.bin fip/
$ cp $FIPDIR/gxl/acs.bin fip/
$ cp $FIPDIR/gxl/bl21.bin fip/
$ cp $FIPDIR/gxl/bl30.bin fip/
$ cp $FIPDIR/gxl/bl301.bin fip/
$ cp $FIPDIR/gxl/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0
```

(continues on next page)

(continued from previous page)

```
$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl30_new.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl31.img
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl33.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl2sig --input fip/bl2_new.bin --output fip/bl2.n.bin.
↪sig
$ $FIPDIR/gxl/aml_encrypt_gxl --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic P212

P212 is a reference board manufactured by Amlogic with the following specifications:

- Amlogic S905X ARM Cortex-A53 quad-core SoC @ 1.5GHz
- ARM Mali 450 GPU
- 2GB DDR3 SDRAM
- 10/100 Ethernet
- HDMI 2.0 4K/60Hz display
- 2 x USB 2.0 Host
- eMMC, microSD
- Infrared receiver
- SDIO WiFi Module
- CVBS+Stereo Audio Jack

Schematics are available from Amlogic on demand.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make p212_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b n-amlogic-openlinux-20170606␣
↪amlogic-u-boot
$ cd amlogic-u-boot
$ make gxl_p212_v1_defconfig
$ make
$ export FIPDIR=$PWD/fip
```

Go back to mainline U-boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/gxl/bl2.bin fip/
$ cp $FIPDIR/gxl/acs.bin fip/
$ cp $FIPDIR/gxl/bl21.bin fip/
$ cp $FIPDIR/gxl/bl30.bin fip/
$ cp $FIPDIR/gxl/bl301.bin fip/
$ cp $FIPDIR/gxl/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0

$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl30_new.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl31.img
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl33.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl2sig --input fip/bl2_new.bin --output fip/bl2.n.bin.
↪sig
```

(continues on next page)

(continued from previous page)

```
$ $FIPDIR/gxl/aml_encrypt_gxl --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic Q200

Q200 is a reference board manufactured by Amlogic with the following specifications:

- Amlogic S912 ARM Cortex-A53 octo-core SoC @ 1.5GHz
- ARM Mali T860 GPU
- 2/3GB DDR4 SDRAM
- 10/100/1000 Ethernet
- HDMI 2.0 4K/60Hz display
- 2 x USB 2.0 Host, 1 x USB 2.0 Device
- 16GB/32GB/64GB eMMC
- 2MB SPI Flash
- microSD
- SDIO Wifi Module, Bluetooth
- IR receiver

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make khadas-vim2_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
```

(continues on next page)

(continued from previous page)

```
$ git clone https://github.com/BayLibre/u-boot.git -b n-amlogic-openlinux-20170606␣
↪amlogic-u-boot
$ cd amlogic-u-boot
$ make gxm_q200_v1_defconfig
$ make
$ export FIPDIR=$PWD/fip
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/gxl/bl2.bin fip/
$ cp $FIPDIR/gxl/acs.bin fip/
$ cp $FIPDIR/gxl/bl21.bin fip/
$ cp $FIPDIR/gxl/bl30.bin fip/
$ cp $FIPDIR/gxl/bl301.bin fip/
$ cp $FIPDIR/gxl/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ python $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0

$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl30_new.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl31.img
$ $FIPDIR/gxl/aml_encrypt_gxl --bl3enc --input fip/bl33.bin
$ $FIPDIR/gxl/aml_encrypt_gxl --bl2sig --input fip/bl2_new.bin --output fip/bl2.n.bin.
↪sig
$ $FIPDIR/gxl/aml_encrypt_gxl --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic S400

S400 is a reference board manufactured by Amlogic with the following specifications:

- Amlogic A113DX ARM Cortex-A53 quad-core SoC @ 1.2GHz
- 1GB DDR4 SDRAM
- 10/100 Ethernet
- 2 x USB 2.0 Host
- eMMC
- Infrared receiver
- SDIO WiFi Module
- MIPI DSI Connector
- Audio HAT Connector
- PCI-E M.2 Connectors

Schematics are available from Amlogic on demand.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make s400_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b n-amlogic-openlinux-20170606␣
↪amlogic-u-boot
$ cd amlogic-u-boot
$ make axg_s400_v1_defconfig
$ make
$ export FIPDIR=$PWD/fip
```

Go back to mainline U-boot source tree then :

```
$ mkdir fip

$ cp $FIPDIR/axg/bl2.bin fip/
$ cp $FIPDIR/axg/acs.bin fip/
$ cp $FIPDIR/axg/bl21.bin fip/
$ cp $FIPDIR/axg/bl30.bin fip/
```

(continues on next page)

(continued from previous page)

```
$ cp $FIPDIR/axg/bl301.bin fip/
$ cp $FIPDIR/axg/bl31.img fip/
$ cp u-boot.bin fip/bl33.bin

$ $FIPDIR/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ $FIPDIR/acs_tool.pyc fip/bl2.bin fip/bl2_acs.bin fip/acs.bin 0

$ $FIPDIR/blx_fix.sh \
    fip/bl2_acs.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/bl21.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/axg/aml_encrypt_axg --bl3sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $FIPDIR/axg/aml_encrypt_axg --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $FIPDIR/axg/aml_encrypt_axg --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33
$ $FIPDIR/axg/aml_encrypt_axg --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $FIPDIR/axg/aml_encrypt_axg --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic SEI510

SEI510 is a customer board manufactured by SEI Robotics with the following specifications:

- Amlogic S905X2 ARM Cortex-A53 quad-core SoC
- 2GB DDR4 SDRAM
- 10/100 Ethernet (Internal PHY)

- 1 x USB 3.0 Host
- eMMC
- SDcard
- Infrared receiver
- SDIO WiFi Module

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make sei510_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b buildroot-openlinux-20180418␣
↪amlogic-u-boot
$ cd amlogic-u-boot
$ make g12a_u200_v1_defconfig
$ make
$ export UBOOTDIR=$PWD
```

Download the latest Amlogic Buildroot package, and extract it :

```
$ wget http://openlinux2.amlogic.com:8000/ARM/filesystem/Linux_BSP/buildroot_openlinux_
↪kernel_4.9_fbdev_20180706.tar.gz
$ tar xfz buildroot_openlinux_kernel_4.9_fbdev_20180706.tar.gz buildroot_openlinux_
↪kernel_4.9_fbdev_20180706/bootloader
$ export BRDIR=$PWD/buildroot_openlinux_kernel_4.9_fbdev_20180706
$ export FIPDIR=$BRDIR/bootloader/uboot-repo/fip
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ wget https://github.com/BayLibre/u-boot/releases/download/v2017.11-libretech-cc/blx_
↪fix_g12a.sh -O fip/blx_fix.sh
$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/amlogic/g12a_u200_v1/firmware/acs.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl2/bin/g12a/bl2.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl30/bin/g12a/bl30.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl31_1.3/bin/g12a/bl31.img fip/
$ cp $FIPDIR/g12a/ddr3_1d.fw fip/
```

(continues on next page)

(continued from previous page)

```
$ cp $FIPDIR/g12a/ddr4_1d.fw fip/
$ cp $FIPDIR/g12a/ddr4_2d.fw fip/
$ cp $FIPDIR/g12a/diag_lpddr4.fw fip/
$ cp $FIPDIR/g12a/lpddr4_1d.fw fip/
$ cp $FIPDIR/g12a/lpddr4_2d.fw fip/
$ cp $FIPDIR/g12a/piei.fw fip/
$ cp u-boot.bin fip/bl33.bin

$ sh fip/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ sh fip/blx_fix.sh \
    fip/bl2.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/acs.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/g12a/aml_encrypt_g12a --bl30sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.g12a.enc \
                                    --level v3
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl30_new.bin.g12a.enc \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33
$ $FIPDIR/g12a/aml_encrypt_g12a --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $FIPDIR/g12a/aml_encrypt_g12a --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc \
            --ddrfw1 fip/ddr4_1d.fw \
            --ddrfw2 fip/ddr4_2d.fw \
            --ddrfw3 fip/ddr3_1d.fw \
            --ddrfw4 fip/piei.fw \
            --ddrfw5 fip/lpddr4_1d.fw \
            --ddrfw6 fip/lpddr4_2d.fw \
            --ddrfw7 fip/diag_lpddr4.fw \
            --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic SEI610

SEI610 is a customer board manufactured by SEI Robotics with the following specifications:

- Amlogic S905X3 ARM Cortex-A55 quad-core SoC
- 2GB DDR4 SDRAM
- 10/100 Ethernet (Internal PHY)
- 1 x USB 3.0 Host
- 1 x USB Type-C DRD
- 1 x FTDI USB Serial Debug Interface
- eMMC
- SDcard
- Infrared receiver
- SDIO WiFi Module

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make sei610_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b buildroot-openlinux-4.9-g12a-
↪201904 amlogic-u-boot
$ cd amlogic-u-boot
$ make sm1_ac200_v1_defconfig
$ make
$ export UBOOTDIR=$PWD
```

Download the latest Amlogic Buildroot package, and extract it :

```
$ wget http://openlinux2.amlogic.com:8000/ARM/filesystem/buildroot-openlinux-A113-
↪201901.tgz
$ tar xfz buildroot-openlinux-A113-201901.tgz buildroot-openlinux-A113-201901/
↪bootloader
$ export BRDIR=$PWD/buildroot-openlinux-A113-201901
$ export FIPDIR=$BRDIR/bootloader/uboot-repo/fip
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ wget https://github.com/BayLibre/u-boot/releases/download/v2017.11-libretech-cc/blx_
↪fix_g12a.sh -O fip/blx_fix.sh
$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/amlogic/g12a_u200_v1/firmware/acs.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl2/bin/g12a/bl2.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl30/bin/g12a/bl30.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl31_1.3/bin/g12a/bl31.img fip/
$ cp $FIPDIR/g12a/ddr3_1d.fw fip/
$ cp $FIPDIR/g12a/ddr4_1d.fw fip/
$ cp $FIPDIR/g12a/ddr4_2d.fw fip/
$ cp $FIPDIR/g12a/diag_lpddr4.fw fip/
$ cp $FIPDIR/g12a/lpddr4_1d.fw fip/
$ cp $FIPDIR/g12a/lpddr4_2d.fw fip/
$ cp $FIPDIR/g12a/piei.fw fip/
$ cp u-boot.bin fip/bl33.bin

$ sh fip/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ sh fip/blx_fix.sh \
    fip/bl2.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/acs.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/g12a/aml_encrypt_g12a --bl30sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.g12a.enc \
                                    --level v3
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl30_new.bin.g12a.enc \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33
```

(continues on next page)

(continued from previous page)

```
$ $FIPDIR/g12a/aml_encrypt_g12a --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $FIPDIR/g12a/aml_encrypt_g12a --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc \
            --ddrfw1 fip/ddr4_1d.fw \
            --ddrfw2 fip/ddr4_2d.fw \
            --ddrfw3 fip/ddr3_1d.fw \
            --ddrfw4 fip/piei.fw \
            --ddrfw5 fip/lpddr4_1d.fw \
            --ddrfw6 fip/lpddr4_2d.fw \
            --ddrfw7 fip/diag_lpddr4.fw \
            --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic U200

U200 is a reference board manufactured by Amlogic with the following specifications:

- Amlogic S905D2 ARM Cortex-A53 quad-core SoC
- 2GB DDR4 SDRAM
- 10/100 Ethernet (Internal PHY)
- 1 x USB 3.0 Host
- eMMC
- SDcard
- Infrared receiver
- SDIO WiFi Module
- MIPI DSI Connector
- Audio HAT Connector
- PCI-E M.2 Connector

Schematics are available from Amlogic on demand.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make u200_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b buildroot-openlinux-20180418␣
↪amlogic-u-boot
$ cd amlogic-u-boot
$ make g12a_u200_v1_defconfig
$ make
$ export UBOOTDIR=$PWD
```

Download the latest Amlogic Buildroot package, and extract it :

```
$ wget http://openlinux2.amlogic.com:8000/ARM/filesystem/Linux_BSP/buildroot_openlinux_
↪kernel_4.9_fbdev_20180706.tar.gz
$ tar xfz buildroot_openlinux_kernel_4.9_fbdev_20180706.tar.gz buildroot_openlinux_
↪kernel_4.9_fbdev_20180706/bootloader
$ export BRDIR=$PWD/buildroot_openlinux_kernel_4.9_fbdev_20180706
$ export FIPDIR=$BRDIR/bootloader/uboot-repo/fip
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ wget https://github.com/BayLibre/u-boot/releases/download/v2017.11-libretech-cc/blx_
↪fix_g12a.sh -O fip/blx_fix.sh
$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/amlogic/g12a_u200_v1/firmware/acs.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl2/bin/g12a/bl2.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl30/bin/g12a/bl30.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl31_1.3/bin/g12a/bl31.img fip/
$ cp $FIPDIR/g12a/ddr3_1d.fw fip/
$ cp $FIPDIR/g12a/ddr4_1d.fw fip/
$ cp $FIPDIR/g12a/ddr4_2d.fw fip/
$ cp $FIPDIR/g12a/diag_lpddr4.fw fip/
$ cp $FIPDIR/g12a/lpddr4_1d.fw fip/
$ cp $FIPDIR/g12a/lpddr4_2d.fw fip/
$ cp $FIPDIR/g12a/piei.fw fip/
$ cp u-boot.bin fip/bl33.bin

$ sh fip/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30
```

(continues on next page)

(continued from previous page)

```
$ sh fip/blx_fix.sh \
    fip/bl2.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/acs.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/g12a/aml_encrypt_g12a --bl30sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.g12a.enc \
                                    --level v3
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl30_new.bin.g12a.enc \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $FIPDIR/g12a/aml_encrypt_g12a --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33
$ $FIPDIR/g12a/aml_encrypt_g12a --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $FIPDIR/g12a/aml_encrypt_g12a --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc \
            --ddrfw1 fip/ddr4_1d.fw \
            --ddrfw2 fip/ddr4_2d.fw \
            --ddrfw3 fip/ddr3_1d.fw \
            --ddrfw4 fip/piei.fw \
            --ddrfw5 fip/lpddr4_1d.fw \
            --ddrfw6 fip/lpddr4_2d.fw \
            --ddrfw7 fip/diag_lpddr4.fw \
            --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

U-Boot for Amlogic W400

U200 is a reference board manufactured by Amlogic with the following specifications:

- Amlogic S922X ARM Cortex-A53 dual-core + Cortex-A73 quad-core SoC
- 2GB DDR4 SDRAM
- 10/100 Ethernet (Internal PHY)
- 1 x USB 3.0 Host
- eMMC
- SDcard

- Infrared receiver
- SDIO WiFi Module
- MIPI DSI Connector
- Audio HAT Connector
- PCI-E M.2 Connector

Schematics are available from Amlogic on demand.

U-Boot compilation

```
$ export CROSS_COMPILE=aarch64-none-elf-
$ make w400_defconfig
$ make
```

Image creation

Amlogic doesn't provide sources for the firmware and for tools needed to create the bootloader image, so it is necessary to obtain them from the git tree published by the board vendor:

```
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ wget https://releases.linaro.org/archive/13.11/components/toolchain/binaries/gcc-
↪linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-aarch64-none-elf-4.8-2013.11_linux.tar.xz
$ tar xvfJ gcc-linaro-arm-none-eabi-4.8-2013.11_linux.tar.xz
$ export PATH=$PWD/gcc-linaro-aarch64-none-elf-4.8-2013.11_linux/bin:$PWD/gcc-linaro-
↪arm-none-eabi-4.8-2013.11_linux/bin:$PATH
$ git clone https://github.com/BayLibre/u-boot.git -b buildroot-openlinux-20180418␣
↪amlogic-u-boot
$ cd amlogic-u-boot
$ make g12b_w400_v1_defconfig
$ make
$ export UBOOTDIR=$PWD
```

Download the latest Amlogic Buildroot package, and extract it :

```
$ wget http://openlinux2.amlogic.com:8000/ARM/filesystem/Linux_BSP/buildroot_openlinux_
↪kernel_4.9_fbdev_20180706.tar.gz
$ tar xfz buildroot_openlinux_kernel_4.9_fbdev_20180706.tar.gz buildroot_openlinux_
↪kernel_4.9_fbdev_20180706/bootloader
$ export BRDIR=$PWD/buildroot_openlinux_kernel_4.9_fbdev_20180706
$ export FIPDIR=$BRDIR/bootloader/uboot-repo/fip
```

Go back to mainline U-Boot source tree then :

```
$ mkdir fip

$ wget https://github.com/BayLibre/u-boot/releases/download/v2017.11-libretech-cc/blx_
↪fix_g12a.sh -O fip/blx_fix.sh
$ cp $UBOOTDIR/build/scp_task/bl301.bin fip/
$ cp $UBOOTDIR/build/board/amlogic/g12b_w400_v1/firmware/acs.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl2/bin/g12b/bl2.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl30/bin/g12b/bl30.bin fip/
$ cp $BRDIR/bootloader/uboot-repo/bl31_1.3/bin/g12b/bl31.img fip/
```

(continues on next page)

(continued from previous page)

```
$ cp $FIPDIR/g12b/ddr3_1d.fw fip/
$ cp $FIPDIR/g12b/ddr4_1d.fw fip/
$ cp $FIPDIR/g12b/ddr4_2d.fw fip/
$ cp $FIPDIR/g12b/diag_lpddr4.fw fip/
$ cp $FIPDIR/g12b/lpddr4_1d.fw fip/
$ cp $FIPDIR/g12b/lpddr4_2d.fw fip/
$ cp $FIPDIR/g12b/piei.fw fip/
$ cp $FIPDIR/g12b/aml_ddr.fw fip/
$ cp u-boot.bin fip/bl33.bin

$ sh fip/blx_fix.sh \
    fip/bl30.bin \
    fip/zero_tmp \
    fip/bl30_zero.bin \
    fip/bl301.bin \
    fip/bl301_zero.bin \
    fip/bl30_new.bin \
    bl30

$ sh fip/blx_fix.sh \
    fip/bl2.bin \
    fip/zero_tmp \
    fip/bl2_zero.bin \
    fip/acs.bin \
    fip/bl21_zero.bin \
    fip/bl2_new.bin \
    bl2

$ $FIPDIR/g12b/aml_encrypt_g12b --bl30sig --input fip/bl30_new.bin \
                                    --output fip/bl30_new.bin.g12a.enc \
                                    --level v3
$ $FIPDIR/g12b/aml_encrypt_g12b --bl3sig --input fip/bl30_new.bin.g12a.enc \
                                    --output fip/bl30_new.bin.enc \
                                    --level v3 --type bl30
$ $FIPDIR/g12b/aml_encrypt_g12b --bl3sig --input fip/bl31.img \
                                    --output fip/bl31.img.enc \
                                    --level v3 --type bl31
$ $FIPDIR/g12b/aml_encrypt_g12b --bl3sig --input fip/bl33.bin --compress lz4 \
                                    --output fip/bl33.bin.enc \
                                    --level v3 --type bl33
$ $FIPDIR/g12b/aml_encrypt_g12b --bl2sig --input fip/bl2_new.bin \
                                    --output fip/bl2.n.bin.sig
$ $FIPDIR/g12b/aml_encrypt_g12b --bootmk \
            --output fip/u-boot.bin \
            --bl2 fip/bl2.n.bin.sig \
            --bl30 fip/bl30_new.bin.enc \
            --bl31 fip/bl31.img.enc \
            --bl33 fip/bl33.bin.enc \
            --ddrfw1 fip/ddr4_1d.fw \
            --ddrfw2 fip/ddr4_2d.fw \
            --ddrfw3 fip/ddr3_1d.fw \
            --ddrfw4 fip/piei.fw \
            --ddrfw5 fip/lpddr4_1d.fw \
            --ddrfw6 fip/lpddr4_2d.fw \
            --ddrfw7 fip/diag_lpddr4.fw \
            --ddrfw8 fip/aml_ddr.fw \
```

(continues on next page)

(continued from previous page)

```
            --level v3
```

and then write the image to SD with:

```
$ DEV=/dev/your_sd_device
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=512 skip=1 seek=1
$ dd if=fip/u-boot.bin.sd.bin of=$DEV conv=fsync,notrunc bs=1 count=444
```

7.1.4 Atmel

AT91 Evaluation kits

Board mapping & boot media

AT91SAM9260EK, AT91SAM9G20EK & AT91SAM9XEEK

Memory map:

```
0x20000000 - 23FFFFFF   SDRAM (64 MB)
0xC0000000 - Cxxxxxxx   Atmel Dataflash card (J13)
0xD0000000 - D07FFFFF   Soldered Atmel Dataflash (AT45DB642)
```

Environment variables

U-Boot environment variables can be stored at different places:

- Dataflash on SPI chip select 1 (default)
- Dataflash on SPI chip select 0 (dataflash card)
- Nand flash

You can choose your storage location at config step (here for at91sam9260ek):

```
make at91sam9260ek_nandflash_config     - use nand flash
make at91sam9260ek_dataflash_cs0_config - use data flash (spi cs0)
make at91sam9260ek_dataflash_cs1_config - use data flash (spi cs1)
```

AT91SAM9261EK, AT91SAM9G10EK

Memory map:

```
0x20000000 - 23FFFFFF   SDRAM (64 MB)
0xC0000000 - C07FFFFF   Soldered Atmel Dataflash (AT45DB642)
0xD0000000 - Dxxxxxxx   Atmel Dataflash card (J22)
```

Environment variables

U-Boot environment variables can be stored at different places:

- Dataflash on SPI chip select 0 (default)
- Dataflash on SPI chip select 3 (dataflash card)
- Nand flash

You can choose your storage location at config step (here for at91sam9260ek):

```
make at91sam9261ek_nandflash_config     - use nand flash
make at91sam9261ek_dataflash_cs0_config - use data flash (spi cs0)
make at91sam9261ek_dataflash_cs3_config - use data flash (spi cs3)
```

AT91SAM9263EK

Memory map:

```
0x20000000 - 23FFFFFF   SDRAM (64 MB)
0xC0000000 - Cxxxxxxx   Atmel Dataflash card (J9)
```

Environment variables

U-Boot environment variables can be stored at different places:

- Dataflash on SPI chip select 0 (dataflash card)
- Nand flash
- Nor flash (not populate by default)

You can choose your storage location at config step (here for at91sam9260ek):

```
make at91sam9263ek_nandflash_config     - use nand flash
make at91sam9263ek_dataflash_cs0_config - use data flash (spi cs0)
make at91sam9263ek_norflash_config      - use nor flash
```

You can choose to boot directly from U-Boot at config step:

```
make at91sam9263ek_norflash_boot_config - boot from nor flash
```

AT91SAM9M10G45EK

Memory map:

```
0x70000000 - 77FFFFFF   SDRAM (128 MB)
```

Environment variables

U-Boot environment variables can be stored at different places:

- Nand flash

You can choose your storage location at config step (here for at91sam9m10g45ek):

```
make at91sam9m10g45ek_nandflash_config          - use nand flash
```

AT91SAM9RLEK

Memory map:

```
0x20000000 - 23FFFFFF   SDRAM (64 MB)
0xC0000000 - C07FFFFF   Soldered Atmel Dataflash (AT45DB642)
```

Environment variables

U-Boot environment variables can be stored at different places:

- Dataflash on SPI chip select 0

- Nand flash.

You can choose your storage location at config step (here for at91sam9rlek):

```
make at91sam9rlek_nandflash_config      - use nand flash
```

AT91SAM9N12EK, AT91SAM9X5EK

Memory map:

```
0x20000000 - 27FFFFFF   SDRAM (128 MB)
```

Environment variables

U-Boot environment variables can be stored at different places:

- Nand flash
- SD/MMC card
- Serialflash/Dataflash on SPI chip select 0

You can choose your storage location at config step (here for at91sam9x5ek):

```
make at91sam9x5ek_dataflash_config      - use data flash
make at91sam9x5ek_mmc_config            - use sd/mmc card
make at91sam9x5ek_nandflash_config      - use nand flash
make at91sam9x5ek_spiflash_config       - use serial flash
```

SAMA5D3XEK

Memory map:

```
0x20000000 - 3FFFFFFF   SDRAM (512 MB)
```

Environment variables

U-Boot environment variables can be stored at different places:

- Nand flash
- SD/MMC card
- Serialflash on SPI chip select 0

You can choose your storage location at config step (here for sama5d3xek):

```
make sama5d3xek_mmc_config              - use SD/MMC card
make sama5d3xek_nandflash_config        - use nand flash
make sama5d3xek_serialflash_config      - use serial flash
```

NAND partition table

All the board support boot from NAND flash will use the following NAND partition table:

```
0x00000000 - 0x0003FFFF bootstrap       (256 KiB)
0x00040000 - 0x000BFFFF u-boot          (512 KiB)
0x000C0000 - 0x000FFFFF env             (256 KiB)
0x00100000 - 0x0013FFFF env_redundant   (256 KiB)
```

(continues on next page)

(continued from previous page)

```
0x00140000 - 0x0017FFFF spare          (256 KiB)
0x00180000 - 0x001FFFFF dtb            (512 KiB)
0x00200000 - 0x007FFFFF kernel         (6 MiB)
0x00800000 - 0xxxxxxxxx rootfs         (All left)
```

Watchdog support

For security reasons, the at91 watchdog is running at boot time and, if deactivated, cannot be used anymore. If you want to use the watchdog, you will need to keep it running in your code (make sure not to disable it in AT91Bootstrap for instance).

In the U-Boot configuration, the AT91 watchdog support is enabled using the CONFIG_WDT and CONFIG_WDT_AT91 options.

7.1.5 Coreboot

Coreboot

Build Instructions for U-Boot as coreboot payload

Building U-Boot as a coreboot payload is just like building U-Boot for targets on other architectures, like below:

```
$ make coreboot_defconfig
$ make all
```

Test with coreboot

For testing U-Boot as the coreboot payload, there are things that need be paid attention to. coreboot supports loading an ELF executable and a 32-bit plain binary, as well as other supported payloads. With the default configuration, U-Boot is set up to use a separate Device Tree Blob (dtb). As of today, the generated u-boot-dtb.bin needs to be packaged by the cbfstool utility (a tool provided by coreboot) manually as coreboot's 'make menuconfig' does not provide this capability yet. The command is as follows:

```
# in the coreboot root directory
$ ./build/util/cbfstool/cbfstool build/coreboot.rom add-flat-binary \
  -f u-boot-dtb.bin -n fallback/payload -c lzma -l 0x1110000 -e 0x1110000
```

Make sure 0x1110000 matches CONFIG_SYS_TEXT_BASE, which is the symbol address of _x86boot_start (in arch/x86/cpu/start.S).

If you want to use ELF as the coreboot payload, change U-Boot configuration to use CONFIG_OF_EMBED instead of CONFIG_OF_SEPARATE.

To enable video you must enable these options in coreboot:

- Set framebuffer graphics resolution (1280x1024 32k-color (1:5:5))
- Keep VESA framebuffer

At present it seems that for Minnowboard Max, coreboot does not pass through the video information correctly (it always says the resolution is 0x0). This works correctly for link though.

64-bit U-Boot

In addition to the 32-bit 'coreboot' build there is a 'coreboot64' build. This produces an image which can be booted from coreboot (32-bit). Internally it works by using a 32-bit SPL binary to switch to 64-bit for running U-Boot. It can be useful for running UEFI applications, for example.

This has only been lightly tested.

7.1.6 Emulation

QEMU ARM

QEMU for ARM supports a special 'virt' machine designed for emulation and virtualization purposes. This document describes how to run U-Boot under it. Both 32-bit ARM and AArch64 are supported.

The 'virt' platform provides the following as the basic functionality:

- A freely configurable amount of CPU cores
- U-Boot loaded and executing in the emulated flash at address 0x0
- A generated device tree blob placed at the start of RAM
- A freely configurable amount of RAM, described by the DTB
- A PL011 serial port, discoverable via the DTB
- An ARMv7/ARMv8 architected timer
- PSCI for rebooting the system
- A generic ECAM-based PCI host controller, discoverable via the DTB

Additionally, a number of optional peripherals can be added to the PCI bus.

Building U-Boot

Set the CROSS_COMPILE environment variable as usual, and run:

- For ARM:

```
make qemu_arm_defconfig
make
```

- For AArch64:

```
make qemu_arm64_defconfig
make
```

Running U-Boot

The minimal QEMU command line to get U-Boot up and running is:

- For ARM:

```
qemu-system-arm -machine virt -bios u-boot.bin
```

- For AArch64:

```
qemu-system-aarch64 -machine virt -cpu cortex-a57 -bios u-boot.bin
```

Note that for some odd reason qemu-system-aarch64 needs to be explicitly told to use a 64-bit CPU or it will boot in 32-bit mode.

Additional persistent U-boot environment support can be added as follows:

- Create envstore.img using qemu-img:

```
qemu-img create -f raw envstore.img 64M
```

- Add a pflash drive parameter to the command line:

```
-drive if=pflash,format=raw,index=1,file=envstore.img
```

Additional peripherals that have been tested to work in both U-Boot and Linux can be enabled with the following command line parameters:

- To add a Serial ATA disk via an Intel ICH9 AHCI controller, pass e.g.:

```
-drive if=none,file=disk.img,id=mydisk -device ich9-ahci,id=ahci -device ide-drive,
↪drive=mydisk,bus=ahci.0
```

- To add an Intel E1000 network adapter, pass e.g.:

```
-netdev user,id=net0 -device e1000,netdev=net0
```

- To add an EHCI-compliant USB host controller, pass e.g.:

```
-device usb-ehci,id=ehci
```

- To add a NVMe disk, pass e.g.:

```
-drive if=none,file=disk.img,id=mydisk -device nvme,drive=mydisk,serial=foo
```

These have been tested in QEMU 2.9.0 but should work in at least 2.5.0 as well.

Debug UART

The debug UART on the ARM virt board uses these settings:

```
CONFIG_DEBUG_UART=y
CONFIG_DEBUG_UART_PL010=y
CONFIG_DEBUG_UART_BASE=0x9000000
CONFIG_DEBUG_UART_CLOCK=0
```

QEMU MIPS

Qemu is a full system emulator. See http://www.nongnu.org/qemu/

Limitations & comments

Supports the "-M mips" configuration of qemu: serial,NE2000,IDE. Supports little and big endian as well as 32 bit and 64 bit. Derived from au1x00 with a lot of things cut out.

Supports emulated flash (patch Jean-Christophe PLAGNIOL-VILLARD) with recent qemu versions. When using emulated flash, launch with -pflash <filename> and erase mips_bios.bin.

Notes for the Qemu MIPS port

Example usage

Using u-boot.bin as ROM (replaces Qemu monitor):

32 bit, big endian

```
make qemu_mips_defconfig
qemu-system-mips -M mips -bios u-boot.bin -nographic
```

32 bit, little endian

```
make qemu_mipsel_defconfig
qemu-system-mipsel -M mips -bios u-boot.bin -nographic
```

64 bit, big endian

```
make qemu_mips64_defconfig
qemu-system-mips64 -cpu MIPS64R2-generic -M mips -bios u-boot.bin -nographic
```

64 bit, little endian

```
make qemu_mips64el_defconfig
qemu-system-mips64el -cpu MIPS64R2-generic -M mips -bios u-boot.bin -nographic
```

or using u-boot.bin from emulated flash:

if you use a QEMU version after commit 4224

```
# create image:
dd of=flash bs=1k count=4k if=/dev/zero
dd of=flash bs=1k conv=notrunc if=u-boot.bin
# start it (see above):
qemu-system-mips[64][el] [-cpu MIPS64R2-generic] -M mips -pflash flash -nographic
```

Download kernel + initrd

On ftp://ftp.denx.de/pub/contrib/Jean-Christophe_Plagniol-Villard/qemu_mips/ you can downland:

```
#config to build the kernel
qemu_mips_defconfig
#patch to fix mips interrupt init on 2.6.24.y kernel
qemu_mips_kernel.patch
initrd.gz
vmlinux
vmlinux.bin
System.map
```

Generate uImage

```
tools/mkimage -A mips -O linux -T kernel -C gzip -a 0x80010000 -e 0x80245650 -n "Linux 2.6.24.y" -d vmlinux.bin.gz uImage
```

Copy uImage to Flash

```
dd if=uImage bs=1k conv=notrunc seek=224 of=flash
```

Generate Ide Disk

```
dd of=ide bs=1k count=100k if=/dev/zero

# Create partion table
sudo sfdisk ide << EOF
label: dos
label-id: 0x6fe3a999
device: image
unit: sectors
image1 : start=       63, size=    32067, Id=83
image2 : start=    32130, size=    32130, Id=83
image3 : start=    64260, size=  4128705, Id=83
EOF
```

Copy to ide

```
dd if=uImage bs=512 conv=notrunc seek=63 of=ide
```

Generate ext2 on part 2 on Copy uImage and initrd.gz

```
# Attached as loop device ide offset = 32130 * 512
sudo losetup -o 16450560 /dev/loop0 ide
# Format as ext2 ( arg2 : nb blocks)
sudo mkfs.ext2 /dev/loop0 16065
sudo losetup -d /dev/loop0
# Mount and copy uImage and initrd.gz to it
sudo mount -o loop,offset=16450560 -t ext2 ide /mnt
sudo mkdir /mnt/boot
cp {initrd.gz,uImage} /mnt/boot/
# Umount it
sudo umount /mnt
```

Set Environment

```
setenv rd_start 0x80800000
setenv rd_size 2663940
setenv kernel BFC38000
setenv oad_addr 80500000
setenv load_addr2 80F00000
setenv kernel_flash BFC38000
setenv load_addr_hello 80200000
setenv bootargs 'root=/dev/ram0 init=/bin/sh'
setenv load_rd_ext2 'ide res; ext2load ide 0:2 ${rd_start} /boot/initrd.gz'
setenv load_rd_tftp 'tftp ${rd_start} /initrd.gz'
```

(continues on next page)

(continued from previous page)

```
setenv load_kernel_hda 'ide res; diskboot ${load_addr} 0:2'
setenv load_kernel_ext2 'ide res; ext2load ide 0:2 ${load_addr} /boot/uImage'
setenv load_kernel_tftp 'tftp ${load_addr} /qemu_mips/uImage'
setenv boot_ext2_ext2 'run load_rd_ext2; run load_kernel_ext2; run addmisc; bootm $
↪{load_addr}'
setenv boot_ext2_flash 'run load_rd_ext2; run addmisc; bootm ${kernel_flash}'
setenv boot_ext2_hda 'run load_rd_ext2; run load_kernel_hda; run addmisc; bootm ${load_
↪addr}'
setenv boot_ext2_tftp 'run load_rd_ext2; run load_kernel_tftp; run addmisc; bootm $
↪{load_addr}'
setenv boot_tftp_hda 'run load_rd_tftp; run load_kernel_hda; run addmisc; bootm ${load_
↪addr}'
setenv boot_tftp_ext2 'run load_rd_tftp; run load_kernel_ext2; run addmisc; bootm $
↪{load_addr}'
setenv boot_tftp_flash 'run load_rd_tftp; run addmisc; bootm ${kernel_flash}'
setenv boot_tftp_tftp 'run load_rd_tftp; run load_kernel_tftp; run addmisc; bootm $
↪{load_addr}'
setenv load_hello_tftp 'tftp ${load_addr_hello} /examples/hello_world.bin'
setenv go_tftp 'run load_hello_tftp; go ${load_addr_hello}'
setenv addmisc 'setenv bootargs ${bootargs} console=ttyS0,${baudrate} rd_start=${rd_
↪start} rd_size=${rd_size} ethaddr=${ethaddr}'
setenv bootcmd 'run boot_tftp_flash'
```

Now you can boot from flash, ide, ide+ext2 and tfp

```
qemu-system-mips -M mips -pflash flash -monitor null -nographic -net nic -net user -
↪tftp `pwd` -hda ide
```

How to debug U-Boot

In order to debug U-Boot you need to start qemu with gdb server support (-s) and waiting the connection to start the CPU (-S)

```
qemu-system-mips -S -s -M mips -pflash flash -monitor null -nographic -net nic -net␣
↪user -tftp `pwd` -hda ide
```

in an other console you start gdb

Debugging of U-Boot Before Relocation

Before relocation, the addresses in the ELF file can be used without any problems by connecting to the gdb server localhost:1234

```
$ mipsel-unknown-linux-gnu-gdb u-boot
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i486-linux-gnu --target=mipsel-unknown-linux-gnu"...
(gdb)  target remote localhost:1234
Remote debugging using localhost:1234
_start () at start.S:64
```

(continues on next page)

(continued from previous page)

```
64          RVECENT(reset,0)        /* U-Boot entry point */
Current language:  auto; currently asm
(gdb)  b board.c:289
Breakpoint 1 at 0xbfc00cc8: file board.c, line 289.
(gdb) c
Continuing.

Breakpoint 1, board_init_f (bootflag=<value optimized out>) at board.c:290
290          relocate_code (addr_sp, id, addr);
Current language:  auto; currently c
(gdb) p/x addr
$1 = 0x87fa0000
```

Debugging of U-Boot After Relocation

For debugging U-Boot after relocation we need to know the address to which U-Boot relocates itself to 0x87fa0000 by default. And replace the symbol table to this offset.

```
(gdb) symbol-file
Discard symbol table from `/private/u-boot-arm/u-boot'? (y or n) y
Error in re-setting breakpoint 1:
No symbol table is loaded.  Use the "file" command.
No symbol file now.
(gdb) add-symbol-file u-boot 0x87fa0000
add symbol table from file "u-boot" at
     .text_addr = 0x87fa0000
(y or n) y
Reading symbols from /private/u-boot-arm/u-boot...done.
Breakpoint 1 at 0x87fa0cc8: file board.c, line 289.
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
0xffffffff87fa0de4 in udelay (usec=<value optimized out>) at time.c:78
78          while ((tmo - read_c0_count()) < 0x7fffffff)
```

QEMU RISC-V

QEMU for RISC-V supports a special 'virt' machine designed for emulation and virtualization purposes. This document describes how to run U-Boot under it. Both 32-bit and 64-bit targets are supported, running in either machine or supervisor mode.

The QEMU virt machine models a generic RISC-V virtual machine with support for the VirtIO standard networking and block storage devices. It has CLINT, PLIC, 16550A UART devices in addition to VirtIO and it also uses device-tree to pass configuration information to guest software. It implements RISC-V privileged architecture spec v1.10.

Building U-Boot

Set the CROSS_COMPILE environment variable as usual, and run:

- For 32-bit RISC-V:

```
make qemu-riscv32_defconfig
make
```

- For 64-bit RISC-V:

```
make qemu-riscv64_defconfig
make
```

This will compile U-Boot for machine mode. To build supervisor mode binaries, use the configurations qemu-riscv32_smode_defconfig and qemu-riscv64_smode_defconfig instead. Note that U-Boot running in supervisor mode requires a supervisor binary interface (SBI), such as RISC-V OpenSBI.

Running U-Boot

The minimal QEMU command line to get U-Boot up and running is:

- For 32-bit RISC-V:

```
qemu-system-riscv32 -nographic -machine virt -bios u-boot
```

- For 64-bit RISC-V:

```
qemu-system-riscv64 -nographic -machine virt -bios u-boot
```

The commands above create targets with 128MiB memory by default. A freely configurable amount of RAM can be created via the '-m' parameter. For example, '-m 2G' creates 2GiB memory for the target, and the memory node in the embedded DTB created by QEMU reflects the new setting.

For instructions on how to run U-Boot in supervisor mode on QEMU with OpenSBI, see the documentation available with OpenSBI: https://github.com/riscv/opensbi/blob/master/docs/platform/qemu_virt.md

These have been tested in QEMU 5.0.0.

Running U-Boot SPL

In the default SPL configuration, U-Boot SPL starts in machine mode. U-Boot proper and OpenSBI (FW_DYNAMIC firmware) are bundled as FIT image and made available to U-Boot SPL. Both are then loaded by U-Boot SPL and the location of U-Boot proper is passed to OpenSBI. After initialization, U-Boot proper is started in supervisor mode by OpenSBI.

OpenSBI must be compiled before compiling U-Boot. Version 0.4 and higher is supported by U-Boot. Clone the OpenSBI repository and run the following command.

```
git clone https://github.com/riscv/opensbi.git
cd opensbi
make PLATFORM=qemu/virt
```

See the OpenSBI documentation for full details: https://github.com/riscv/opensbi/blob/master/docs/platform/qemu_virt.md

To make the FW_DYNAMIC binary (build/platform/qemu/virt/firmware/fw_dynamic.bin) available to U-Boot, either copy it into the U-Boot root directory or specify its location with the OPENSBI environment variable. Afterwards, compile U-Boot with the following commands.

- For 32-bit RISC-V:

```
make qemu-riscv32_spl_defconfig
make
```

- For 64-bit RISC-V:

```
make qemu-riscv64_spl_defconfig
make
```

The minimal QEMU commands to run U-Boot SPL in both 32-bit and 64-bit configurations are:

- For 32-bit RISC-V:

```
qemu-system-riscv32 -nographic -machine virt -bios spl/u-boot-spl \
-device loader,file=u-boot.itb,addr=0x80200000
```

- For 64-bit RISC-V:

```
qemu-system-riscv64 -nographic -machine virt -bios spl/u-boot-spl \
-device loader,file=u-boot.itb,addr=0x80200000
```

An attached disk can be emulated by adding:

```
-device ich9-ahci,id=ahci \
-drive if=none,file=riscv64.img,format=raw,id=mydisk \
-device ide-hd,drive=mydisk,bus=ahci.0
```

You will have to run 'scsi scan' to use it.

QEMU x86

Build instructions for bare mode

To build u-boot.rom for QEMU x86 targets, just simply run:

```
$ make qemu-x86_defconfig (for 32-bit)
$ make qemu-x86_64_defconfig (for 64-bit)
$ make all
```

Note this default configuration will build a U-Boot for the QEMU x86 i440FX board. To build a U-Boot against QEMU x86 Q35 board, you can change the build configuration during the 'make menuconfig' process like below:

```
Device Tree Control  --->
    ...
    (qemu-x86_q35) Default Device Tree for DT control
```

Test with QEMU for bare mode

QEMU is a fancy emulator that can enable us to test U-Boot without access to a real x86 board. Please make sure your QEMU version is 2.3.0 or above test U-Boot. To launch QEMU with u-boot.rom, call QEMU as follows:

```
$ qemu-system-i386 -nographic -bios path/to/u-boot.rom
```

This will instantiate an emulated x86 board with i440FX and PIIX chipset. QEMU also supports emulating an x86 board with Q35 and ICH9 based chipset, which is also supported by U-Boot. To instantiate such a machine, call QEMU with:

```
$ qemu-system-i386 -nographic -bios path/to/u-boot.rom -M q35
```

Note by default QEMU instantiated boards only have 128 MiB system memory. But it is enough to have U-Boot boot and function correctly. You can increase the system memory by pass '-m' parameter to QEMU if you want more memory:

```
$ qemu-system-i386 -nographic -bios path/to/u-boot.rom -m 1024
```

This creates a board with 1 GiB system memory. Currently U-Boot for QEMU only supports 3 GiB maximum system memory and reserves the last 1 GiB address space for PCI device memory-mapped I/O and other stuff, so the maximum value of '-m' would be 3072.

QEMU emulates a graphic card which U-Boot supports. Removing '-nographic' will show QEMU's VGA console window. Note this will disable QEMU's serial output. If you want to check both consoles, use '-serial stdio'.

Multicore is also supported by QEMU via '-smp n' where n is the number of cores to instantiate. Note, the maximum supported CPU number in QEMU is 255.

U-Boot uses 'distro_bootcmd' by default when booting on x86 QEMU. This tries to load a boot script, kernel, and ramdisk from several different interfaces. For the default boot order, see 'qemu-x86.h'. For more information, see 'README.distro'. Most Linux distros can be booted by writing a uboot script. For example, Debian (stretch) can be booted by creating a script file named 'boot.txt' with the contents:

```
setenv bootargs root=/dev/sda1 ro
load ${devtype} ${devnum}:${distro_bootpart} ${kernel_addr_r} /vmlinuz
load ${devtype} ${devnum}:${distro_bootpart} ${ramdisk_addr_r} /initrd.img
zboot ${kernel_addr_r} - ${ramdisk_addr_r} ${filesize}
```

Then compile and install it with:

```
$ apt install u-boot-tools && \
  mkimage -T script -C none -n "Boot script" -d boot.txt /boot/boot.scr
```

The fw_cfg interface in QEMU also provides information about kernel data, initrd, command-line arguments and more. U-Boot supports directly accessing these informtion from fw_cfg interface, which saves the time of loading them from hard disk or network again, through emulated devices. To use it , simply providing them in QEMU command line:

```
$ qemu-system-i386 -nographic -bios path/to/u-boot.rom -m 1024 \
  -kernel /path/to/bzImage -append 'root=/dev/ram console=ttyS0' \
  -initrd /path/to/initrd -smp 8
```

Note: -initrd and -smp are both optional

Then start QEMU, in U-Boot command line use the following U-Boot command to setup kernel:

```
=> qfw
qfw - QEMU firmware interface

Usage:
qfw <command>
    - list                             : print firmware(s) currently loaded
    - cpus                             : print online cpu number
    - load <kernel addr> <initrd addr> : load kernel and initrd (if any) and setup for␣
↪zboot

=> qfw load
loading kernel to address 01000000 size 5d9d30 initrd 04000000 size 1b1ab50
```

Here the kernel (bzImage) is loaded to 01000000 and initrd is to 04000000. Then, 'zboot' can be used to boot the kernel:

```
=> zboot 01000000 - 04000000 1b1ab50
```

To run 64-bit U-Boot, qemu-system-x86_64 should be used instead, e.g.:

```
$ qemu-system-x86_64 -nographic -bios path/to/u-boot.rom
```

A specific CPU can be specified via the '-cpu' parameter but please make sure the specified CPU supports 64-bit like '-cpu core2duo'. Conversely '-cpu pentium' won't work for obvious reasons that the processor only supports 32-bit.

Note 64-bit support is very preliminary at this point. Lots of features are missing in the 64-bit world. One notable feature is the VGA console support which is currently missing, so that you must specify '-nographic' to get 64-bit U-Boot up and running.

7.1.7 Freescale

B4860QDS

The B4860QDS is a Freescale reference board that hosts the B4860 SoC (and variants).

B4860 Overview

The B4860 QorIQ Qonverge device is a Freescale high-end, multicore SoC based on StarCore and Power Architecture® cores. It targets the broadband wireless infrastructure and builds upon the proven success of the existing multicore DSPs and Power CPUs. It is designed to bolster the rapidly changing and expanding wireless markets, such as 3GLTE (FDD and TDD), LTE-Advanced, and UMTS.

The B4860 is a highly-integrated StarCore and Power Architecture processor that contains:

- Six fully-programmable StarCore SC3900 FVP subsystems, divided into three clusters-each core runs up to 1.2 GHz, with an architecture highly optimized for wireless base station applications
- Four dual-thread e6500 Power Architecture processors organized in one cluster-each core runs up to 1.8 GHz
- Two DDR3/3L controllers for high-speed, industry-standard memory interface each runs at up to 1866.67 MHz
- MAPLE-B3 hardware acceleration-for forward error correction schemes including Turbo or Viterbi decoding, Turbo encoding and rate matching, MIMO MMSE equalization scheme, matrix operations, CRC insertion and check, DFT/iDFT and FFT/iFFT calculations, PUSCH/PDSCH acceleration, and UMTS chip rate acceleration
- CoreNet fabric that fully supports coherency using MESI protocol between the e6500 cores, SC3900 FVP cores, memories and external interfaces. CoreNet fabric interconnect runs at 667 MHz and supports coherent and non-coherent out of order transactions with prioritization and bandwidth allocation amongst CoreNet endpoints.
- Data Path Acceleration Architecture, which includes the following:
  - Frame Manager (FMan), which supports in-line packet parsing and general classification to enable policing and QoS-based packet distribution
  - Queue Manager (QMan) and Buffer Manager (BMan), which allow offloading of queue management, task management, load distribution, flow ordering, buffer management, and allocation tasks from the cores
  - Security engine (SEC 5.3)-crypto-acceleration for protocols such as IPsec, SSL, and 802.16
  - RapidIO manager (RMAN) - Support SRIO types 8, 9, 10, and 11 (inbound and outbound). Supports types 5, 6 (outbound only)

- Large internal cache memory with snooping and stashing capabilities for bandwidth saving and high utilization of processor elements. The 9856-Kbyte internal memory space includes the following:
  - 32 Kbyte L1 ICache per e6500/SC3900 core
  - 32 Kbyte L1 DCache per e6500/SC3900 core
  - 2048 Kbyte unified L2 cache for each SC3900 FVP cluster
  - 2048 Kbyte unified L2 cache for the e6500 cluster
  - Two 512 Kbyte shared L3 CoreNet platform caches (CPC)
- Sixteen 10-GHz SerDes lanes serving:
  - Two Serial RapidIO interfaces
  - Each supports up to 4 lanes and a total of up to 8 lanes
- Up to 8-lanes Common Public Radio Interface (CPRI) controller for glue-less antenna connection
- Two 10-Gbit Ethernet controllers (10GEC)
- Six 1G/2.5-Gbit Ethernet controllers for network communications
- PCI Express controller
- Debug (Aurora)
- Two OCeaN DMAs
- Various system peripherals
- 182 32-bit timers

B4860QDS Overview

- DDRC1: Ten separate DDR3 parts of 16-bit to support 72-bit (ECC) at 1866MT/s, ECC, 4 GB of memory in two ranks of 2 GB.
- DDRC2: Five separate DDR3 parts of 16-bit to support 72-bit (ECC) at 1866MT/s, ECC, 2 GB of memory. Single rank.
- SerDes 1 multiplexing: Two Vitesse (transmit and receive path) cross-point 16x16 switch VSC3316
- SerDes 2 multiplexing: Two Vitesse (transmit and receive path) cross-point 8x8 switch VSC3308
- USB 2.0 ULPI PHY USB3315 by SMSC supports USB port in host mode. B4860 UART port is available over USB-to-UART translator USB2SER or over RS232 flat cable.
- A Vitesse dual SGMII phy VSC8662 links the B4860 SGMII lines to 2xRJ-45 copper connectors for Stand-alone mode and to the 1000Base-X over AMC MicroTCA connector ports 0 and 2 for AMC mode.
- The B4860 configuration may be loaded from nine bits coded reset configuration reset source. The RCW source is set by appropriate DIP-switches.
- 16-bit NOR Flash / PROMJet
- QIXIS 8-bit NOR Flash Emulator
- 8-bit NAND Flash
- 24-bit SPI Flash
- Long address I2C EEPROM
- Available debug interfaces are:
  - On-board eCWTAP controller with ETH and USB I/F
  - JTAG/COP 16-pin header for any external TAP controller
  - External JTAG source over AMC to support B2B configuration

  - 70-pin Aurora debug connector

- **QIXIS (FPGA) logic:**

    - 2 KB internal memory space including

- IDT840NT4 clock synthesizer provides B4860 essential clocks : SYSCLK, DDRCLK1,2 and RTCCLK.
- Two 8T49N222A SerDes ref clock devices support two SerDes port clock frequency - total four refclk, including CPRI clock scheme.

B4420 Personality

B4420 is a reduced personality of B4860 with less core/clusters(both SC3900 and e6500), less DDR controllers, less serdes lanes, less SGMII interfaces and reduced target frequencies.

Key differences between B4860 and B4420

B4420 has:

1. Less e6500 cores: 1 cluster with 2 e6500 cores
2. Less SC3900 cores/clusters: 1 cluster with 2 SC3900 cores per cluster
3. Single DDRC
4. 2X 4 lane serdes
5. 3 SGMII interfaces
6. no sRIO
7. no 10G

B4860QDS Default Settings

Switch Settings

```
SW1  OFF [0] OFF [0] OFF [0] OFF [0] OFF [0] OFF [0] OFF [0] OFF [0]
SW2  ON      ON      ON      ON      ON      ON      OFF     OFF
SW3  OFF     OFF     OFF     ON      OFF     OFF     ON      OFF
SW5  OFF     OFF     OFF     OFF     OFF     OFF     ON      ON
```

Note:

- PCIe slots modes: All the PCIe devices work as Root Complex.
- Boot location: NOR flash.

SysClk/Core(e6500)/CCB/DDR/FMan/DDRCLK/StarCore/CPRI-Maple/eTVPE-Maple/ULB-Maple
66MHz/1.6GHz/667MHz/1.6GHz data rate/667MHz/133MHz/1200MHz/500MHz/800MHz/667MHz

NAND boot:

```
SW1 [1.1] = 0
SW2 [1.1] = 1
SW3 [1:4] = 0001
```

NOR boot:

```
SW1 [1.1] = 1
SW2 [1.1] = 0
SW3 [1:4] = 1000
```

B4420QDS Default Settings

Switch Settings

```
SW1  OFF[0]  OFF [0] OFF [0] OFF [0] OFF [0] OFF [0] OFF [0] OFF [0]
SW2  ON      OFF     ON      OFF     ON      ON      OFF     OFF
SW3  OFF     OFF     OFF     ON      OFF     OFF     ON      OFF
SW5  OFF     OFF     OFF     OFF     OFF     OFF     ON      ON
```

Note:

- PCIe slots modes: All the PCIe devices work as Root Complex.
- Boot location: NOR flash.

SysClk/Core(e6500)/CCB/DDR/FMan/DDRCLK/StarCore/CPRI-Maple/eTVPE-Maple/ULB-Maple
66MHz/1.6GHz/667MHz/1.6GHz data rate/667MHz/133MHz/1200MHz/500MHz/800MHz/667MHz

NAND boot:

```
SW1 [1.1] = 0
SW2 [1.1] = 1
SW3 [1:4] = 0001
```

NOR boot:

```
SW1 [1.1] = 1
SW2 [1.1] = 0
SW3 [1:4] = 1000
```

Memory map on B4860QDS

The addresses in brackets are physical addresses.

Start Address	End Address	Description	Size
0xF_FFDF_1000	0xF_FFFF_FFFF	Free	2 MB
0xF_FFDF_0000	0xF_FFDF_0FFF	IFC - FPGA	4 KB
0xF_FF81_0000	0xF_FFDE_FFFF	Free	5 MB
0xF_FF80_0000	0xF_FF80_FFFF	IFC NAND Flash	64 KB
0xF_FF00_0000	0xF_FF7F_FFFF	Free	8 MB
0xF_FE00_0000	0xF_FEFF_FFFF	CCSRBAR	16 MB
0xF_F801_0000	0xF_FDFF_FFFF	Free	95 MB
0xF_F800_0000	0xF_F800_FFFF	PCIe I/O Space	64 KB
0xF_F600_0000	0xF_F7FF_FFFF	QMAN s/w portal	32 MB
0xF_F400_0000	0xF_F5FF_FFFF	BMAN s/w portal	32 MB
0xF_F000_0000	0xF_F3FF_FFFF	Free	64 MB
0xF_E800_0000	0xF_EFFF_FFFF	IFC NOR Flash	128 MB
0xF_E000_0000	0xF_E7FF_FFFF	Promjet	128 MB
0xF_A0C0_0000	0xF_DFFF_FFFF	Free	1012 MB
0xF_A000_0000	0xF_A0BF_FFFF	MAPLE0/1/2	12 MB
0xF_0040_0000	0xF_9FFF_FFFF	Free	12 GB
0xF_0000_0000	0xF_01FF_FFFF	DCSR	32 MB
0xC_4000_0000	0xE_FFFF_FFFF	Free	11 GB
0xC_3000_0000	0xC_3FFF_FFFF	sRIO-2 I/O	256 MB
0xC_2000_0000	0xC_2FFF_FFFF	sRIO-1 I/O	256 MB
0xC_0000_0000	0xC_1FFF_FFFF	PCIe Mem Space	512 MB
0x1_0000_0000	0xB_FFFF_FFFF	Free	44 GB
0x0_8000_0000	0x0_FFFF_FFFF	DDRC1	2 GB
0x0_0000_0000	0x0_7FFF_FFFF	DDRC2	2 GB

Memory map on B4420QDS

The addresses in brackets are physical addresses.

Start Address	End Address	Description	Size
0xF_FFDF_1000	0xF_FFFF_FFFF	Free	2 MB
0xF_FFDF_0000	0xF_FFDF_0FFF	IFC - FPGA	4 KB
0xF_FF81_0000	0xF_FFDE_FFFF	Free	5 MB
0xF_FF80_0000	0xF_FF80_FFFF	IFC NAND Flash	64 KB
0xF_FF00_0000	0xF_FF7F_FFFF	Free	8 MB
0xF_FE00_0000	0xF_FEFF_FFFF	CCSRBAR	16 MB
0xF_F801_0000	0xF_FDFF_FFFF	Free	95 MB
0xF_F800_0000	0xF_F800_FFFF	PCIe I/O Space	64 KB
0xF_F600_0000	0xF_F7FF_FFFF	QMAN s/w portal	32 MB
0xF_F400_0000	0xF_F5FF_FFFF	BMAN s/w portal	32 MB
0xF_F000_0000	0xF_F3FF_FFFF	Free	64 MB
0xF_E800_0000	0xF_EFFF_FFFF	IFC NOR Flash	128 MB
0xF_E000_0000	0xF_E7FF_FFFF	Promjet	128 MB
0xF_A0C0_0000	0xF_DFFF_FFFF	Free	1012 MB
0xF_A000_0000	0xF_A0BF_FFFF	MAPLE0/1/2	12 MB
0xF_0040_0000	0xF_9FFF_FFFF	Free	12 GB
0xF_0000_0000	0xF_01FF_FFFF	DCSR	32 MB
0xC_4000_0000	0xE_FFFF_FFFF	Free	11 GB
0xC_3000_0000	0xC_3FFF_FFFF	sRIO-2 I/O	256 MB
0xC_2000_0000	0xC_2FFF_FFFF	sRIO-1 I/O	256 MB
0xC_0000_0000	0xC_1FFF_FFFF	PCIe Mem Space	512 MB
0x1_0000_0000	0xB_FFFF_FFFF	Free	44 GB
0x0_0000_0000	0x0_FFFF_FFFF	DDRC1	4 GB

NOR Flash memory Map on B4860 and B4420QDS

Start	End	Definition	Size
0xEFF40000	0xEFFFFFFF	U-Boot (current bank)	768KB
0xEFF20000	0xEFF3FFFF	U-Boot env (current bank)	128KB
0xEFF00000	0xEFF1FFFF	FMAN Ucode (current bank)	128KB
0xEF300000	0xEFEFFFFF	rootfs (alternate bank)	12MB
0xEE800000	0xEE8FFFFF	device tree (alternate bank)	1MB
0xEE020000	0xEE6FFFFF	Linux.uImage (alternate bank)	6MB+896KB
0xEE000000	0xEE01FFFF	RCW (alternate bank)	128KB
0xEDF40000	0xEDFFFFFF	U-Boot (alternate bank)	768KB
0xEDF20000	0xEDF3FFFF	U-Boot env (alternate bank)	128KB
0xEDF00000	0xEDF1FFFF	FMAN ucode (alternate bank)	128KB
0xED300000	0xEDEFFFFF	rootfs (current bank)	12MB
0xEC800000	0xEC8FFFFF	device tree (current bank)	1MB
0xEC020000	0xEC6FFFFF	Linux.uImage (current bank)	6MB+896KB
0xEC000000	0xEC01FFFF	RCW (current bank)	128KB

Various Software configurations/environment variables/commands

The below commands apply to both B4860QDS and B4420QDS.

U-Boot environment variable hwconfig

The default hwconfig is:

```
hwconfig=fsl_ddr:ctlr_intlv=null,bank_intlv=cs0_cs1;usb1:dr_mode=host,phy_type=ulpi
```

Note: For USB gadget set "dr_mode=peripheral"

FMAN Ucode versions

fsl_fman_ucode_B4860_106_3_6.bin

Switching to alternate bank

Commands for switching to alternate bank.

1. To change from vbank0 to vbank2

```
=> qixis_reset altbank (it will boot using vbank2)
```

2. To change from vbank2 to vbank0

```
=> qixis reset (it will boot using vbank0)
```

To change personality of board

For changing personality from B4860 to B4420

1. Boot from vbank0
2. Flash vbank2 with b4420 rcw and U-Boot

3. Give following commands to uboot prompt

```
=> mw.b ffdf0040 0x30;
=> mw.b ffdf0010 0x00;
=> mw.b ffdf0062 0x02;
=> mw.b ffdf0050 0x02;
=> mw.b ffdf0010 0x30;
=> reset
```

Note:

- Power off cycle will lead to default switch settings.
- 0xffdf0000 is the address of the QIXIS FPGA.

Switching between NOR and NAND boot(RCW src changed from NOR <-> NAND)

To change from NOR to NAND boot give following command on uboot prompt

```
=> mw.b ffdf0040 0x30
=> mw.b ffdf0010 0x00
=> mw.b 0xffdf0050 0x08
=> mw.b 0xffdf0060 0x82
=> mw.b ffdf0061 0x00
=> mw.b ffdf0010 0x30
=> reset
```

To change from NAND to NOR boot give following command on uboot prompt:

```
=> mw.b ffdf0040 0x30
=> mw.b ffdf0010 0x00
=> mw.b 0xffdf0050 0x00(for vbank0) or (mw.b 0xffdf0050 0x02 for vbank2)
=> mw.b 0xffdf0060 0x12
=> mw.b ffdf0061 0x01
=> mw.b ffdf0010 0x30
=> reset
```

Note:

- Power off cycle will lead to default switch settings.
- 0xffdf0000 is the address of the QIXIS FPGA.

Ethernet interfaces for B4860QDS

Serdes protocosl tested: * 0x2a, 0x8d (serdes1, serdes2) [DEFAULT] * 0x2a, 0xb2 (serdes1, serdes2)

When using [DEFAULT] RCW, which including 2 * 1G SGMII on board and 2 * 1G SGMII on SGMII riser card.

Under U-Boot these network interfaces are recognized as:

```
FM1@DTSEC3, FM1@DTSEC4, FM1@DTSEC5 and FM1@DTSEC6.
```

On Linux the interfaces are renamed as:

```
eth2 -> fm1-gb2
eth3 -> fm1-gb3
eth4 -> fm1-gb4
eth5 -> fm1-gb5
```

RCW and Ethernet interfaces for B4420QDS

Serdes protocosl tested: * 0x18, 0x9e (serdes1, serdes2)

Under U-Boot these network interfaces are recognized as:

```
FM1@DTSEC3, FM1@DTSEC4 and  e1000#0.
```

On Linux the interfaces are renamed as:

```
eth2 -> fm1-gb2
eth3 -> fm1-gb3
```

NAND boot with 2 Stage boot loader

PBL initialise the internal SRAM and copy SPL(160KB) in SRAM. SPL further initialise DDR using SPD and environment variables and copy U-Boot(768 KB) from flash to DDR. Finally SPL transer control to U-Boot for futher booting.

SPL has following features:

- Executes within 256K
- No relocation required

Run time view of SPL framework during boot:

Area | Address
Secure boot | 0xFFFC0000 (32KB) headers |
GD, BD | 0xFFFC8000 (4KB)
ENV | 0xFFFC9000 (8KB)
HEAP | 0xFFFCB000 (30KB)
STACK | 0xFFFD8000 (22KB)
U-Boot SPL | 0xFFFD8000 (160KB)

NAND Flash memory Map on B4860 and B4420QDS

Start	End	Definition	Size
0x000000	0x0FFFFF	U-Boot	1MB
0x140000	0x15FFFF	U-Boot env	128KB
0x1A0000	0x1BFFFF	FMAN Ucode	128KB

imx8mm_evk

U-Boot for the NXP i.MX8MM EVK board

Quick Start

- Build the ARM Trusted firmware binary
- Get ddr firmware
- Build U-Boot
- Boot

Get and Build the ARM Trusted firmware

Note: builddir is U-Boot build directory (source directory for in-tree builds) Get ATF from: https://source.codeaurora.org/external/imx/imx-atf branch: imx_4.19.35_1.0.0

```
$ make PLAT=imx8mm bl31
$ cp build/imx8mm/release/bl31.bin $(builddir)
```

Get the ddr firmware

```
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.0.bin
$ chmod +x firmware-imx-8.0.bin
$ ./firmware-imx-8.0
$ cp firmware-imx-8.0/firmware/ddr/synopsys/lpddr4*.bin $(builddir)
```

Build U-Boot

```
$ export CROSS_COMPILE=aarch64-poky-linux-
$ make imx8mm_evk_defconfig
$ export ATF_LOAD_ADDR=0x920000
$ make flash.bin
```

Burn the flash.bin to MicroSD card offset 33KB:

```
$sudo dd if=flash.bin of=/dev/sd[x] bs=1024 seek=33 conv=notrunc
```

Boot

Set Boot switch to SD boot

imx8mn_evk

U-Boot for the NXP i.MX8MN EVK board

Quick Start

- Build the ARM Trusted firmware binary
- Get firmware-imx package
- Build U-Boot
- Boot

Get and Build the ARM Trusted firmware

Note: srctree is U-Boot source directory Get ATF from: https://source.codeaurora.org/external/imx/imx-atf branch: imx_4.19.35_1.1.0

```
$ make PLAT=imx8mn bl31
$ cp build/imx8mn/release/bl31.bin $(srctree)
```

Get the ddr firmware

```
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.5.bin
$ chmod +x firmware-imx-8.5.bin
$ ./firmware-imx-8.5
$ cp firmware-imx-8.5/firmware/ddr/synopsys/ddr4*.bin $(srctree)
```

Build U-Boot

```
$ export CROSS_COMPILE=aarch64-poky-linux-
$ make imx8mn_ddr4_evk_defconfig
$ export ATF_LOAD_ADDR=0x960000
$ make flash.bin
```

Burn the flash.bin to MicroSD card offset 32KB:

```
$sudo dd if=flash.bin of=/dev/sd[x] bs=1024 seek=32 conv=notrunc
```

Boot

Set Boot switch to SD boot

imx8mp_evk

U-Boot for the NXP i.MX8MP EVK board

Quick Start

- Build the ARM Trusted firmware binary
- Get the firmware-imx package
- Build U-Boot
- Boot

Get and Build the ARM Trusted firmware

Note: $(srctree) is the U-Boot source directory Get ATF from: https://source.codeaurora.org/external/imx/imx-atf branch: imx_5.4.3_2.0.0

```
$ make PLAT=imx8mp bl31
$ sudo cp build/imx8mp/release/bl31.bin $(srctree)
```

Get the ddr firmware

```
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.7.bin
$ chmod +x firmware-imx-8.7.bin
$ ./firmware-imx-8.7
$ sudo cp firmware-imx-8.7/firmware/ddr/synopsys/lpddr4_pmu_train_1d_dmem_201904.bin
↪$(srctree)/lpddr4_pmu_train_1d_dmem.bin
```

(continues on next page)

(continued from previous page)

```
$ sudo cp firmware-imx-8.7/firmware/ddr/synopsys/lpddr4_pmu_train_1d_imem_201904.bin
↪$(srctree)/lpddr4_pmu_train_1d_imem.bin
$ sudo cp firmware-imx-8.7/firmware/ddr/synopsys/lpddr4_pmu_train_2d_dmem_201904.bin
↪$(srctree)/lpddr4_pmu_train_2d_dmem.bin
$ sudo cp firmware-imx-8.7/firmware/ddr/synopsys/lpddr4_pmu_train_2d_imem_201904.bin
↪$(srctree)/lpddr4_pmu_train_2d_imem.bin
```

Build U-Boot

```
$ export CROSS_COMPILE=aarch64-poky-linux-
$ make imx8mp_evk_defconfig
$ export ATF_LOAD_ADDR=0x960000
$ make flash.bin
```

Burn the flash.bin to the MicroSD card at offset 32KB:

```
$sudo dd if=flash.bin of=/dev/sd[x] bs=1K seek=32 conv=notrunc; sync
```

Boot

Set Boot switch to SD boot Use /dev/ttyUSB2 for U-Boot console

imx8mq_evk

U-Boot for the NXP i.MX8MQ EVK board

Quick Start

- Build the ARM Trusted firmware binary
- Get ddr and hdmi fimware
- Build U-Boot
- Boot

Get and Build the ARM Trusted firmware

Note: srctree is U-Boot source directory Get ATF from: https://source.codeaurora.org/external/imx/imx-atf branch: imx_4.19.35_1.0.0

```
$ make PLAT=imx8mq bl31
$ cp build/imx8mq/release/bl31.bin $(builddir)
```

Get the ddr and hdmi firmware

```
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-7.9.bin
$ chmod +x firmware-imx-7.9.bin
$ ./firmware-imx-7.9.bin
$ cp firmware-imx-7.9/firmware/hdmi/cadence/signed_hdmi_imx8m.bin $(builddir)
$ cp firmware-imx-7.9/firmware/ddr/synopsys/lpddr4*.bin $(builddir)
```

Build U-Boot

```
$ export CROSS_COMPILE=aarch64-poky-linux-
$ make imx8mq_evk_defconfig
$ make flash.bin
```

Burn the flash.bin to MicroSD card offset 33KB:

```
$sudo dd if=flash.bin of=/dev/sd[x] bs=1024 seek=33 conv=notrunc
```

Boot

Set Boot switch SW801: 1100 and Bmode: 10 to boot from Micro SD.

imx8qxp_mek

U-Boot for the NXP i.MX8QXP EVK board

Quick Start

- Build the ARM Trusted firmware binary
- Get scfw_tcm.bin and ahab-container.img
- Build U-Boot
- Flash the binary into the SD card
- Boot

Get and Build the ARM Trusted firmware

```
$ git clone https://source.codeaurora.org/external/imx/imx-atf
$ cd imx-atf/
$ git checkout origin/imx_4.19.35_1.1.0 -b imx_4.19.35_1.1.0
$ make PLAT=imx8qx bl31
```

Get scfw_tcm.bin and ahab-container.img

```
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/imx-sc-firmware-1.2.7.1.bin
$ chmod +x imx-sc-firmware-1.2.7.1.bin
$ ./imx-sc-firmware-1.2.7.1.bin
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/imx-seco-2.3.1.bin
$ chmod +x imx-seco-2.3.1.bin
$ ./imx-seco-2.3.1.bin
```

Copy the following binaries to U-Boot folder:

```
$ cp imx-atf/build/imx8qx/release/bl31.bin .
$ cp imx-seco-2.3.1/firmware/seco/mx8qx-ahab-container.img ./ahab-container.img
$ cp imx-sc-firmware-1.2.7.1/mx8qx-mek-scfw-tcm.bin   .
```

Build U-Boot

```
$ make imx8qxp_mek_defconfig
$ make flash.bin
```

Flash the binary into the SD card

Burn the flash.bin binary to SD card offset 32KB:

```
$ sudo dd if=flash.bin of=/dev/sd[x] bs=1024 seek=32 conv=notrunc
```

Boot

Set Boot switch SW2: 1100.

imxrt1020-evk

How to use U-Boot on NXP i.MXRT1020 EVK

- Build U-Boot for i.MXRT1020 EVK:

```
$ make mrproper
$ make imxrt1020-evk_defconfig
$ make
```

This will generate the SPL image called SPL and the u-boot.img.

- Flash the SPL image into the micro SD card:

```
$sudo dd if=SPL of=/dev/sdX bs=1k seek=1 conv=notrunc; sync
```

- Flash the u-boot.img image into the micro SD card:

```
$sudo dd if=u-boot.img of=/dev/sdX bs=1k seek=128 conv=notrunc; sync
```

- Jumper settings:

  ```
  SW8: 0 1 1 0
  ```

where 0 means bottom position and 1 means top position (from the switch label numbers reference).

- Connect the USB cable between the EVK and the PC for the console. The USB console connector is the one close the ethernet connector
- Insert the micro SD card in the board, power it up and U-Boot messages should come up.

imxrt1050-evk

How to use U-Boot on NXP i.MXRT1050 EVK

- Build U-Boot for i.MXRT1050 EVK:

```
$ make mrproper
$ make imxrt1050-evk_defconfig
$ make
```

This will generate the SPL image called SPL and the u-boot.img.

- Flash the SPL image into the micro SD card:

```
$sudo dd if=SPL of=/dev/sdX bs=1k seek=1 conv=notrunc; sync
```

- Flash the u-boot.img image into the micro SD card:

```
$sudo dd if=u-boot.img of=/dev/sdX bs=1k seek=128 conv=notrunc; sync
```

- Jumper settings:

  ```
  SW7: 1 0 1 0
  ```

where 0 means bottom position and 1 means top position (from the switch label numbers reference).

- Connect the USB cable between the EVK and the PC for the console. The USB console connector is the one close the ethernet connector
- Insert the micro SD card in the board, power it up and U-Boot messages should come up.

mx6sabreauto

How to use and build U-Boot on mx6sabreauto

mx6sabreauto_defconfig target supports mx6q/mx6dl/mx6qp sabreauto variants.

In order to build it:

```
$ make mx6sabreauto_defconfig
$ make
```

This will generate the SPL and u-boot-dtb.img binaries.

- Flash the SPL binary into the SD card:

```
$ sudo dd if=SPL of=/dev/sdX bs=1K seek=1 conv=notrunc && sync
```

- Flash the u-boot-dtb.img binary into the SD card:

```
$ sudo dd if=u-boot-dtb.img of=/dev/sdX bs=1K seek=69 conv=notrunc && sync
```

Booting via Falcon mode

Write in mx6sabreauto_defconfig the following define below:

CONFIG_SPL_OS_BOOT=y

In order to build it:

```
$ make mx6sabreauto_defconfig
$ make
```

This will generate the SPL image called SPL and the u-boot-dtb.img.

- Flash the SPL image into the SD card:

```
$ sudo dd if=SPL of=/dev/sdb bs=1K seek=1 conv=notrunc && sync
```

- Flash the u-boot-dtb.img image into the SD card:

```
$ sudo dd if=u-boot-dtb.img of=/dev/sdb bs=1K seek=69 conv=notrunc && sync
```

Create a FAT16 boot partition to store uImage and the dtb file, then copy the files there:

```
$ sudo cp uImage /media/boot
$ sudo cp imx6dl-sabreauto.dtb /media/boot
```

Create a partition for root file system and extract it there:

```
$ sudo tar xvf rootfs.tar.gz -C /media/root
```

The SD card must have enough space for raw "args" and "kernel". To configure Falcon mode for the first time, on U-Boot do the following commands:

- Load dtb file from boot partition:

```
# load mmc 0:1 ${fdt_addr} imx6dl-sabreauto.dtb
```

- Load kernel image from boot partition:

```
# load mmc 0:1 ${loadaddr} uImage
```

- Write kernel at 2MB offset:

```
# mmc write ${loadaddr} 0x1000 0x4000
```

- Setup kernel bootargs:

```
# setenv bootargs "console=ttymxc3,115200 root=/dev/mmcblk0p1 rootfstype=ext4 rootwait quiet rw"
```

- Prepare args:

```
# spl export fdt ${loadaddr} - ${fdt_addr}
```

- Write args 1MB data (0x800 sectors) to 1MB offset (0x800 sectors):

```
# mmc write 18000000 0x800 0x800
```

- Restart the board and then SPL binary will launch the kernel directly.

mx6sabresd

How to use and build U-Boot on mx6sabresd

The following methods can be used for booting mx6sabresd boards:

1. Booting from SD card
2. Booting from eMMC
3. Booting via Falcon mode (SPL launches the kernel directly)

1. Booting from SD card via SPL

mx6sabresd_defconfig target supports mx6q/mx6dl/mx6qp sabresd variants.

In order to build it:

```
$ make mx6sabresd_defconfig
$ make
```

This will generate the SPL and u-boot-dtb.img binaries.

- Flash the SPL binary into the SD card:

```
$ sudo dd if=SPL of=/dev/sdX bs=1K seek=1 conv=notrunc && sync
```

- Flash the u-boot-dtb.img binary into the SD card:

```
$ sudo dd if=u-boot-dtb.img of=/dev/sdX bs=1K seek=69 conv=notrunc && sync
```

2. Booting from eMMC

```
$ make mx6sabresd_defconfig
$ make
```

This will generate the SPL and u-boot-dtb.img binaries.

- Boot first from SD card as shown in the previous section

In U-boot change the eMMC partition config:

```
=> mmc partconf 2 1 0 0
```

Mount the eMMC in the host PC:

```
=> ums 0 mmc 2
```

- Flash SPL and u-boot-dtb.img binaries into the eMMC:

```
$ sudo dd if=SPL of=/dev/sdX bs=1K seek=1 conv=notrunc && sync
$ sudo dd if=u-boot-dtb.img of=/dev/sdX bs=1K seek=69 conv=notrunc && sync
```

Set SW6 to eMMC 8-bit boot: 11010110

3. Booting via Falcon mode

```
$ make mx6sabresd_defconfig
$ make
```

This will generate the SPL image called SPL and the u-boot-dtb.img.

- Flash the SPL image into the SD card

```
$ sudo dd if=SPL of=/dev/sdX bs=1K seek=1 oflag=sync status=none conv=notrunc && sync
```

- Flash the u-boot-dtb.img image into the SD card

```
$ sudo dd if=u-boot-dtb.img of=/dev/sdX bs=1K seek=69 oflag=sync status=none␣
↪conv=notrunc && sync
```

Create a partition for root file system and extract it there

```
$ sudo tar xvf rootfs.tar.gz -C /media/root
```

The SD card must have enough space for raw "args" and "kernel". To configure Falcon mode for the first time, on U-Boot do the following commands:

- Setup the IP server:

```
# setenv serverip <server_ip_address>
```

- Download dtb file:

```
# dhcp ${fdt_addr} imx6q-sabresd.dtb
```

- Download kernel image:

```
# dhcp ${loadaddr} uImage
```

- Write kernel at 2MB offset:

```
# mmc write ${loadaddr} 0x1000 0x4000
```

- Setup kernel bootargs:

```
# setenv bootargs "console=ttymxc0,115200 root=/dev/mmcblk1p1 rootfstype=ext4␣
↪rootwait quiet rw"
```

- Prepare args:

```
# spl export fdt ${loadaddr} - ${fdt_addr}
```

- Write args 1MB data (0x800 sectors) to 1MB offset (0x800 sectors):

```
# mmc write 18000000 0x800 0x800
```

- Press KEY_VOL_UP key, power up the board and then SPL binary will launch the kernel directly.

mx6ul_14x14_evk

How to use U-Boot on Freescale MX6UL 14x14 EVK

- Build U-Boot for MX6UL 14x14 EVK:

```
$ make mrproper
$ make mx6ul_14x14_evk_defconfig
$ make
```

This will generate the SPL image called SPL and the u-boot.img.

1. Booting via SDCard

- Flash the SPL image into the micro SD card:

```
sudo dd if=SPL of=/dev/mmcblk0 bs=1k seek=1 conv=notrunc; sync
```

- Flash the u-boot.img image into the micro SD card:

```
sudo dd if=u-boot.img of=/dev/mmcblk0 bs=1k seek=69 conv=notrunc; sync
```

- Jumper settings:

```
SW601: 0 0 1 0
Sw602: 1 0
```

where 0 means bottom position and 1 means top position (from the switch label numbers reference).

- Connect the USB cable between the EVK and the PC for the console. The USB console connector is the one close the push buttons
- Insert the micro SD card in the board, power it up and U-Boot messages should come up.

2. Booting via Serial Download Protocol (SDP)

The mx6ulevk board can boot from USB OTG port using the SDP, target will enter in SDP mode in case an SD Card is not connect or boot switches are set as below:

```
Sw602: 0 1
SW601: x x x x
```

The following tools can be used to boot via SDP, for both tools you must connect an USB cable in USB OTG port.

- Method 1: Universal Update Utility (uuu)

The UUU binary can be downloaded in release tab from link below: https://github.com/NXPmicro/mfgtools

The following script should be created to boot SPL + u-boot-dtb.img binaries:

```
$ cat uuu_script
  uuu_version 1.1.4

  SDP: boot -f SPL
  SDPU: write -f u-boot-dtb.img -addr 0x877fffc0
  SDPU: jump -addr 0x877fffc0
  SDPU: done
```

Please note that the address above is calculated based on SYS_TEXT_BASE address:

0x877fffc0 = 0x87800000 (SYS_TEXT_BASE) - 0x40 (U-Boot proper Header size)

Power on the target and run the following command from U-Boot root directory:

```
$ sudo ./uuu uuu_script
```

- Method 2: imx usb loader tool (imx_usb):

The imx_usb_loader tool can be downloaded in link below: https://github.com/boundarydevices/imx_usb_loader

Build the source code and run the following commands from U-Boot root directory:

```
$ sudo ./imx_usb SPL
$ sudo ./imx_usb u-boot-dtb.img
```

mx6ullevk

How to use U-Boot on Freescale MX6ULL 14x14 EVK

- First make sure you have installed the dtc package (device tree compiler):

```
$ sudo apt-get install device-tree-compiler
```

- Build U-Boot for MX6ULL 14x14 EVK:

```
$ make mrproper
$ make mx6ull_14x14_evk_defconfig
$ make
```

This generates the u-boot-dtb.imx image in the current directory.

- Flash the u-boot-dtb.imx image into the micro SD card:

```
$ sudo dd if=u-boot-dtb.imx of=/dev/sdb bs=1K seek=1 conv=notrunc && sync
```

- Jumper settings:

```
SW601: 0 0 1 0
Sw602: 1 0
```

Where 0 means bottom position and 1 means top position (from the switch label numbers reference).

Connect the USB cable between the EVK and the PC for the console. (The USB console connector is the one close the push buttons)

Insert the micro SD card in the board, power it up and U-Boot messages should come up.

The link for the board: http://www.nxp.com/products/microcontrollers-and- processors/arm-processors/i.mx-applications-processors/i.mx-6-processors/ i.mx6qp/evaluation-kit-for-the-i.mx-6ull-applications-processor:MCIMX6ULL-EVK

7.1.8 Google

Chromebook Coral

Coral is a Chromebook (or really about 20 different Chromebooks) which use the Intel Apollo Lake platform (APL). The 'reef' Chromebooks use the same APL SoC so should also work. Some later ones based on Glacier Lake (GLK) need various changes in GPIOs, etc. but are very similar.

It is hoped that this port can enable ports to embedded APL boards which are starting to appear.

Note that booting U-Boot on APL is already supported by coreboot and Slim Bootloader. This documentation refers to a 'bare metal' port.

Boot flow - TPL

Apollo Lake boots via an IFWI (Integrated Firmware Image). TPL is placed in this, in the IBBL entry.

On boot, an on-chip microcontroller called the CSE (Converged Security Engine) sets up some SDRAM at ffff8000 and loads the TPL image to that address. The SRAM extends up to the top of 32-bit address space, but the last 2KB is the start16 region, so the TPL image must be 30KB at most, and CONFIG_TPL_TEXT_BASE must be ffff8000. Actually the start16 region is small and it could probably move from f800 to fe00, providing another 1.5KB, but TPL is only about 19KB so there is no need to change it at present. The size limit is enforced by CONFIG_TPL_SIZE_LIMIT to avoid producing images that won't boot.

TPL (running from start.S) first sets up CAR (Cache-as-RAM) which provides larger area of RAM for use while booting. CAR is mapped at CONFIG_SYS_CAR_ADDR (fef00000) and is 768KB in size. It then sets up the stack in the botttom 64KB of this space (i.e. below fef10000). This means that the stack and early malloc() region in TPL can be 64KB at most.

TPL operates without CONFIG_TPL_PCI enabled so PCI config access must use the x86-specific functions pci_x86_write_config(), etc. SPL creates a simple-bus device so that PCI devices are bound by driver model. Then arch_cpu_init_tpl() is called to early init on various devices. This includes placing PCI devices at hard-coded addresses in the memory map. PCI auto-config is not used.

Most of the 16KB ROM is mapped into the very top of memory, except for the Intel descriptor (first 4KB) and the space for SRAM as above.

TPL does not set up a bloblist since at present it does not have anything to pass to SPL.

Once TPL is done it loads SPL from ROM using either the memory-mapped SPI or by using the Intel fast SPI driver. SPL is loaded into CAR, at the address given by CONFIG_SPL_TEXT_BASE, which is normally fef10000.

Note that booting using the SPI driver results in an TPL image that is about 26KB in size instead of 19KB. Also boot speed is worse by about 340ms. If you really want to use the driver, enable CONFIG_APL_SPI_FLASH_BOOT and set BOOT_FROM_FAST_SPI_FLASH to true[2].

Boot flow - SPL

SPL (running from start_from_tpl.S) continues to use the same stack as TPL. It calls arch_cpu_init_spl() to set up a few devices, then init_dram() loads the FSP-M binary into CAR and runs to, to set up SDRAM. The address of the output 'HOB' list (Hand-off-block) is stored into gd->arch.hob_list for parsing. There is a 2GB chunk of SDRAM starting at 0 and the rest is at 4GB.

PCI auto-config is not used in SPL either, but CONFIG_SPL_PCI is defined, so proper PCI access is available and normal dm_pci_read_config() calls can be used. However PCI auto-config is not used so the same static memory mapping set up by TPL is still active.

SPL on x86 always runs with CONFIG_SPL_SEPARATE_BSS=y and BSS is at 120000 (see u-boot-spl.lds). This works because SPL doesn't access BSS until after board_init_r(), as per the rules, and DRAM is available then.

SPL sets up a bloblist and passes the SPL hand-off information to U-Boot proper. This includes a pointer to the HOB list as well as DRAM information. See struct arch_spl_handoff. The bloblist address is set by CONFIG_BLOBLIST_ADDR, normally 100000.

SPL uses SPI flash to update the MRC caches in ROM. This speeds up subsequent boots. Be warned that SPL can take 30 seconds without this cache! This is a known issue with Intel SoCs with modern DRAM and apparently cannot be improved. The MRC caches are used to work around this.

Once SPL is finished it loads U-Boot into SDRAM at CONFIG_SYS_TEXT_BASE, which is normally 1110000. Note that CAR is still active.

Boot flow - U-Boot pre-relocation

U-Boot (running from start_from_spl.S) starts running in RAM and uses the same stack as SPL. It does various init activities before relocation. Notably arch_cpu_init_dm() sets up the pin muxing for the chip using a very large table in the device tree.

PCI auto-config is not used before relocation, but CONFIG_PCI of course is defined, so proper PCI access is available. The same static memory mapping set up by TPL is still active until relocation.

As per usual, U-Boot allocates memory at the top of available RAM (a bit below 2GB in this case) and copies things there ready to relocate itself. Notably reserve_arch() does not reserve space for the HOB list returned by FSP-M since this is already located in RAM.

U-Boot then shuts down CAR and jumps to its relocated version.

Boot flow - U-Boot post-relocation

U-Boot starts up normally, running near the top of RAM. After driver model is running, arch_fsp_init_r() is called which loads and runs the FSP-S binary. This updates the HOB list to include graphics information, used by the fsp_video driver.

PCI autoconfig is done and a few devices are probed to complete init. Most others are started only when they are used.

Note that FSP-S is supposed to run after CAR has been shut down, which happens immediately before U-Boot starts up in its relocated position. Therefore we cannot run FSP-S before relocation. On the other hand we must run it before PCI auto-config is done, since FSP-S may show or hide devices. The first device that probes PCI after relocation is the serial port, in initr_serial(), so FSP-S must run before that. A corollary is that loading FSP-S must be done without using the SPI driver, to avoid probing PCI and causing an autoconfig, so memory-mapped reading is always used for FSP-S.

It would be possible to tear down CAR in SPL instead of U-Boot. The SPL handoff information could make sure it does not include any pointers into CAR (in fact it doesn't). But tearing down CAR in U-Boot allows the initial state used by TPL and SPL to be read by U-Boot, which seems useful. It also matches how older platforms start up (those that don't use SPL).

Performance

Bootstage is used through all phases of U-Boot to keep accurate timimgs for boot. Use 'bootstage report' in U-Boot to see the report, e.g.:

```
Timer summary in microseconds (16 records):
       Mark    Elapsed  Stage
          0          0  reset
    155,325    155,325  TPL
    204,014     48,689  end TPL
    204,385        371  SPL
    738,633    534,248  end SPL
    739,161        528  board_init_f
    842,764    103,603  board_init_r
  1,166,233    323,469  main_loop
  1,166,283         50  id=175

Accumulated time:
                    62  fast_spi
                   202  dm_r
                 7,779  dm_spl
                15,555  dm_f
               208,357  fsp-m
               239,847  fsp-s
               292,143  mmap_spi
```

CPU performance is about 3500 DMIPS:

```
=> dhry
1000000 iterations in 161 ms: 6211180/s, 3535 DMIPS
```

Partial memory map

```
ffffffff       Top of ROM (and last byte of 32-bit address space)
ffff8000       TPL loaded here (from IFWI)
ff000000       Bottom of ROM
fefc0000       Top of CAR region
fef96000       Stack for FSP-M
fef40000 59000 FSP-M
```

(continues on next page)

(continued from previous page)

```
fef11000       SPL loaded here
fef10000       CONFIG_BLOBLIST_ADDR
fef10000       Stack top in TPL, SPL and U-Boot before relocation
fef00000  1000 CONFIG_BOOTSTAGE_STASH_ADDR
fef00000       Base of CAR region

   30000       AP_DEFAULT_BASE (used to start up additional CPUs)
   f0000       CONFIG_ROM_TABLE_ADDR
  120000       BSS (defined in u-boot-spl.lds)
  200000       FSP-S (which is run after U-Boot is relocated)
 1110000       CONFIG_SYS_TEXT_BASE
```

Supported peripherals

- UART
- SPI flash
- Video
- MMC (dev 0) and micro-SD (dev 1)
- Chrome OS EC
- Keyboard
- USB

To do

- **Finish peripherals**
  - left-side USB
  - USB-C
  - Cr50 (security chip: a basic driver is running but not included here)
  - Sound (Intel I2S support exists, but need da7219 driver)
  - Various minor features supported by LPC, etc.
- Booting Chrome OS, e.g. with verified boot
- Integrate with Chrome OS vboot
- Improvements to booting from coreboot (i.e. as a coreboot target)
- Use FSP-T binary instead of our own CAR implementation
- Use the official FSP package instead of the coreboot one
- Enable all CPU cores
- Suspend / resume
- ACPI

Credits

This is a spare-time project conducted slowly over a long period of time.

Much of the code for this port came from Coreboot, an open-source firmware project similar to U-Boot's SPL in terms of features.

Also see [2] for information about the boot flow used by coreboot. It is similar, but has an extra postcar stage. U-Boot doesn't need this since it supports relocating itself in memory.

[2] Intel PDF https://www.coreboot.org/images/2/23/Apollolake_SoC.pdf

Chromebook Link

First, you need the following binary blobs:

- descriptor.bin - Intel flash descriptor
- me.bin - Intel Management Engine
- mrc.bin - Memory Reference Code, which sets up SDRAM
- video ROM - sets up the display

You can get these binary blobs by:

```
$ git clone http://review.coreboot.org/p/blobs.git
$ cd blobs
```

Find the following files:

- ./mainboard/google/link/descriptor.bin
- ./mainboard/google/link/me.bin
- ./northbridge/intel/sandybridge/systemagent-r6.bin

The 3rd one should be renamed to mrc.bin. As for the video ROM, you can get it here and rename it to vga.bin. Make sure all these binary blobs are put in the board directory.

Now you can build U-Boot and obtain u-boot.rom:

```
$ make chromebook_link_defconfig
$ make all
```

Chromebook Samus

First, you need the following binary blobs:

- descriptor.bin - Intel flash descriptor
- me.bin - Intel Management Engine
- mrc.bin - Memory Reference Code, which sets up SDRAM
- refcode.elf - Additional Reference code
- vga.bin - video ROM, which sets up the display

If you have a samus you can obtain them from your flash, for example, in developer mode on the Chromebook (use Ctrl-Alt-F2 to obtain a terminal and log in as 'root'):

```
cd /tmp
flashrom -w samus.bin
scp samus.bin username@ip_address:/path/to/somewhere
```

If not see the coreboot tree where you can use:

```
bash crosfirmware.sh samus
```

to get the image. There is also an 'extract_blobs.sh' scripts that you can use on the 'coreboot-Google_Samus.*' file to short-circuit some of the below.

Then 'ifdtool -x samus.bin' on your development machine will produce:

```
flashregion_0_flashdescriptor.bin
flashregion_1_bios.bin
flashregion_2_intel_me.bin
```

Rename flashregion_0_flashdescriptor.bin to descriptor.bin Rename flashregion_2_intel_me.bin to me.bin You can ignore flashregion_1_bios.bin - it is not used.

To get the rest, use 'cbfstool samus.bin print':

```
samus.bin: 8192 kB, bootblocksize 2864, romsize 8388608, offset 0x700000
alignment: 64 bytes, architecture: x86
```

Name	Offset	Type	Size
cmos_layout.bin	0x700000	cmos_layout	1164
pci8086,0406.rom	0x7004c0	optionrom	65536
spd.bin	0x710500	(unknown)	4096
cpu_microcode_blob.bin	0x711540	microcode	70720
fallback/romstage	0x722a00	stage	54210
fallback/ramstage	0x72fe00	stage	96382
config	0x7476c0	raw	6075
fallback/vboot	0x748ec0	stage	15980
fallback/refcode	0x74cd80	stage	75578
fallback/payload	0x75f500	payload	62878
u-boot.dtb	0x76eb00	(unknown)	5318
(empty)	0x770000	null	196504
mrc.bin	0x79ffc0	(unknown)	222876
(empty)	0x7d66c0	null	167320

You can extract what you need:

```
cbfstool samus.bin extract -n pci8086,0406.rom -f vga.bin
cbfstool samus.bin extract -n fallback/refcode -f refcode.rmod
cbfstool samus.bin extract -n mrc.bin -f mrc.bin
cbfstool samus.bin extract -n fallback/refcode -f refcode.bin -U
```

Note that the -U flag is only supported by the latest cbfstool. It unpacks and decompresses the stage to produce a coreboot rmodule. This is a simple representation of an ELF file. You need the patch "Support decoding a stage with compression".

Put all 5 files into board/google/chromebook_samus.

Now you can build U-Boot and obtain u-boot.rom:

```
$ make chromebook_samus_defconfig
$ make all
```

If you are using em100, then this command will flash write -Boot:

```
em100 -s -d filename.rom -c W25Q64CV -r
```

Flash map for samus / broadwell:

fffff800 SYS_X86_START16

ffff0000 RESET_SEG_START

fffd8000 TPL_TEXT_BASE

fffa0000 X86_MRC_ADDR

fff90000 VGA_BIOS_ADDR

ffed0000 SYS_TEXT_BASE

ffea0000 X86_REFCODE_ADDR

ffe70000 SPL_TEXT_BASE

ffbf8000 CONFIG_ENV_OFFSET (environemnt offset)

ffbe0000 rw-mrc-cache (Memory-reference-code cache)

ffa00000 <spare>

ff801000 intel-me (address set by descriptor.bin)

ff800000 intel-descriptor

7.1.9 Intel

Bayley Bay CRB

This uses as FSP as with Crown Bay, except it is for the Atom E3800 series. Download this and get the .fd file (BAYTRAIL_FSP_GOLD_003_16-SEP-2014.fd at the time of writing). Put it in the corresponding board directory and rename it to fsp.bin.

Obtain the VGA RAM (Vga.dat at the time of writing) and put it into the same board directory as vga.bin.

You still need two more binary blobs. For Bayley Bay, they can be extracted from the sample SPI image provided in the FSP (SPI.bin at the time of writing):

```
$ ./tools/ifdtool -x BayleyBay/SPI.bin
$ cp flashregion_0_flashdescriptor.bin board/intel/bayleybay/descriptor.bin
$ cp flashregion_2_intel_me.bin board/intel/bayleybay/me.bin
```

Now you can build U-Boot and obtain u-boot.rom:

```
$ make bayleybay_defconfig
$ make all
```

Note that the debug version of the FSP is bigger in size. If this version is used, CONFIG_FSP_ADDR needs to be configured to 0xfffb0000 instead of the default value 0xfffc0000.

Cherry Hill CRB

This uses Intel FSP for Braswell platform. Download it from Intel FSP website, put the .fd file to the board directory and rename it to fsp.bin.

Extract descriptor.bin and me.bin from the original BIOS on the board using ifdtool and put them to the board directory as well.

Note the FSP package for Braswell does not ship a traditional legacy VGA BIOS image for the integrated graphics device. Instead a new binary called Video BIOS Table (VBT) is shipped. Put it to the board directory and rename it to vbt.bin if you want graphics support in U-Boot.

Now you can build U-Boot and obtain u-boot.rom:

```
$ make cherryhill_defconfig
$ make all
```

An important note for programming u-boot.rom to the on-board SPI flash is that you need make sure the SPI flash's 'quad enable' bit in its status register matches the settings in the descriptor.bin, otherwise the board won't boot.

For the on-board SPI flash MX25U6435F, this can be done by writing 0x40 to the status register by DediProg in: Config > Modify Status Register > Write Status Register(s) > Register1 Value(Hex). This is is a one-time change. Once set, it persists in SPI flash part regardless of the u-boot.rom image burned.

Cougar Canyon 2 CRB

This uses Intel FSP for 3rd generation Intel Core and Intel Celeron processors with mobile Intel HM76 and QM77 chipsets platform. Download it from Intel FSP website and put the .fd file (CHIEFRIVER_FSP_GOLD_001_09-OCTOBER-2013.fd at the time of writing) in the board directory and rename it to fsp.bin.

Now build U-Boot and obtain u-boot.rom:

```
$ make cougarcanyon2_defconfig
$ make all
```

The board has two 8MB SPI flashes mounted, which are called SPI-0 and SPI-1 in the board manual. The SPI-0 flash should have flash descriptor plus ME firmware and SPI-1 flash is used to store U-Boot. For convenience, the complete 8MB SPI-0 flash image is included in the FSP package (named Rom00_8M_MB_PPT.bin). Program this image to the SPI-0 flash according to the board manual just once and we are all set. For programming U-Boot we just need to program SPI-1 flash. Since the default u-boot.rom image for this board is set to 2MB, it should be programmed to the last 2MB of the 8MB chip, address range [600000, 7FFFFF].

Crown Bay CRB

U-Boot support of Intel Crown Bay board relies on a binary blob called Firmware Support Package (FSP) to perform all the necessary initialization steps as documented in the BIOS Writer Guide, including initialization of the CPU, memory controller, chipset and certain bus interfaces.

Download the Intel FSP for Atom E6xx series and Platform Controller Hub EG20T, install it on your host and locate the FSP binary blob. Note this platform also requires a Chipset Micro Code (CMC) state machine binary to be present in the SPI flash where u-boot.rom resides, and this CMC binary blob can be found in this FSP package too.

- ./FSP/QUEENSBAY_FSP_GOLD_001_20-DECEMBER-2013.fd
- ./Microcode/C0_22211.BIN

Rename the first one to fsp.bin and second one to cmc.bin and put them in the board directory.

Note the FSP release version 001 has a bug which could cause random endless loop during the FspInit call. This bug was published by Intel although Intel did not describe any details. We need manually apply the patch to the FSP binary using any hex editor (eg: bvi). Go to the offset 0x1fcd8 of the FSP binary, change the following five bytes values from orginally E8 42 FF FF FF to B8 00 80 0B 00.

As for the video ROM, you need manually extract it from the Intel provided BIOS for Crown Bay here, using the AMI MMTool. Check PCI option ROM ID 8086:4108, extract and save it as vga.bin in the board directory.

Now you can build U-Boot and obtain u-boot.rom:

```
$ make crownbay_defconfig
$ make all
```

Edison

Build Instructions for U-Boot as main bootloader

Simple you can build U-Boot and obtain u-boot.bin:

```
$ make edison_defconfig
$ make all
```

Updating U-Boot on Edison

By default Intel Edison boards are shipped with preinstalled heavily patched U-Boot v2014.04. Though it supports DFU which we may be able to use.

1. Prepare u-boot.bin as described in chapter above. You still need one more step (if and only if you have original U-Boot), i.e. run the following command:

   ```
   $ truncate -s %4096 u-boot.bin
   ```

2. Run your board and interrupt booting to U-Boot console. In the console call:

   ```
   => run do_force_flash_os
   ```

3. Wait for few seconds, it will prepare environment variable and runs DFU. Run DFU command from the host system:

   ```
   $ dfu-util -v -d 8087:0a99 --alt u-boot0 -D u-boot.bin
   ```

4. Return to U-Boot console and following hint. i.e. push Ctrl+C, and reset the board:

   ```
   => reset
   ```

Updating U-Boot using xFSTK

You can also update U-Boot using the xfstk-dldr-solo tool if you can build it. One way to do that is to follow the xFSTK instructions. You may need to use a virtual machine running Ubuntu Trusty. Once you have built it and installed libboost-all-dev, you can copy xfstk-dldr-solo to /usr/local/bin and libboost_program_options.so.1.54.0 to /usr/lib/i386-linux-gnu/ and with luck it will work. You might fine this drive helpful.

If it does, then you can download and unpack the Edison reocovery image, install dfu-util, reset your board and flash U-Boot like this:

```
$ xfstk-dldr-solo --gpflags 0x80000007 \
   --osimage u-boot-edison.img \
   --fwdnx recover/edison_dnx_fwr.bin \
   --fwimage recover/edison_ifwi-dbg-00.bin \
   --osdnx recover/edison_dnx_osr.bin
```

This should show the following

```
XFSTK Downloader Solo 0.0.0
Copyright (c) 2015 Intel Corporation
Build date and time: Aug 15 2020 15:07:13

.Intel SoC Device Detection Found
```

(continues on next page)

(continued from previous page)

```
Parsing Commandline....
Registering Status Callback....
.Initiating Download Process....
.......(lots of dots)........XFSTK-STATUS--Reconnecting to device - Attempt #1
.......(even more dots)......................
```

You have about 10 seconds after resetting the board to type the above command. If you want to check if the board is ready, type:

```
lsusb |egrep "8087|8086"
Bus 001 Device 004: ID 8086:e005 Intel Corp.
```

If you see a device with the same ID as above, the board is waiting for your command.

After about 5 seconds you should see some console output from the board:

```
******************************
PSH KERNEL VERSION: b0182b2b
              WR: 20104000
******************************

SCU IPC: 0x800000d0  0xfffce92c

PSH miaHOB version: TNG.B0.VVBD.0000000c

microkernel built 11:24:08 Feb  5 2015

******* PSH loader *******
PCM page cache size = 192 KB
Cache Constraint = 0 Pages
Arming IPC driver ..
Adding page store pool ..
PagestoreAddr(IMR Start Address) = 0x04899000
pageStoreSize(IMR Size)          = 0x00080000

*** Ready to receive application ***

After another 10 seconds the xFSTK tool completes and the board resets. About
10 seconds after that should see the above message again and then within a
few seconds U-Boot should start on your board:
```

```
U-Boot 2020.10-rc3 (Sep 03 2020 - 18:44:28 -0600)

CPU:   Genuine Intel(R) CPU   4000  @  500MHz
DRAM:  980.6 MiB
WDT:   Started with servicing (60s timeout)
MMC:   mmc@ff3fc000: 0, mmc@ff3fa000: 1
Loading Environment from MMC... OK
In:    serial
Out:   serial
Err:   serial
Saving Environment to MMC... Writing to redundant MMC(0)... OK
Saving Environment to MMC... Writing to MMC(0)... OK
Net:   No ethernet found.
Hit any key to stop autoboot:  0
Target:blank
Partitioning using GPT
```

(continues on next page)

(continued from previous page)

```
Writing GPT: success!
Saving Environment to MMC... Writing to redundant MMC(0)... OK
Flashing already done...
5442816 bytes read in 238 ms (21.8 MiB/s)
Valid Boot Flag
Setup Size = 0x00003c00
Magic signature found
Using boot protocol version 2.0c
Linux kernel version 3.10.17-poky-edison+ (ferry@kalamata) #1 SMP PREEMPT Mon Jan 11␣
↪14:54:18 CET 2016
Building boot_params at 0x00090000
Loading bzImage at address 100000 (5427456 bytes)
Magic signature found
Kernel command line: "rootwait root=PARTUUID=ada722ed-6410-764e-8619-abff6f66e10e␣
↪rootfstype=ext4 console=ttyMFD2 earlyprintk=ttyMFD2,keep loglevel=4 g_multi.ethernet_
↪config=cdc systemd.unit=multi-user.target hardware_id=00 g_multi.
↪iSerialNumber=2249baf774c675598661a63098c0ad41 g_multi.dev_addr=02:00:86:c0:ad:41␣
↪platform_mrfld_audio.audio_codec=dummy"
Magic signature found

Starting kernel ...

...

Poky (Yocto Project Reference Distro) 1.7.2 edison ttyMFD2

edison login:
```

Galileo

Only one binary blob is needed for Remote Management Unit (RMU) within Intel Quark SoC. Not like FSP, U-Boot does not call into the binary. The binary is needed by the Quark SoC itself.

You can get the binary blob from Quark Board Support Package from Intel website:

- ./QuarkSocPkg/QuarkNorthCluster/Binary/QuarkMicrocode/RMU.bin

Rename the file and put it to the board directory by:

```
$ cp RMU.bin board/intel/galileo/rmu.bin
```

Now you can build U-Boot and obtain u-boot.rom:

```
$ make galileo_defconfig
$ make all
```

Minnowboard MAX

This uses as FSP as with Crown Bay, except it is for the Atom E3800 series. Download this and get the .fd file (BAYTRAIL_FSP_GOLD_003_16-SEP-2014.fd at the time of writing). Put it in the corresponding board directory and rename it to fsp.bin.

Obtain the VGA RAM (Vga.dat at the time of writing) and put it into the same board directory as vga.bin.

You still need two more binary blobs. For Minnowboard MAX, we can reuse the same ME firmware above, but for flash descriptor, we need get that somewhere else, as the one above does not seem to work, probably because it is not designed for the Minnowboard MAX. Now download the original firmware image for this board from:

- http://firmware.intel.com/sites/default/files/2014-WW42.4-MinnowBoardMax.73-64-bit.bin_Release.zip

Unzip it:

```
$ unzip 2014-WW42.4-MinnowBoardMax.73-64-bit.bin_Release.zip
```

Use ifdtool in the U-Boot tools directory to extract the images from that file, for example:

```
$ ./tools/ifdtool -x MNW2MAX1.X64.0073.R02.1409160934.bin
```

This will provide the descriptor file - copy this into the correct place:

```
$ cp flashregion_0_flashdescriptor.bin board/intel/minnowmax/descriptor.bin
```

Now you can build U-Boot and obtain u-boot.rom:

```
$ make minnowmax_defconfig
$ make all
```

Checksums are as follows (but note that newer versions will invalidate this):

```
$ md5sum -b board/intel/minnowmax/*.bin
ffda9a3b94df5b74323afb328d51e6b4  board/intel/minnowmax/descriptor.bin
69f65b9a580246291d20d08cbef9d7c5  board/intel/minnowmax/fsp.bin
894a97d371544ec21de9c3e8e1716c4b  board/intel/minnowmax/me.bin
a2588537da387da592a27219d56e9962  board/intel/minnowmax/vga.bin
```

The ROM image is broken up into these parts:

Offset	Description	Controlling config
000000	descriptor.bin	Hard-coded to 0 in ifdtool
001000	me.bin	Set by the descriptor
500000	<spare>	
6ef000	Environment	CONFIG_ENV_OFFSET
6f0000	MRC cache	CONFIG_ENABLE_MRC_CACHE
700000	u-boot-dtb.bin	CONFIG_SYS_TEXT_BASE
7b0000	vga.bin	CONFIG_VGA_BIOS_ADDR
7c0000	fsp.bin	CONFIG_FSP_ADDR
7f8000	<spare>	(depends on size of fsp.bin)
7ff800	U-Boot 16-bit boot	CONFIG_SYS_X86_START16

Overall ROM image size is controlled by CONFIG_ROM_SIZE.

Note that the debug version of the FSP is bigger in size. If this version is used, CONFIG_FSP_ADDR needs to be configured to 0xfffb0000 instead of the default value 0xfffc0000.

Slim Bootloader

Introduction

This target is to enable U-Boot as a payload of Slim Bootloader (a.k.a SBL) boot firmware which currently supports QEMU, Apollolake, Whiskeylake, Coffeelake-R platforms.

The Slim Bootloader is designed with multi-stages (Stage1A/B, Stage2, Payload) architecture to cover from reset vector to OS booting and it consumes Intel FSP for silicon initialization.

- Stage1A: Reset vector, CAR init with FSP-T
- Stage1B: Memory init with FSP-M, CAR teardown, Continue execution in memory
- Stage2 : Rest of Silicon init with FSP-S, Create HOB, Hand-off to Payload
- Payload: Payload init with HOB, Load OS from media, Booting OS

The Slim Bootloader stages (Stage1A/B, Stage2) focus on chipset, hardware and platform specific initialization, and it provides useful information to a payload in a HOB (Hand-Off Block) which has serial port, memory map, performance data info and so on. This is Slim Bootloader architectural design to make a payload light-weight, platform independent and more generic across different boot solutions or payloads, and to minimize hardware re-initialization in a payload.

Build Instruction for U-Boot as a Slim Bootloader payload

Build U-Boot and obtain u-boot-dtb.bin:

```
$ make distclean
$ make slimbootloader_defconfig
$ make all
```

Prepare Slim Bootloader

1. Setup Build Environment for Slim Bootloader.

   Refer to Getting Started page in Slim Bootloader document site.

2. Get source code. Let's simply clone the repo:

   ```
   $ git clone https://github.com/slimbootloader/slimbootloader.git
   ```

3. Copy u-boot-dtb.bin to Slim Bootloader. Slim Bootloader looks for a payload from the specific location. Copy the build u-boot-dtb.bin to the expected location:

   ```
   $ mkdir -p <Slim Bootloader Dir>/PayloadPkg/PayloadBins/
   $ cp <U-Boot Dir>/u-boot-dtb.bin <Slim Bootloader Dir>/PayloadPkg/PayloadBins/u-
   ↪boot-dtb.bin
   ```

Build Instruction for Slim Bootloader for QEMU target

Slim Bootloader supports multiple payloads, and a board of Slim Bootloader detects its target payload by PayloadId in board configuration. The PayloadId can be any 4 Bytes value.

1. Update PayloadId. Let's use 'U-BT' as an example:

   ```
   $ vi Platform/QemuBoardPkg/CfgData/CfgDataExt_Brd1.dlt
   -GEN_CFG_DATA.PayloadId                    | 'AUTO'
   +GEN_CFG_DATA.PayloadId                    | 'U-BT'
   ```

2. Update payload text base. PAYLOAD_EXE_BASE must be the same as U-Boot CONFIG_SYS_TEXT_BASE in board/intel/slimbootloader/Kconfig. PAYLOAD_LOAD_HIGH must be 0:

   ```
   $ vi Platform/QemuBoardPkg/BoardConfig.py
   +               self.PAYLOAD_LOAD_HIGH    = 0
   +               self.PAYLOAD_EXE_BASE     = 0x00100000
   ```

3. Build QEMU target. Make sure u-boot-dtb.bin and U-BT PayloadId in build command. The output is Outputs/qemu/SlimBootloader.bin:

```
$ python BuildLoader.py build qemu -p "OsLoader.efi:LLDR:Lz4;u-boot-dtb.bin:U-
↪BT:Lzma"
```

4. Launch Slim Bootloader on QEMU. You should reach at U-Boot serial console:

```
$ qemu-system-x86_64 -machine q35 -nographic -serial mon:stdio -pflash Outputs/
↪qemu/SlimBootloader.bin
```

Test Linux booting on QEMU target

Let's use LeafHill (APL) Yocto image for testing. Download it from http://downloads.yoctoproject.org/releases/yocto/yocto-2.0/machines/leafhill/.

1. Prepare Yocto hard disk image:

```
$ wget http://downloads.yoctoproject.org/releases/yocto/yocto-2.0/machines/
↪leafhill/leafhill-4.0-jethro-2.0.tar.bz2
$ tar -xvf leafhill-4.0-jethro-2.0.tar.bz2
$ ls -l leafhill-4.0-jethro-2.0/binary/core-image-sato-intel-corei7-64.hddimg
```

2. Launch Slim Bootloader on QEMU with disk image:

```
$ qemu-system-x86_64 -machine q35 -nographic -serial mon:stdio -pflash Outputs/
↪qemu/SlimBootloader.bin -drive id=mydrive,if=none,file=/path/to/core-image-sato-
↪intel-corei7-64.hddimg,format=raw -device ide-hd,drive=mydrive
```

3. Update boot environment values on shell:

```
=> setenv bootfile vmlinuz
=> setenv bootdev scsi
=> boot
```

Build Instruction for Slim Bootloader for LeafHill (APL) target

Prepare U-Boot and Slim Bootloader as described at the beginning of this page. Also, the PayloadId needs to be set for APL board.

1. Update PayloadId. Let's use 'U-BT' as an example:

```
$ vi Platform/ApollolakeBoardPkg/CfgData/CfgData_Int_LeafHill.dlt
-GEN_CFG_DATA.PayloadId                     | 'AUTO
+GEN_CFG_DATA.PayloadId                     | 'U-BT'
```

2. Update payload text base.

- PAYLOAD_EXE_BASE must be the same as U-Boot CONFIG_SYS_TEXT_BASE in board/intel/slimbootloader/Kconfig.
- PAYLOAD_LOAD_HIGH must be 0:

```
$ vi Platform/ApollolakeBoardPkg/BoardConfig.py
+               self.PAYLOAD_LOAD_HIGH    = 0
+               self.PAYLOAD_EXE_BASE     = 0x00100000
```

3. Build APL target. Make sure u-boot-dtb.bin and U-BT PayloadId in build command. The output is Outputs/apl/Stitch_Components.zip:

```
$ python BuildLoader.py build apl -p "OsLoader.efi:LLDR:Lz4;u-boot-dtb.bin:U-
↪BT:Lzma"
```

4. Stitch IFWI.

   Refer to Apollolake page in Slim Bootloader document site:

```
$ python Platform/ApollolakeBoardPkg/Script/StitchLoader.py -i <Existing IFWI> -s␣
↪Outputs/apl/Stitch_Components.zip -o <Output IFWI>
```

5. Flash IFWI.

   Use DediProg to flash IFWI. You should reach at U-Boot serial console.

Build Instruction to use ELF U-Boot

1. Enable CONFIG_OF_EMBED:

```
$ vi configs/slimbootloader_defconfig
+CONFIG_OF_EMBED=y
```

2. Build U-Boot:

```
$ make distclean
$ make slimbootloader_defconfig
$ make all
$ strip u-boot (removing symbol for reduced size)
```

3. Do same steps as above

- Copy u-boot (ELF) to PayloadBins directory
- Update PayloadId 'U-BT' as above.
- No need to set PAYLOAD_LOAD_HIGH and PAYLOAD_EXE_BASE.
- Build Slim Bootloader. Use u-boot instead of u-boot-dtb.bin:

```
$ python BuildLoader.py build <qemu or apl> -p "OsLoader.efi:LLDR:Lz4;u-boot:U-
↪BT:Lzma"
```

7.1.10 Kontron

Summary

The Kontron SMARC-sAL28 board is a TSN-enabled dual-core ARM A72 processor module with an on-chip 6-port TSN switch and a 3D GPU.

Quickstart

Compile U-Boot

Configure and compile the binary:

```
$ make kontron_sl28_defconfig
$ CROSS_COMPILE=aarch64-linux-gnu make
```

Copy u-boot.rom to a TFTP server.

Install the bootloader on the board

Please note, this bootloader doesn't support the builtin watchdog (yet), therefore you have to disable it, see below. Otherwise you'll end up in the failsafe bootloader on every reset:

```
> tftp path/to/u-boot.rom
> sf probe 0
> sf update $fileaddr 0x210000 $filesize
```

The board is fully failsafe, you can't break anything. But because you've disabled the builtin watchdog you might have to manually enter failsafe mode by asserting the `FORCE_RECOV#` line during board reset.

Disable the builtin watchdog

- boot into the failsafe bootloader, either by asserting the `FORCE_RECOV#` line or if you still have the original bootloader installed you can use the command:

  ```
  > wdt dev cpld_watchdog@4a; wdt expire 1
  ```

- in the failsafe bootloader use the "sl28 nvm" command to disable the automatic start of the builtin watchdog:

  ```
  > sl28 nvm 0008
  ```

- power-cycle the board

Useful I2C tricks

The board has a board management controller which is not supported in u-boot (yet). But you can use the i2c command to access it.

- reset into failsafe bootloader:

  ```
  > i2c mw 4a 5.1 0; i2c mw 4a 6.1 6b; i2c mw 4a 4.1 42
  ```

- read board management controller version:

  ```
  > i2c md 4a 3.1 1
  ```

Non-volatile Board Configuration Bits

The board has 16 configuration bits which are stored in the CPLD and are non-volatile. These can be changed by the *sl28 nvm* command.

Bit	Description
0	Power-on inhibit
1	Enable eMMC boot
2	Enable watchdog by default
3	Disable failsafe watchdog by default
4	Clock generator selection bit 0
5	Clock generator selection bit 1
6	Disable CPU SerDes clock #2 and PCIe-A clock output
7	Disable PCIe-B and PCIe-C clock output
8	Keep onboard PHYs in reset
9	Keep USB hub in reset
10	Keep eDP-to-LVDS converter in reset
11	Enable I2C stuck recovery on I2C PM and I2C GP busses
12	Enable automatic onboard PHY H/W reset
13	reserved
14	Used by the RCW to determine boot source
15	Used by the RCW to determine boot source

Please note, that if the board is in failsafe mode, the bits will have the factory defaults, ie. all bits are off.

Power-On Inhibit

If this is set, the board doesn't automatically turn on when power is applied. Instead, the user has to either toggle the `PWR_BTN#` line or use any other wake-up source such as RTC alarm or Wake-on-LAN.

eMMC Boot

If this is set, the RCW will be fetched from the on-board eMMC at offset 1MiB. For further details, have a look at the Reset Configuration Word Documentation.

Watchdog

By default, the CPLD watchdog is enabled in failsafe mode. Using bits 2 and 3, the user can change its mode or disable it altogether.

Bit 2	Bit 3	Description
0	0	Watchdog enabled, failsafe mode
0	1	Watchdog disabled
1	0	Watchdog enabled, failsafe mode
1	1	Watchdog enabled, normal mode

Clock Generator Select

The board is prepared to supply different SerDes clock speeds. But for now, only setting 0 is supported, otherwise the CPU will hang because the PLL will not lock.

Clock Output Disable And Keep Devices In Reset

To safe power, the user might disable different devices and clock output of the board. It is not supported to disable the "CPU SerDes clock #2" for now, otherwise the CPU will hang because the PLL will not lock.

Automatic reset of the onboard PHYs

By default, there is no hardware reset of the onboard PHY. This is because for Wake-on-LAN, some registers have to retain their values. If you don't use the WOL feature and a soft reset of the PHY is not enough you can enable the hardware reset. The onboard PHY hardware reset follows the power-on reset.

Further documentation

- Vendor Documentation
- Reset Configuration Word Documentation

7.1.11 Renesas

R0P7752C00000RZ board

This board specification

The R0P7752C00000RZ(board config name:sh7752evb) has the following device:

- SH7752 (SH-4A)
- DDR3-SDRAM 512MB
- SPI ROM 8MB
- Gigabit Ethernet controllers
- eMMC 4GB

Configuration for This board

You can select the configuration as follows:

- make sh7752evb_config

This board specific command

This board has the following its specific command:

write_mac: You can write MAC address to SPI ROM.

Usage 1: Write MAC address

```
write_mac [GETHERC ch0] [GETHERC ch1]

For example:
=> write_mac 74:90:50:00:33:9e 74:90:50:00:33:9f
```

- We have to input the command as a single line (without carriage return)
- We have to reset after input the command.

Usage 2: Show current data

```
write_mac

For example:
=> write_mac
```

(continues on next page)

(continued from previous page)

```
   GETHERC ch0 = 74:90:50:00:33:9e
   GETHERC ch1 = 74:90:50:00:33:9f
```

Update SPI ROM

1. Copy u-boot image to RAM area.
2. Probe SPI device.

```
=> sf probe 0
SF: Detected MX25L6405D with page size 64KiB, total 8 MiB
```

3. Erase SPI ROM.

```
=> sf erase 0 80000
```

4. Write u-boot image to SPI ROM.

```
=> sf write 0x48000000 0 80000
```

SH7753 EVB board

This board specification

The SH7753 EVB (board config name:sh7753evb) has the following device:

- SH7753 (SH-4A)
- DDR3-SDRAM 512MB
- SPI ROM 8MB
- Gigabit Ethernet controllers
- eMMC 4GB

Configuration for This board

You can select the configuration as follows:

- make sh7753evb_config

This board specific command

This board has the following its specific command:

write_mac: You can write MAC address to SPI ROM.

Usage 1: Write MAC address

```
write_mac [GETHERC ch0] [GETHERC ch1]

For example:
=> write_mac 74:90:50:00:33:9e 74:90:50:00:33:9f
```

- We have to input the command as a single line (without carriage return)
- We have to reset after input the command.

Usage 2: Show current data

```
write_mac

For example:
=> write_mac
   GETHERC ch0 = 74:90:50:00:33:9e
   GETHERC ch1 = 74:90:50:00:33:9f
```

Update SPI ROM

1. Copy u-boot image to RAM area.
2. Probe SPI device.

```
=> sf probe 0
SF: Detected MX25L6405D with page size 64KiB, total 8 MiB
```

3. Erase SPI ROM.

```
=> sf erase 0 80000
```

4. Write u-boot image to SPI ROM.

```
=> sf write 0x48000000 0 80000
```

7.1.12 Rockchip

ROCKCHIP

About this

This document describes the information about Rockchip supported boards and it's usage steps.

Rockchip boards

Rockchip is SoC solutions provider for tablets & PCs, streaming media TV boxes, AI audio & vision, IoT hardware.

A wide range of Rockchip SoCs with associated boardsare supported in mainline U-Boot.

List of mainline supported rockchip boards:

- **rk3036**
  - Rockchip Evb-RK3036 (evb-rk3036)
  - Kylin (kylin_rk3036)
- **rk3128**
  - Rockchip Evb-RK3128 (evb-rk3128)
- **rk3229**
  - Rockchip Evb-RK3229 (evb-rk3229)
- **rk3288**
  - Rockchip Evb-RK3288 (evb-rk3288)

  - Firefly-RK3288 (firefly-rk3288)
  - MQmaker MiQi (miqi-rk3288)
  - Phytec RK3288 PCM-947 (phycore-rk3288)
  - PopMetal-RK3288 (popmetal-rk3288)
  - Radxa Rock 2 Square (rock2)
  - Tinker-RK3288 (tinker-rk3288)
  - Google Jerry (chromebook_jerry)
  - Google Mickey (chromebook_mickey)
  - Google Minnie (chromebook_minnie)
  - Google Speedy (chromebook_speedy)
  - Amarula Vyasa-RK3288 (vyasa-rk3288)
- **rk3308**
  - Rockchip Evb-RK3308 (evb-rk3308)
  - Roc-cc-RK3308 (roc-cc-rk3308)
- **rk3328**
  - Rockchip Evb-RK3328 (evb-rk3328)
  - Pine64 Rock64 (rock64-rk3328)
  - Firefly-RK3328 (roc-cc-rk3328)
  - Radxa Rockpi E (rock-pi-e-rk3328)
- **rk3368**
  - GeekBox (geekbox)
  - PX5 EVB (evb-px5)
  - Rockchip Sheep (sheep-rk3368)
  - Theobroma Systems RK3368-uQ7 SoM - Lion (lion-rk3368)
- **rk3399**
  - 96boards RK3399 Ficus (ficus-rk3399)
  - 96boards Rock960 (rock960-rk3399)
  - Firefly-RK3399 (firefly_rk3399)
  - Firefly ROC-RK3399-PC
  - FriendlyElec NanoPC-T4 (nanopc-t4-rk3399)
  - FriendlyElec NanoPi M4 (nanopi-m4-rk3399)
  - FriendlyARM NanoPi NEO4 (nanopi-neo4-rk3399)
  - Google Bob (chromebook_bob)
  - Khadas Edge (khadas-edge-rk3399)
  - Khadas Edge-Captain (khadas-edge-captain-rk3399)
  - Khadas Edge-V (hadas-edge-v-rk3399)
  - Orange Pi RK3399 (orangepi-rk3399)
  - Pine64 RockPro64 (rockpro64-rk3399)
  - Radxa ROCK Pi 4 (rock-pi-4-rk3399)

  - Rockchip Evb-RK3399 (evb_rk3399)
  - Theobroma Systems RK3399-Q7 SoM - Puma (puma_rk3399)
- **rv1108**
  - Rockchip Evb-rv1108 (evb-rv1108)
  - Elgin-R1 (elgin-rv1108)
- **rv3188**
  - Radxa Rock (rock)

Building

TF-A

TF-A would require to build for ARM64 Rockchip SoCs platforms.

To build TF-A:

```
git clone https://github.com/ARM-software/arm-trusted-firmware.git
cd arm-trusted-firmware
make realclean
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=rk3399
```

Specify the PLAT= with desired rockchip platform to build TF-A for.

U-Boot

To build rk3328 boards:

```
export BL31=/path/to/arm-trusted-firmware/to/bl31.elf
make evb-rk3328_defconfig
make
```

To build rk3288 boards:

```
make evb-rk3288_defconfig
make
```

To build rk3368 boards:

```
export BL31=/path/to/arm-trusted-firmware/to/bl31.elf
make evb-px5_defconfig
make
```

To build rk3399 boards:

```
export BL31=/path/to/arm-trusted-firmware/to/bl31.elf
make evb-rk3399_defconfig
make
```

Flashing

1. Package the image with U-Boot TPL/SPL

SD Card

All rockchip platforms, except rk3128 (which doesn't use SPL) are now supporting single boot image using binman and pad_cat.

To write an image that boots from an SD card (assumed to be /dev/sda):

```
sudo dd if=u-boot-rockchip.bin of=/dev/sda seek=64
sync
```

eMMC

eMMC flash would probe on mmc0 in most of the rockchip platforms.

Create GPT partition layout as defined in configurations:

```
mmc dev 0
gpt write mmc 0 $partitions
```

Connect the USB-OTG cable between host and target device.

Launch fastboot at target:

```
fastboot 0
```

Upon successful gadget connection,host show the USB device like:

```
lsusb
Bus 001 Device 020: ID 2207:330c Fuzhou Rockchip Electronics Company RK3399 in Mask␣
↪ROM mode
```

Program the flash:

```
sudo fastboot -i 0x2207 flash loader1 idbloader.img
sudo fastboot -i 0x2207 flash loader2 u-boot.itb
```

Note: for rockchip 32-bit platforms the U-Boot proper image is u-boot-dtb.img

SPI

Generating idbloader for SPI boot would require to input a multi image image format to mkimage tool instead of concerting (like for MMC boot).

SPL-alone SPI boot image:

```
./tools/mkimage -n rk3399 -T rkspi -d spl/u-boot-spl.bin idbloader.img
```

TPL+SPL SPI boot image:

```
./tools/mkimage -n rk3399 -T rkspi -d tpl/u-boot-tpl.bin:spl/u-boot-spl.bin idbloader.
↪img
```

Copy SPI boot images into SD card and boot from SD:

```
sf probe
load mmc 1:1 $kernel_addr_r idbloader.img
sf erase 0 +$filesize
sf write $kernel_addr_r 0 ${filesize}
```

(continues on next page)

(continued from previous page)

```
load mmc 1:1 ${kernel_addr_r} u-boot.itb
sf erase 0x60000 +$filesize
sf write $kernel_addr_r 0x60000 ${filesize}
```

2. Package the image with Rockchip miniloader

Image package with Rockchip miniloader requires robin [1].

Create idbloader.img

```
cd u-boot
./tools/mkimage -n px30 -T rksd -d rkbin/bin/rk33/px30_ddr_333MHz_v1.15.bin idbloader.
↪img
cat rkbin/bin/rk33/px30_miniloader_v1.22.bin >> idbloader.img
sudo dd if=idbloader.img of=/dev/sda seek=64
```

Create trust.img

```
cd rkbin
./tools/trust_merger RKTRUST/PX30TRUST.ini
sudo dd if=trust.img of=/dev/sda seek=24576
```

Create uboot.img

```
rbink/tools/loaderimage --pack --uboot u-boot-dtb.bin uboot.img 0x200000
sudo dd if=uboot.img of=/dev/sda seek=16384
```

Note: 1. 0x200000 is load address and it's an optional in some platforms. 2. rkbin binaries are kept on updating, so would recommend to use the latest versions.

TODO

- Add rockchip idbloader image building
- Add rockchip TPL image building
- Document SPI flash boot
- Add missing SoC's with it boards list

[1] https://github.com/rockchip-linux/rkbin

7.1.13 SiFive

HiFive Unleashed

FU540-C000 RISC-V SoC

The FU540-C000 is the world's first 4+1 64-bit RISC-V SoC from SiFive.

The HiFive Unleashed development platform is based on FU540-C000 and capable of running Linux.

Mainline support

The support for following drivers are already enabled:

1. SiFive UART Driver.
2. SiFive PRCI Driver for clock.
3. Cadence MACB ethernet driver for networking support.
4. SiFive SPI Driver.
5. MMC SPI Driver for MMC/SD support.

Booting from MMC using FSBL

Building

1. Add the RISC-V toolchain to your PATH.
2. Setup ARCH & cross compilation environment variable:

```
export CROSS_COMPILE=<riscv64 toolchain prefix>
```

3. make sifive_fu540_defconfig
4. make

Flashing

The current U-Boot port is supported in S-mode only and loaded from DRAM.

A prior stage M-mode firmware/bootloader (e.g OpenSBI) is required to boot the u-boot.bin in S-mode and provide M-mode runtime services.

Currently, the u-boot.bin is used as a payload of the OpenSBI FW_PAYLOAD firmware. We need to compile OpenSBI with below command:

```
make PLATFORM=generic FW_PAYLOAD_PATH=<path to u-boot-dtb.bin>
```

More detailed description of steps required to build FW_PAYLOAD firmware is beyond the scope of this document. Please refer OpenSBI documenation. (Note: OpenSBI git repo is at https://github.com/riscv/opensbi.git)

Once the prior stage firmware/bootloader binary is generated, it should be copied to the first partition of the sdcard.

```
sudo dd if=<prior_stage_firmware_binary> of=/dev/disk2s1 bs=1024
```

Booting

Once you plugin the sdcard and power up, you should see the U-Boot prompt.

Sample boot log from HiFive Unleashed board

```
U-Boot 2019.07-00024-g350ff02f5b (Jul 22 2019 - 11:45:02 +0530)

CPU:   rv64imafdc
Model: SiFive HiFive Unleashed A00
DRAM:  8 GiB
MMC:   spi@10050000:mmc@0: 0
```

(continues on next page)

(continued from previous page)

```
In:    serial@10010000
Out:   serial@10010000
Err:   serial@10010000
Net:   eth0: ethernet@10090000
Hit any key to stop autoboot:  0
=> version
U-Boot 2019.07-00024-g350ff02f5b (Jul 22 2019 - 11:45:02 +0530)

riscv64-linux-gcc.br_real (Buildroot 2018.11-rc2-00003-ga0787e9) 8.2.0
GNU ld (GNU Binutils) 2.31.1
=> mmc info
Device: spi@10050000:mmc@0
Manufacturer ID: 3
OEM: 5344
Name: SU08G
Bus Speed: 20000000
Mode: SD Legacy
Rd Block Len: 512
SD version 2.0
High Capacity: Yes
Capacity: 7.4 GiB
Bus Width: 1-bit
Erase Group Size: 512 Bytes
=> mmc part

Partition Map for MMC device 0  --   Partition Type: EFI

Part    Start LBA       End LBA         Name
        Attributes
        Type GUID
        Partition GUID
  1     0x00000800      0x000107ff      "bootloader"
        attrs:  0x0000000000000000
        type:   2e54b353-1271-4842-806f-e436d6af6985
        guid:   393bbd36-7111-491c-9869-ce24008f6403
  2     0x00040800      0x00ecdfde      ""
        attrs:  0x0000000000000000
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        guid:   7fc9a949-5480-48c7-b623-04923080757f
```

Now you can configure your networking, tftp server and use tftp boot method to load uImage.

```
=> setenv ipaddr 10.206.7.133
=> setenv netmask 255.255.252.0
=> setenv serverip 10.206.4.143
=> setenv gateway 10.206.4.1
```

If you want to use a flat kernel image such as Image file

```
=> tftpboot ${kernel_addr_r} /sifive/fu540/Image
ethernet@10090000: PHY present at 0
ethernet@10090000: Starting autonegotiation...
ethernet@10090000: Autonegotiation complete
ethernet@10090000: link up, 1000Mbps full-duplex (lpa: 0x3c00)
Using ethernet@10090000 device
TFTP from server 10.206.4.143; our IP address is 10.206.7.133
```

(continues on next page)

(continued from previous page)

```
Filename '/sifive/fu540/Image'.
Load address: 0x84000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##########################################
         1.2 MiB/s
done
Bytes transferred = 8867100 (874d1c hex)
```

Or if you want to use a compressed kernel image file such as Image.gz

```
=> tftpboot ${kernel_addr_r} /sifive/fu540/Image.gz
ethernet@10090000: PHY present at 0
ethernet@10090000: Starting autonegotiation...
ethernet@10090000: Autonegotiation complete
ethernet@10090000: link up, 1000Mbps full-duplex (lpa: 0x3c00)
Using ethernet@10090000 device
TFTP from server 10.206.4.143; our IP address is 10.206.7.133
Filename '/sifive/fu540/Image.gz'.
Load address: 0x84000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
```

(continues on next page)

(continued from previous page)

```
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##########################################
         1.2 MiB/s
done
Bytes transferred = 4809458 (4962f2 hex)
=>setenv kernel_comp_addr_r 0x90000000
=>setenv kernel_comp_size 0x500000
```

By this time, correct kernel image is loaded and required environment variables are set. You can proceed to load the ramdisk and device tree from the tftp server as well.

```
=> tftpboot ${ramdisk_addr_r} /sifive/fu540/uRamdisk
ethernet@10090000: PHY present at 0
ethernet@10090000: Starting autonegotiation...
ethernet@10090000: Autonegotiation complete
ethernet@10090000: link up, 1000Mbps full-duplex (lpa: 0x3c00)
Using ethernet@10090000 device
TFTP from server 10.206.4.143; our IP address is 10.206.7.133
Filename '/sifive/fu540/uRamdisk'.
Load address: 0x88300000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##############
         418.9 KiB/s
done
Bytes transferred = 2398272 (249840 hex)
=> tftpboot ${fdt_addr_r} /sifive/fu540/hifive-unleashed-a00.dtb
ethernet@10090000: PHY present at 0
ethernet@10090000: Starting autonegotiation...
ethernet@10090000: Autonegotiation complete
ethernet@10090000: link up, 1000Mbps full-duplex (lpa: 0x7c00)
Using ethernet@10090000 device
TFTP from server 10.206.4.143; our IP address is 10.206.7.133
Filename '/sifive/fu540/hifive-unleashed-a00.dtb'.
Load address: 0x88000000
Loading: ##
         1000 Bytes/s
done
```

(continues on next page)

(continued from previous page)

```
Bytes transferred = 5614 (15ee hex)
=> setenv bootargs "root=/dev/ram rw console=ttySIF0 ip=dhcp earlycon=sbi"
=> booti ${kernel_addr_r} ${ramdisk_addr_r} ${fdt_addr_r}
## Loading init Ramdisk from Legacy Image at 88300000 ...
   Image Name:   Linux RootFS
   Image Type:   RISC-V Linux RAMDisk Image (uncompressed)
   Data Size:    2398208 Bytes = 2.3 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Using Device Tree in place at 0000000088000000, end 00000000880045ed

Starting kernel ...

[    0.000000] OF: fdt: Ignoring memory range 0x80000000 - 0x80200000
[    0.000000] Linux version 5.3.0-rc1-00003-g460ac558152f (anup@anup-lab-machine)␣
↪(gcc version 8.2.0 (Buildroot 2018.11-rc2-00003-ga0787e9)) #6 SMP Mon Jul 22␣
↪10:01:01 IST 2019
[    0.000000] earlycon: sbi0 at I/O port 0x0 (options '')
[    0.000000] printk: bootconsole [sbi0] enabled
[    0.000000] Initial ramdisk at: 0x(____ptrval____) (2398208 bytes)
[    0.000000] Zone ranges:
[    0.000000]   DMA32    [mem 0x0000000080200000-0x00000000ffffffff]
[    0.000000]   Normal   [mem 0x0000000100000000-0x000000027fffffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080200000-0x000000027fffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080200000-0x000000027fffffff]
[    0.000000] software IO TLB: mapped [mem 0xfbfff000-0xfffff000] (64MB)
[    0.000000] CPU with hartid=0 is not available
[    0.000000] CPU with hartid=0 is not available
[    0.000000] elf_hwcap is 0x112d
[    0.000000] percpu: Embedded 18 pages/cpu s34584 r8192 d30952 u73728
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 2067975
[    0.000000] Kernel command line: root=/dev/ram rw console=ttySIF0 ip=dhcp␣
↪earlycon=sbi
[    0.000000] Dentry cache hash table entries: 1048576 (order: 11, 8388608 bytes,␣
↪linear)
[    0.000000] Inode-cache hash table entries: 524288 (order: 10, 4194304 bytes,␣
↪linear)
[    0.000000] Sorting __ex_table...
[    0.000000] mem auto-init: stack:off, heap alloc:off, heap free:off
[    0.000000] Memory: 8182308K/8386560K available (5916K kernel code, 368K rwdata,␣
↪1840K rodata, 213K init, 304K bss, 204252K reserved, 0K cma-reserved)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
[    0.000000] rcu: Hierarchical RCU implementation.
[    0.000000] rcu:     RCU restricting CPUs from NR_CPUS=8 to nr_cpu_ids=4.
[    0.000000] rcu: RCU calculated value of scheduler-enlistment delay is 25 jiffies.
[    0.000000] rcu: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
[    0.000000] NR_IRQS: 0, nr_irqs: 0, preallocated irqs: 0
[    0.000000] plic: mapped 53 interrupts with 4 handlers for 9 contexts.
[    0.000000] riscv_timer_init_dt: Registering clocksource cpuid [0] hartid [1]
[    0.000000] clocksource: riscv_clocksource: mask: 0xffffffffffffffff max_cycles:␣
↪0x1d854df40, max_idle_ns: 3526361616960 ns
```

(continues on next page)

(continued from previous page)

```
[    0.000006] sched_clock: 64 bits at 1000kHz, resolution 1000ns, wraps every␣
↪2199023255500ns
[    0.008559] Console: colour dummy device 80x25
[    0.012989] Calibrating delay loop (skipped), value calculated using timer␣
↪frequency.. 2.00 BogoMIPS (lpj=4000)
[    0.023104] pid_max: default: 32768 minimum: 301
[    0.028273] Mount-cache hash table entries: 16384 (order: 5, 131072 bytes, linear)
[    0.035765] Mountpoint-cache hash table entries: 16384 (order: 5, 131072 bytes,␣
↪linear)
[    0.045307] rcu: Hierarchical SRCU implementation.
[    0.049875] smp: Bringing up secondary CPUs ...
[    0.055729] smp: Brought up 1 node, 4 CPUs
[    0.060599] devtmpfs: initialized
[    0.064819] random: get_random_u32 called from bucket_table_alloc.isra.10+0x4e/
↪0x160 with crng_init=0
[    0.073720] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_
↪ns: 7645041785100000 ns
[    0.083176] futex hash table entries: 1024 (order: 4, 65536 bytes, linear)
[    0.090721] NET: Registered protocol family 16
[    0.106319] vgaarb: loaded
[    0.108670] SCSI subsystem initialized
[    0.112515] usbcore: registered new interface driver usbfs
[    0.117758] usbcore: registered new interface driver hub
[    0.123167] usbcore: registered new device driver usb
[    0.128905] clocksource: Switched to clocksource riscv_clocksource
[    0.141239] NET: Registered protocol family 2
[    0.145506] tcp_listen_portaddr_hash hash table entries: 4096 (order: 4, 65536␣
↪bytes, linear)
[    0.153754] TCP established hash table entries: 65536 (order: 7, 524288 bytes,␣
↪linear)
[    0.163466] TCP bind hash table entries: 65536 (order: 8, 1048576 bytes, linear)
[    0.173468] TCP: Hash tables configured (established 65536 bind 65536)
[    0.179739] UDP hash table entries: 4096 (order: 5, 131072 bytes, linear)
[    0.186627] UDP-Lite hash table entries: 4096 (order: 5, 131072 bytes, linear)
[    0.194117] NET: Registered protocol family 1
[    0.198417] RPC: Registered named UNIX socket transport module.
[    0.203887] RPC: Registered udp transport module.
[    0.208664] RPC: Registered tcp transport module.
[    0.213429] RPC: Registered tcp NFSv4.1 backchannel transport module.
[    0.219944] PCI: CLS 0 bytes, default 64
[    0.224170] Unpacking initramfs...
[    0.262347] Freeing initrd memory: 2336K
[    0.266531] workingset: timestamp_bits=62 max_order=21 bucket_order=0
[    0.280406] NFS: Registering the id_resolver key type
[    0.284798] Key type id_resolver registered
[    0.289048] Key type id_legacy registered
[    0.293114] nfs4filelayout_init: NFSv4 File Layout Driver Registering...
[    0.300262] NET: Registered protocol family 38
[    0.304432] Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
[    0.311862] io scheduler mq-deadline registered
[    0.316461] io scheduler kyber registered
[    0.356421] Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
[    0.363004] 10010000.serial: ttySIF0 at MMIO 0x10010000 (irq = 4, base_baud = 0) is␣
↪a SiFive UART v0
[    0.371468] printk: console [ttySIF0] enabled
[    0.371468] printk: console [ttySIF0] enabled
```

(continues on next page)

(continued from previous page)

```
[    0.380223] printk: bootconsole [sbi0] disabled
[    0.380223] printk: bootconsole [sbi0] disabled
[    0.389589] 10011000.serial: ttySIF1 at MMIO 0x10011000 (irq = 1, base_baud = 0) is␣
↪a SiFive UART v0
[    0.398680] [drm] radeon kernel modesetting enabled.
[    0.412395] loop: module loaded
[    0.415214] sifive_spi 10040000.spi: mapped; irq=3, cs=1
[    0.420628] sifive_spi 10050000.spi: mapped; irq=5, cs=1
[    0.425897] libphy: Fixed MDIO Bus: probed
[    0.429964] macb 10090000.ethernet: Registered clk switch 'sifive-gemgxl-mgmt'
[    0.436743] macb: GEM doesn't support hardware ptp.
[    0.441621] libphy: MACB_mii_bus: probed
[    0.601316] Microsemi VSC8541 SyncE 10090000.ethernet-ffffffff:00: attached PHY␣
↪driver [Microsemi VSC8541 SyncE] (mii_bus:phy_addr=10090000.ethernet-ffffffff:00,␣
↪irq=POLL)
[    0.615857] macb 10090000.ethernet eth0: Cadence GEM rev 0x10070109 at 0x10090000␣
↪irq 6 (70:b3:d5:92:f2:f3)
[    0.625634] e1000e: Intel(R) PRO/1000 Network Driver - 3.2.6-k
[    0.631381] e1000e: Copyright(c) 1999 - 2015 Intel Corporation.
[    0.637382] ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
[    0.643799] ehci-pci: EHCI PCI platform driver
[    0.648261] ehci-platform: EHCI generic platform driver
[    0.653497] ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
[    0.659599] ohci-pci: OHCI PCI platform driver
[    0.664055] ohci-platform: OHCI generic platform driver
[    0.669448] usbcore: registered new interface driver uas
[    0.674575] usbcore: registered new interface driver usb-storage
[    0.680642] mousedev: PS/2 mouse device common for all mice
[    0.709493] mmc_spi spi1.0: SD/MMC host mmc0, no DMA, no WP, no poweroff, cd polling
[    0.716615] usbcore: registered new interface driver usbhid
[    0.722023] usbhid: USB HID core driver
[    0.726738] NET: Registered protocol family 10
[    0.731359] Segment Routing with IPv6
[    0.734332] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[    0.740687] NET: Registered protocol family 17
[    0.744660] Key type dns_resolver registered
[    0.806775] mmc0: host does not support reading read-only switch, assuming write-
↪enable
[    0.814020] mmc0: new SDHC card on SPI
[    0.820137] mmcblk0: mmc0:0000 SU08G 7.40 GiB
[    0.850220]  mmcblk0: p1 p2
[    3.821524] macb 10090000.ethernet eth0: link up (1000/Full)
[    3.828938] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[    3.848919] Sending DHCP requests .., OK
[    6.252076] IP-Config: Got DHCP answer from 10.206.4.1, my address is 10.206.7.133
[    6.259624] IP-Config: Complete:
[    6.262831]      device=eth0, hwaddr=70:b3:d5:92:f2:f3, ipaddr=10.206.7.133,␣
↪mask=255.255.252.0, gw=10.206.4.1
[    6.272809]      host=dhcp-10-206-7-133, domain=sdcorp.global.sandisk.com, nis-
↪domain=(none)
[    6.281228]      bootserver=10.206.126.11, rootserver=10.206.126.11, rootpath=
[    6.281232]      nameserver0=10.86.1.1, nameserver1=10.86.2.1
[    6.294179]      ntpserver0=10.86.1.1, ntpserver1=10.86.2.1
[    6.301026] Freeing unused kernel memory: 212K
[    6.304683] This architecture does not have kernel memory protection.
[    6.311121] Run /init as init process
```

(continues on next page)

(continued from previous page)

```
           _  _
          | ||_|
          | | _ ____  _   _  _  _
          | || |  _ \| | | |\ \/ /
          | || | | | | |_| |/    \
          |_||_|_| |_|\____|\_/\_/

               Busybox Rootfs

Please press Enter to activate this console.
/ #
```

Booting from MMC using U-Boot SPL

Building

Before building U-Boot SPL, OpenSBI must be built first. OpenSBI can be cloned and built for FU540 as below:

```
git clone https://github.com/riscv/opensbi.git
cd opensbi
make PLATFORM=generic
export OPENSBI=<path to opensbi/build/platform/generic/firmware/fw_dynamic.bin>
```

Now build the U-Boot SPL and U-Boot proper

```
cd <U-Boot-dir>
make sifive_fu540_defconfig
make
```

This will generate spl/u-boot-spl.bin and FIT image (u-boot.itb)

Flashing

ZSBL loads the U-Boot SPL (u-boot-spl.bin) from a partition with GUID type 5B193300-FC78-40CD-8002-E86C45580B47

U-Boot SPL expects a U-Boot FIT image (u-boot.itb) from a partition with GUID type 2E54B353-1271-4842-806F-E436D6AF6985

FIT image (u-boot.itb) is a combination of fw_dynamic.bin, u-boot-nodtb.bin and device tree blob (hifive-unleashed-a00.dtb)

Format the SD card (make sure the disk has GPT, otherwise use gdisk to switch)

```
# sudo sgdisk --clear \
> --set-alignment=2 \
> --new=1:34:2081 --change-name=1:loader1 --typecode=1:5B193300-FC78-40CD-8002-
↪E86C45580B47 \
> --new=2:2082:10273 --change-name=2:loader2 --typecode=2:2E54B353-1271-4842-806F-
↪E436D6AF6985 \
> --new=3:10274: --change-name=3:rootfs --typecode=3:0FC63DAF-8483-4772-8E79-
↪3D69D8477DE4 \
> /dev/sda
```

Program the SD card

```
sudo dd if=spl/u-boot-spl.bin of=/dev/sda seek=34
sudo dd if=u-boot.itb of=/dev/sda seek=2082
```

Booting

Once you plugin the sdcard and power up, you should see the U-Boot prompt.

Sample boot log from HiFive Unleashed board

```
U-Boot SPL 2020.04-rc2-00109-g63efc7e07e-dirty (Apr 30 2020 - 13:52:36 +0530)
Trying to boot from MMC1


U-Boot 2020.04-rc2-00109-g63efc7e07e-dirty (Apr 30 2020 - 13:52:36 +0530)

CPU:   rv64imafdc
Model: SiFive HiFive Unleashed A00
DRAM:  8 GiB
MMC:   spi@10050000:mmc@0: 0
In:    serial@10010000
Out:   serial@10010000
Err:   serial@10010000
Net:   eth0: ethernet@10090000
Hit any key to stop autoboot:  0
=> version
U-Boot 2020.04-rc2-00109-g63efc7e07e-dirty (Apr 30 2020 - 13:52:36 +0530)

riscv64-unknown-linux-gnu-gcc (crosstool-NG 1.24.0.37-3f461da) 9.2.0
GNU ld (crosstool-NG 1.24.0.37-3f461da) 2.32
=> mmc info
Device: spi@10050000:mmc@0
Manufacturer ID: 3
OEM: 5344
Name: SC16G
Bus Speed: 20000000
Mode: SD Legacy
Rd Block Len: 512
SD version 2.0
High Capacity: Yes
Capacity: 14.8 GiB
Bus Width: 1-bit
Erase Group Size: 512 Bytes
=> mmc part

Partition Map for MMC device 0  --   Partition Type: EFI

Part    Start LBA       End LBA         Name
Attributes
Type GUID
Partition GUID
1     0x00000022      0x00000821      "loader1"
attrs:  0x0000000000000000
type:   5b193300-fc78-40cd-8002-e86c45580b47
guid:   66e2b5d2-74db-4df8-ad6f-694b3617f87f
```

(continues on next page)

(continued from previous page)

```
2     0x00000822      0x00002821      "loader2"
attrs:  0x0000000000000000
type:   2e54b353-1271-4842-806f-e436d6af6985
guid:   8befaeaf-bca0-435d-b002-e201f37c0a2f
3     0x00002822      0x01dacbde      "rootfs"
attrs:  0x0000000000000000
type:   0fc63daf-8483-4772-8e79-3d69d8477de4
type:   linux
guid:   9faa81b6-39b1-4418-af5e-89c48f29c20d
```

Booting from SPI

Use Building steps from "Booting from MMC using U-Boot SPL" section.

Partition the SPI in Linux via mtdblock. (Require to boot the board in SD boot mode by enabling MTD block in Linux)

Use prebuilt image from here [1], which support to partition the SPI flash.

```
# sgdisk --clear \
> --set-alignment=2 \
> --new=1:40:2087 --change-name=1:loader1 --typecode=1:5B193300-FC78-40CD-8002-
↪E86C45580B47 \
> --new=2:2088:10279 --change-name=2:loader2 --typecode=2:2E54B353-1271-4842-806F-
↪E436D6AF6985 \
> --new=3:10536:65494 --change-name=3:rootfs --typecode=3:0FC63DAF-8483-4772-8E79-
↪3D69D8477DE4 \
> /dev/mtdblock0
```

Program the SPI (Require to boot the board in SD boot mode)

Execute below steps on U-Boot proper,

```
tftpboot $kernel_addr_r u-boot-spl.bin
sf erase 0x5000 $filesize
sf write $kernel_addr_r 0x5000 $filesize

tftpboot $kernel_addr_r u-boot.itb
sf erase 0x105000 $filesize
sf write $kernel_addr_r 0x105000 $filesize
```

Power off the board

Change DIP switches MSEL[3:0] are set to 0110

Power up the board.

[1] https://github.com/amarula/bsp-sifive

7.1.14 Sipeed

MAIX

Several of the Sipeed Maix series of boards cotain the Kendryte K210 processor, a 64-bit RISC-V CPU. This processor contains several peripherals to accelerate neural network processing and other "ai" tasks. This includes a "KPU" neural network processor, an audio processor supporting beamforming reception, and a digital video port supporting capture and output at VGA resolution. Other peripherals include 8M of SRAM (accessible with and without caching); remappable pins, including 40 GPIOs; AES, FFT, and SHA256 accelerators; a DMA controller; and I2C, I2S, and SPI controllers. Maix peripherals vary, but include spi flash; on-board usb-serial bridges; ports for cameras, displays, and sd cards; and ESP32 chips.

Currently, only the Sipeed MAIX BiT V2.0 (bitm) and Sipeed MAIXDUINO are supported, but the boards are fairly similar.

Documentation for Maix boards is available from Sipeed's website. Documentation for the Kendryte K210 is available from Kendryte's website. However, hardware details are rather lacking, so most technical reference has been taken from the standalone sdk.

Build and boot steps

To build U-Boot, run

```
make <defconfig>
make CROSS_COMPILE=<your cross compile prefix>
```

To flash U-Boot, run

```
kflash -tp /dev/<your tty here> -B <board_id> u-boot-dtb.bin
```

The board provides two serial devices, e.g.

- /dev/serial/by-id/usb-Kongou_Hikari_Sipeed-Debug_12345678AB-if00-port0
- /dev/serial/by-id/usb-Kongou_Hikari_Sipeed-Debug_12345678AB-if01-port0

Which one is used for flashing depends on the board.

Currently only a small subset of the board features are supported. So we can use the same default configuration and device tree. In the long run we may need separate settings.

Board	defconfig	board_id	TTY device
Sipeed MAIX BiT	sipeed_maix_bitm_defconfig	bit	first
Sipeed MAIX BiT with Mic	sipeed_maix_bitm_defconfig	bit_mic	first
Sipeed MAIXDUINO	sipeed_maix_bitm_defconfig	maixduino	first
Sipeed MAIX GO		goE	second
Sipeed MAIX ONE DOCK		dan	first

Flashing causes a reboot of the device. Parameter -t specifies that the serial console shall be opened immediately. Boot output should look like the following:

```
U-Boot 2020.04-rc2-00087-g2221cc09c1-dirty (Feb 28 2020 - 13:53:09 -0500)

DRAM:  8 MiB
In:    serial@38000000
Out:   serial@38000000
Err:   serial@38000000
=>
```

OpenSBI

OpenSBI is an open source supervisor execution environment implementing the RISC-V Supervisor Binary Interface Specification [1]. One of its features is to intercept run-time exceptions, e.g. for unaligned access or illegal instructions, and to emulate the failing instructions.

The OpenSBI source can be downloaded via:

```
git clone https://github.com/riscv/opensbi
```

As OpenSBI will be loaded at 0x80000000 we have to adjust the U-Boot text base. Furthermore we have to enable building U-Boot for S-mode:

```
CONFIG_SYS_TEXT_BASE=0x80020000
CONFIG_RISCV_SMODE=y
```

Both settings are contained in sipeed_maix_smode_defconfig so we can build U-Boot with:

```
make sipeed_maix_smode_defconfig
make
```

To build OpenSBI with U-Boot as a payload:

```
cd opensbi
make \
PLATFORM=kendryte/k210 \
FW_PAYLOAD=y \
FW_PAYLOAD_OFFSET=0x20000 \
FW_PAYLOAD_PATH=<path to U-Boot>/u-boot-dtb.bin
```

The value of FW_PAYLOAD_OFFSET must match CONFIG_SYS_TEXT_BASE - 0x80000000.

The file to flash is build/platform/kendryte/k210/firmware/fw_payload.bin.

Loading Images

To load a kernel, transfer it over serial.

```
=> loady 80000000 1500000
## Switch baudrate to 1500000 bps and press ENTER ...

*** baud: 1500000

*** baud: 1500000 ***
## Ready for binary (ymodem) download to 0x80000000 at 1500000 bps...
C
*** file: loader.bin
$ sz -vv loader.bin
Sending: loader.bin
Bytes Sent:2478208   BPS:72937
Sending:
Ymodem sectors/kbytes sent:   0/ 0k
Transfer complete

*** exit status: 0 ***
## Total Size      = 0x0025d052 = 2478162 Bytes
## Switch baudrate to 115200 bps and press ESC ...

*** baud: 115200

*** baud: 115200 ***
=>
```

Running Programs

Binaries

To run a bare binary, use the go command:

```
=> loady
## Ready for binary (ymodem) download to 0x80000000 at 115200 bps...
C
*** file: ./examples/standalone/hello_world.bin
$ sz -vv ./examples/standalone/hello_world.bin
Sending: hello_world.bin
Bytes Sent:   4864   BPS:649
Sending:
Ymodem sectors/kbytes sent:   0/ 0k
Transfer complete

*** exit status: 0 ***
(CAN) packets, 5 retries
## Total Size      = 0x000012f8 = 4856 Bytes
=> go 80000000
## Starting application at 0x80000000 ...
Example expects ABI version 9
Actual U-Boot ABI version 9
Hello World
argc = 1
argv[0] = "80000000"
argv[1] = "<NULL>"
Hit any key to exit ...
```

Legacy Images

To run legacy images, use the bootm command:

```
$ tools/mkimage -A riscv -O u-boot -T standalone -C none -a 80000000 -e 80000000 -d␣
↪examples/standalone/hello_world.bin hello_world.img
Image Name:
Created:      Thu Mar  5 12:04:10 2020
Image Type:   RISC-V U-Boot Standalone Program (uncompressed)
Data Size:    4856 Bytes = 4.74 KiB = 0.00 MiB
Load Address: 80000000
Entry Point:  80000000

$ picocom -b 115200 /dev/ttyUSB0
=> loady
## Ready for binary (ymodem) download to 0x80000000 at 115200 bps...
C
*** file: hello_world.img
$ sz -vv hello_world.img
Sending: hello_world.img
Bytes Sent:   4992   BPS:665
Sending:
Ymodem sectors/kbytes sent:   0/ 0k
Transfer complete

*** exit status: 0 ***
CAN) packets, 3 retries
## Total Size      = 0x00001338 = 4920 Bytes
```

(continues on next page)

(continued from previous page)

```
=> bootm
## Booting kernel from Legacy Image at 80000000 ...
   Image Name:
   Image Type:   RISC-V U-Boot Standalone Program (uncompressed)
   Data Size:    4856 Bytes = 4.7 KiB
   Load Address: 80000000
   Entry Point:  80000000
   Verifying Checksum ... OK
   Loading Standalone Program
Example expects ABI version 9
Actual U-Boot ABI version 9
Hello World
argc = 0
argv[0] = "<NULL>"
Hit any key to exit ...
```

Pin Assignment

The K210 contains a Fully Programmable I/O Array (FPIOA), which can remap any of its 256 input functions to any any of 48 output pins. The following table has the default pin assignments for the BitM.

Pin	Function	Comment
IO_0	JTAG_TCLK	
IO_1	JTAG_TDI	
IO_2	JTAG_TMS	
IO_3	JTAG_TDO	
IO_4	UARTHS_RX	
IO_5	UARTHS_TX	
IO_6		Not set
IO_7		Not set
IO_8	GPIO_0	
IO_9	GPIO_1	
IO_10	GPIO_2	
IO_11	GPIO_3	
IO_12	GPIO_4	Green LED
IO_13	GPIO_5	Red LED
IO_14	GPIO_6	Blue LED
IO_15	GPIO_7	
IO_16	GPIOHS_0	ISP
IO_17	GPIOHS_1	
IO_18	I2S0_SCLK	MIC CLK
IO_19	I2S0_WS	MIC WS
IO_20	I2S0_IN_D0	MIC SD
IO_21	GPIOHS_5	
IO_22	GPIOHS_6	
IO_23	GPIOHS_7	
IO_24	GPIOHS_8	
IO_25	GPIOHS_9	
IO_26	SPI1_D1	MMC MISO
IO_27	SPI1_SCLK	MMC CLK
IO_28	SPI1_D0	MMC MOSI
IO_29	GPIOHS_13	MMC CS
IO_30	GPIOHS_14	

Continued on next page

Table 1 – continued from previous page

Pin	Function	Comment
IO_31	GPIOHS_15	
IO_32	GPIOHS_16	
IO_33	GPIOHS_17	
IO_34	GPIOHS_18	
IO_35	GPIOHS_19	
IO_36	GPIOHS_20	Panel CS
IO_37	GPIOHS_21	Panel RST
IO_38	GPIOHS_22	Panel DC
IO_39	SPI0_SCK	Panel WR
IO_40	SCCP_SDA	
IO_41	SCCP_SCLK	
IO_42	DVP_RST	
IO_43	DVP_VSYNC	
IO_44	DVP_PWDN	
IO_45	DVP_HSYNC	
IO_46	DVP_XCLK	
IO_47	DVP_PCLK	

Over- and Under-clocking

To change the clock speed of the K210, you will need to enable `CONFIG_CLK_K210_SET_RATE` and edit the board's device tree. To do this, add a section to `arch/riscv/arch/riscv/dts/k210-maix-bit.dts` like the following:

```
&sysclk {
    assigned-clocks = <&sysclk K210_CLK_PLL0>;
    assigned-clock-rates = <800000000>;
};
```

There are three PLLs on the K210: PLL0 is the parent of most of the components, including the CPU and RAM. PLL1 is the parent of the neural network coprocessor. PLL2 is the parent of the sound processing devices. Note that child clocks of PLL0 and PLL2 run at *half* the speed of the PLLs. For example, if PLL0 is running at 800 MHz, then the CPU will run at 400 MHz. This is the example given above. The CPU can be overclocked to around 600 MHz, and underclocked to 26 MHz.

It is possible to set PLL2's parent to PLL0. The plls are more accurate when converting between similar frequencies. This makes it easier to get an accurate frequency for I2S. As an example, consider sampling an I2S device at 44.1 kHz. On this device, the I2S serial clock runs at 64 times the sample rate. Therefore, we would like to run PLL2 at an even multiple of 2.8224 MHz. If PLL2's parent is IN0, we could use a frequency of 390 MHz (the same as the CPU's default speed). Dividing by 138 yields a serial clock of about 2.8261 MHz. This results in a sample rate of 44.158 kHz—around 50 Hz or .1% too fast. If, instead, we set PLL2's parent to PLL1 running at 390 MHz, and request a rate of 2.8224 * 136 = 383.8464 MHz, the achieved rate is 383.90625 MHz. Dividing by 136 yields a serial clock of about 2.8228 MHz. This results in a sample rate of 44.107 kHz—just 7 Hz or .02% too fast. This configuration is shown in the following example:

```
&sysclk {
    assigned-clocks = <&sysclk K210_CLK_PLL1>, <&sysclk K210_CLK_PLL2>;
    assigned-clock-parents = <0>, <&sysclk K210_CLK_PLL1>;
    assigned-clock-rates = <390000000>, <383846400>;
};
```

There are a couple of quirks to the PLLs. First, there are more frequency ratios just above and below 1.0, but there is a small gap around 1.0. To be explicit, if the input frequency is 100 MHz, it would be impossible to have an output of 99 or 101 MHz. In addition, there is a maximum frequency for the internal VCO, so higher input/output frequencies will be less accurate than lower ones.

Technical Details

Boot Sequence

1. RESET pin is deasserted. The pin is connected to the RESET button. It can also be set to low via either the DTR or the RTS line of the serial interface (depending on the board).
2. Both harts begin executing at `0x00001000`.
3. Both harts jump to firmware at `0x88000000`.
4. One hart is chosen as a boot hart.
5. Firmware reads the value of pin `IO_16` (ISP). This pin is connected to the `BOOT` button. The pin can equally be set to low via either the DTR or RTS line of the serial interface (depending on the board).
   - If the pin is low, enter ISP mode. This mode allows loading data to ram, writing it to flash, and booting from specific addresses.
   - If the pin is high, continue boot.
6. Firmware reads the next stage from flash (SPI3) to address `0x80000000`.
   - If byte 0 is 1, the next stage is decrypted using the built-in AES accelerator and the one-time programmable, 128-bit AES key.
   - Bytes 1 to 4 hold the length of the next stage.
   - The SHA-256 sum of the next stage is automatically calculated, and verified against the 32 bytes following the next stage.
7. The boot hart sends an IPI to the other hart telling it to jump to the next stage.
8. The boot hart jumps to `0x80000000`.

Debug UART

The Debug UART is provided with the following settings:

```
CONFIG_DEBUG_UART=y
CONFIG_DEBUG_UART_SIFIVE=y
CONFIG_DEBUG_UART_BASE=0x38000000
CONFIG_DEBUG_UART_CLOCK=390000000
```

Resetting the board

The MAIX boards can be reset using the DTR and RTS lines of the serial console. How the lines are used depends on the specific board. See the code of kflash.py for details.

This is the reset sequence for the MAXDUINO and MAIX BiT with Mic:

```
def reset(self):
     self.device.setDTR(False)
     self.device.setRTS(False)
     time.sleep(0.1)
     self.device.setDTR(True)
     time.sleep(0.1)
     self.device.setDTR(False)
     time.sleep(0.1)
```

and this for the MAIX Bit:

```
def reset(self):
     self.device.setDTR(False)
     self.device.setRTS(False)
     time.sleep(0.1)
     self.device.setRTS(True)
     time.sleep(0.1)
     self.device.setRTS(False)
     time.sleep(0.1)
```

Memory Map

Address	Size	Description
0x00000000	0x1000	debug
0x00001000	0x1000	rom
0x02000000	0xC000	clint
0x0C000000	0x4000000	plic
0x38000000	0x1000	uarths
0x38001000	0x1000	gpiohs
0x40000000	0x400000	sram0 (non-cached)
0x40400000	0x200000	sram1 (non-cached)
0x40600000	0x200000	airam (non-cached)
0x40800000	0xC00000	kpu
0x42000000	0x400000	fft
0x50000000	0x1000	dmac
0x50200000	0x200000	apb0
0x50200000	0x80	gpio
0x50210000	0x100	uart0
0x50220000	0x100	uart1
0x50230000	0x100	uart2
0x50240000	0x100	spi slave
0x50250000	0x200	i2s0
0x50250200	0x200	apu
0x50260000	0x200	i2s1
0x50270000	0x200	i2s2
0x50280000	0x100	i2c0
0x50290000	0x100	i2c1
0x502A0000	0x100	i2c2
0x502B0000	0x100	fpioa
0x502C0000	0x100	sha256
0x502D0000	0x100	timer0
0x502E0000	0x100	timer1
0x502F0000	0x100	timer2
0x50400000	0x200000	apb1
0x50400000	0x100	wdt0
0x50410000	0x100	wdt1
0x50420000	0x100	otp control
0x50430000	0x100	dvp
0x50440000	0x100	sysctl
0x50450000	0x100	aes
0x50460000	0x100	rtc
0x52000000	0x4000000	apb2
0x52000000	0x100	spi0

Continued on next page

Table 2 – continued from previous page

Address	Size	Description
0x53000000	0x100	spi1
0x54000000	0x200	spi3
0x80000000	0x400000	sram0 (cached)
0x80400000	0x200000	sram1 (cached)
0x80600000	0x200000	airam (cached)
0x88000000	0x20000	otp
0x88000000	0xC200	firmware
0x8801C000	0x1000	riscv priv spec 1.9 config
0x8801D000	0x2000	flattened device tree (contains only addresses and interrupts)
0x8801F000	0x1000	credits

Links

[1] **https://github.com/riscv/riscv-sbi-doc** RISC-V Supervisor Binary Interface Specification

7.1.15 STMicroelectronics

STM32MP15x boards

This is a quick instruction for setup STM32MP15x boards.

Supported devices

U-Boot supports STMP32MP15x SoCs:

- STM32MP157
- STM32MP153
- STM32MP151

The STM32MP15x is a Cortex-A MPU aimed at various applications.

It features:

- Dual core Cortex-A7 application core (Single on STM32MP151)
- 2D/3D image composition with GPU (only on STM32MP157)
- Standard memories interface support
- Standard connectivity, widely inherited from the STM32 MCU family
- Comprehensive security support

Each line comes with a security option (cryptography & secure boot) and a Cortex-A frequency option:

- A : Cortex-A7 @ 650 MHz
- C : Secure Boot + HW Crypto + Cortex-A7 @ 650 MHz
- D : Cortex-A7 @ 800 MHz
- F : Secure Boot + HW Crypto + Cortex-A7 @ 800 MHz

Everything is supported in Linux but U-Boot is limited to:

1. UART
2. SD card/MMC controller (SDMMC)
3. NAND controller (FMC)

4. NOR controller (QSPI)
5. USB controller (OTG DWC2)
6. Ethernet controller

And the necessary drivers

1. I2C
2. STPMIC1 (PMIC and regulator)
3. Clock, Reset, Sysreset
4. Fuse

Currently the following boards are supported:

- stm32mp157a-dk1.dts
- stm32mp157c-dk2.dts
- stm32mp157c-ed1.dts
- stm32mp157c-ev1.dts
- stm32mp15xx-dhcor-avenger96.dts

Boot Sequences

3 boot configurations are supported with:

ROM code	**FSBL**	**SSBL**	**OS**
	First Stage Bootloader	Second Stage Bootloader	Linux Kernel
	embedded RAM	DDR	

The Trusted boot chain

defconfig_file : stm32mp15_trusted_defconfig

ROM code	FSBL	SSBL	OS
	Trusted Firmware-A (TF-A)	U-Boot	Linux
TrustZone	secure monitor		

TF-A performs a full initialization of Secure peripherals and installs a secure monitor, BL32:

- SPMin provided by TF-A or
- OP-TEE from specific partitions (teeh, teed, teex).

U-Boot is running in normal world and uses the secure monitor to access to secure resources.

The Basic boot chain

defconfig_file : stm32mp15_basic_defconfig

ROM code	FSBL	SSBL	OS
	U-Boot SPL	U-Boot	Linux
TrustZone		PSCI from U-Boot	

SPL has limited security initialization

U-Boot is running in secure mode and provide a secure monitor to the kernel with only PSCI support (Power State Coordination Interface defined by ARM).

All the STM32MP15x boards supported by U-Boot use the same generic board stm32mp1 which support all the bootable devices.

Each board is configured only with the associated device tree.

Device Tree Selection

You need to select the appropriate device tree for your board, the supported device trees for STM32MP15x are:

- ev1: eval board with pmic stpmic1 (ev1 = mother board + daughter ed1)
  - stm32mp157c-ev1
- ed1: daughter board with pmic stpmic1
  - stm32mp157c-ed1
- dk1: Discovery board
  - stm32mp157a-dk1
- dk2: Discovery board = dk1 with a BT/WiFI combo and a DSI panel
  - stm32mp157c-dk2
- avenger96: Avenger96 board from Arrow Electronics based on DH Elec. DHCOR SoM
  - stm32mp15xx-dhcor-avenger96

Build Procedure

1. Install the required tools for U-Boot
   - install package needed in U-Boot makefile (libssl-dev, swig, libpython-dev...)
   - install ARMv7 toolchain for 32bit Cortex-A (from Linaro, from SDK for STM32MP15x, or any crosstoolchains from your distribution) (you can use any gcc cross compiler compatible with U-Boot)
2. Set the cross compiler:
   ```
   # export CROSS_COMPILE=/path/to/toolchain/arm-linux-gnueabi-
   ```
3. Select the output directory (optional):
   ```
   # export KBUILD_OUTPUT=/path/to/output
   ```
   for example: use one output directory for each configuration:
   ```
   # export KBUILD_OUTPUT=stm32mp15_trusted
   # export KBUILD_OUTPUT=stm32mp15_basic
   ```
   you can build outside of code directory:
   ```
   # export KBUILD_OUTPUT=../build/stm32mp15_trusted
   ```
4. Configure U-Boot:

```
# make <defconfig_file>
```

with <defconfig_file>:

- For **trusted** boot mode : **stm32mp15_trusted_defconfig**
- For basic boot mode: stm32mp15_basic_defconfig

5. Configure the device-tree and build the U-Boot image:

```
# make DEVICE_TREE=<name> all
```

Examples:

a) trusted boot on ev1:

```
# export KBUILD_OUTPUT=stm32mp15_trusted
# make stm32mp15_trusted_defconfig
# make DEVICE_TREE=stm32mp157c-ev1 all
```

b) trusted with OP-TEE boot on dk2:

```
# export KBUILD_OUTPUT=stm32mp15_trusted
# make stm32mp15_trusted_defconfig
# make DEVICE_TREE=stm32mp157c-dk2 all
```

c) basic boot on ev1:

```
# export KBUILD_OUTPUT=stm32mp15_basic
# make stm32mp15_basic_defconfig
# make DEVICE_TREE=stm32mp157c-ev1 all
```

d) basic boot on ed1:

```
# export KBUILD_OUTPUT=stm32mp15_basic
# make stm32mp15_basic_defconfig
# make DEVICE_TREE=stm32mp157c-ed1 all
```

e) basic boot on dk1:

```
# export KBUILD_OUTPUT=stm32mp15_basic
# make stm32mp15_basic_defconfig
# make DEVICE_TREE=stm32mp157a-dk1 all
```

f) basic boot on avenger96:

```
# export KBUILD_OUTPUT=stm32mp15_basic
# make stm32mp15_basic_defconfig
# make DEVICE_TREE=stm32mp15xx-dhcor-avenger96 all
```

6. Output files

   BootRom and TF-A expect binaries with STM32 image header SPL expects file with U-Boot uImage header

   So in the output directory (selected by KBUILD_OUTPUT), you can found the needed files:

- For **Trusted** boot (with or without OP-TEE)
  - FSBL = **tf-a.stm32** (provided by TF-A compilation)
  - SSBL = **u-boot.stm32**
- For Basic boot

- FSBL = spl/u-boot-spl.stm32
- SSBL = u-boot.img

Switch Setting for Boot Mode

You can select the boot mode, on the board with one switch, to select the boot pin values = BOOT0, BOOT1, BOOT2

Boot Mode	*BOOT2*	*BOOT1*	*BOOT0*
Recovery	0	0	0
NOR	0	0	1
eMMC	0	1	0
NAND	0	1	1
Reserved	1	0	0
SD-Card	1	0	1
Recovery	1	1	0
SPI-NAND	1	1	1

- on the **daugther board ed1 = MB1263** with the switch SW1
- on **Avenger96** with switch S3 (NOR and SPI-NAND are not applicable)
- on board **DK1/DK2** with the switch SW1 = BOOT0, BOOT2 with only 2 pins available (BOOT1 is forced to 0 and NOR not supported), the possible value becomes:

Boot Mode	*BOOT2*	*BOOT0*
Recovery	0	0
NOR (NA)	0	1
Reserved	1	0
SD-Card	1	1

Recovery is a boot from serial link (UART/USB) and it is used with STM32CubeProgrammer tool to load executable in RAM and to update the flash devices available on the board (NOR/NAND/eMMC/SD card).

The communication between HOST and board is based on

- for UARTs : the uart protocol used with all MCU STM32
- for USB : based on USB DFU 1.1 (without the ST extensions used on MCU STM32)

Prepare an SD card

The minimal requirements for STMP32MP15x boot up to U-Boot are:

- GPT partitioning (with gdisk or with sgdisk)
- 2 fsbl partitions, named fsbl1 and fsbl2, size at least 256KiB
- one ssbl partition for U-Boot

Then the minimal GPT partition is:

Num	*Name*	*Size*	*Content*
1	fsbl1	256 KiB	TF-A or SPL
2	fsbl2	256 KiB	TF-A or SPL
3	ssbl	enought	U-Boot
4	<any>	<any>	Rootfs

Add a 4th partition (Rootfs) marked bootable with a file extlinux.conf following the Generic Distribution feature (doc/README.distro for use).

According the used card reader select the correct block device (for example /dev/sdx or /dev/mmcblk0).

In the next example, it is /dev/mmcblk0

For example: with gpt table with 128 entries

a) remove previous formatting:

```
# sgdisk -o /dev/<SD card dev>
```

b) create minimal image:

```
# sgdisk --resize-table=128 -a 1 \
-n 1:34:545         -c 1:fsbl1 \
-n 2:546:1057               -c 2:fsbl2 \
-n 3:1058:5153              -c 3:ssbl \
-n 4:5154:              -c 4:rootfs \
-p /dev/<SD card dev>
```

With other partition for kernel one partition rootfs for kernel.

c) copy the FSBL (2 times) and SSBL file on the correct partition. in this example in partition 1 to 3

for basic boot mode : <SD card dev> = /dev/mmcblk0:

```
# dd if=u-boot-spl.stm32 of=/dev/mmcblk0p1
# dd if=u-boot-spl.stm32 of=/dev/mmcblk0p2
# dd if=u-boot.img of=/dev/mmcblk0p3
```

for trusted boot mode:

```
# dd if=tf-a.stm32 of=/dev/mmcblk0p1
# dd if=tf-a.stm32 of=/dev/mmcblk0p2
# dd if=u-boot.stm32 of=/dev/mmcblk0p3
```

To boot from SD card, select BootPinMode = 1 0 1 and reset.

Prepare eMMC

You can use U-Boot to copy binary in eMMC.

In the next example, you need to boot from SD card and the images (u-boot-spl.stm32, u-boot.img) are presents on SD card (mmc 0) in ext4 partition 4 (bootfs).

To boot from SD card, select BootPinMode = 1 0 1 and reset.

Then you update the eMMC with the next U-Boot command :

a) prepare GPT on eMMC, example with 2 partitions, bootfs and roots:

```
# setenv emmc_part "name=ssbl,size=2MiB;name=bootfs,type=linux,bootable,size=64MiB;
↪name=rootfs,type=linux,size=512"
# gpt write mmc 1 ${emmc_part}
```

b) copy SPL on eMMC on firts boot partition (SPL max size is 256kB, with LBA 512, 0x200):

```
# ext4load mmc 0:4 0xC0000000 u-boot-spl.stm32
# mmc dev 1
# mmc partconf 1 1 1 1
```

(continues on next page)

(continued from previous page)

```
# mmc write ${fileaddr} 0 200
# mmc partconf 1 1 1 0
```

c) copy U-Boot in first GPT partition of eMMC:

```
# ext4load mmc 0:4 0xC0000000 u-boo t.img
# mmc dev 1
# part start mmc 1 1 partstart
# mmc write ${fileaddr} ${partstart} ${filesize}
```

To boot from eMMC, select BootPinMode = 0 1 0 and reset.

MAC Address

Please read doc/README.enetaddr for the implementation guidelines for mac id usage. Basically, environment has precedence over board specific storage.

For STMicroelectonics board, it is retrieved in STM32MP15x OTP :

- OTP_57[31:0] = MAC_ADDR[31:0]
- OTP_58[15:0] = MAC_ADDR[47:32]

To program a MAC address on virgin OTP words above, you can use the fuse command on bank 0 to access to internal OTP and lock them:

Prerequisite: check if a MAC address isn't yet programmed in OTP

1) check OTP: their value must be equal to 0:

```
STM32MP> fuse sense 0 57 2
Sensing bank 0:
Word 0x00000039: 00000000 00000000
```

2) check environment variable:

```
STM32MP> env print ethaddr
## Error: "ethaddr" not defined
```

3) check lock status of fuse 57 & 58 (at 0x39, 0=unlocked, 1=locked):

```
STM32MP> fuse sense 0 0x10000039 2
Sensing bank 0:
   Word 0x10000039: 00000000 00000000
```

Example to set mac address "12:34:56:78:9a:bc"

1) Write OTP:

```
STM32MP> fuse prog -y 0 57 0x78563412 0x0000bc9a
```

2) Read OTP:

```
STM32MP> fuse sense 0 57 2
Sensing bank 0:
Word 0x00000039: 78563412 0000bc9a
```

3) Lock OTP:

```
STM32MP> fuse prog 0 0x10000039 1 1

STM32MP> fuse sense 0 0x10000039 2
Sensing bank 0:
   Word 0x10000039: 00000001 00000001
```

4) next REBOOT, in the trace:

```
### Setting environment from OTP MAC address = "12:34:56:78:9a:bc"
```

5) check env update:

```
STM32MP> env print ethaddr
ethaddr=12:34:56:78:9a:bc
```

Warning: This command can't be executed twice on the same board as OTP are protected. It is already done for the board provided by STMicroelectronics.

Coprocessor firmware

U-Boot can boot the coprocessor before the kernel (coprocessor early boot).

a) Manuallly by using rproc commands (update the bootcmd)

   Configurations:

```
# env set name_copro "rproc-m4-fw.elf"
# env set dev_copro 0
# env set loadaddr_copro 0xC1000000
```

   Load binary from bootfs partition (number 4) on SD card (mmc 0):

```
# ext4load mmc 0:4 ${loadaddr_copro} ${name_copro}
```

   => ${filesize} variable is updated with the size of the loaded file.

   Start M4 firmware with remote proc command:

```
# rproc init
# rproc load ${dev_copro} ${loadaddr_copro} ${filesize}
# rproc start ${dev_copro}"00270033
```

b) Automatically by using FIT feature and generic DISTRO bootcmd

   see examples in the board stm32mp1 directory: fit_copro_kernel_dtb.its

   Generate FIT including kernel + device tree + M4 firmware with cfg with M4 boot:

```
$> mkimage -f fit_copro_kernel_dtb.its fit_copro_kernel_dtb.itb
```

   Then using DISTRO configuration file: see extlinux.conf to select the correct configuration:

   - stm32mp157c-ev1-m4
   - stm32mp157c-dk2-m4

DFU support

The DFU is supported on ST board.

The env variable dfu_alt_info is automatically build, and all the memory present on the ST boards are exported.

The dfu mode is started by the command:

```
STM32MP> dfu 0
```

On EV1 board, booting from SD card, without OP-TEE:

```
STM32MP> dfu 0 list
DFU alt settings list:
dev: RAM alt: 0 name: uImage layout: RAM_ADDR
dev: RAM alt: 1 name: devicetree.dtb layout: RAM_ADDR
dev: RAM alt: 2 name: uramdisk.image.gz layout: RAM_ADDR
dev: eMMC alt: 3 name: mmc0_fsbl1 layout: RAW_ADDR
dev: eMMC alt: 4 name: mmc0_fsbl2 layout: RAW_ADDR
dev: eMMC alt: 5 name: mmc0_ssbl layout: RAW_ADDR
dev: eMMC alt: 6 name: mmc0_bootfs layout: RAW_ADDR
dev: eMMC alt: 7 name: mmc0_vendorfs layout: RAW_ADDR
dev: eMMC alt: 8 name: mmc0_rootfs layout: RAW_ADDR
dev: eMMC alt: 9 name: mmc0_userfs layout: RAW_ADDR
dev: eMMC alt: 10 name: mmc1_boot1 layout: RAW_ADDR
dev: eMMC alt: 11 name: mmc1_boot2 layout: RAW_ADDR
dev: eMMC alt: 12 name: mmc1_ssbl layout: RAW_ADDR
dev: eMMC alt: 13 name: mmc1_bootfs layout: RAW_ADDR
dev: eMMC alt: 14 name: mmc1_vendorfs layout: RAW_ADDR
dev: eMMC alt: 15 name: mmc1_rootfs layout: RAW_ADDR
dev: eMMC alt: 16 name: mmc1_userfs layout: RAW_ADDR
dev: MTD alt: 17 name: nor0 layout: RAW_ADDR
dev: MTD alt: 18 name: nand0 layout: RAW_ADDR
dev: VIRT alt: 19 name: OTP layout: RAW_ADDR
dev: VIRT alt: 20 name: PMIC layout: RAW_ADDR
```

All the supported device are exported for dfu-util tool:

```
$> dfu-util -l
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=20, name="PMIC", serial=
↪"002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=19, name="OTP", serial=
↪"002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=18, name="nand0",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=17, name="nor0", serial=
↪"002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=16, name="mmc1_userfs",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=15, name="mmc1_rootfs",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=14, name="mmc1_vendorfs
↪", serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=13, name="mmc1_bootfs",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=12, name="mmc1_ssbl",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=11, name="mmc1_boot2",␣
↪serial="002700333338511934383330"
```

(continues on next page)

(continued from previous page)

```
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=10, name="mmc1_boot1",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=9, name="mmc0_userfs",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=8, name="mmc0_rootfs",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=7, name="mmc0_vendorfs",
↪ serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=6, name="mmc0_bootfs",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=5, name="mmc0_ssbl",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=4, name="mmc0_fsbl2",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=3, name="mmc0_fsbl1",␣
↪serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=2, name="uramdisk.image.
↪gz", serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=1, name="devicetree.dtb
↪", serial="002700333338511934383330"
Found DFU: [0483:df11] ver=9999, devnum=99, cfg=1, intf=0, alt=0, name="uImage",␣
↪serial="002700333338511934383330"
```

You can update the boot device:

- SD card (mmc0)

```
$> dfu-util -d 0483:5720 -a 3 -D tf-a-stm32mp157c-ev1-trusted.stm32
$> dfu-util -d 0483:5720 -a 4 -D tf-a-stm32mp157c-ev1-trusted.stm32
$> dfu-util -d 0483:5720 -a 5 -D u-boot-stm32mp157c-ev1-trusted.img
$> dfu-util -d 0483:5720 -a 6 -D st-image-bootfs-openstlinux-weston-stm32mp1.ext4
$> dfu-util -d 0483:5720 -a 7 -D st-image-vendorfs-openstlinux-weston-stm32mp1.ext4
$> dfu-util -d 0483:5720 -a 8 -D st-image-weston-openstlinux-weston-stm32mp1.ext4
$> dfu-util -d 0483:5720 -a 9 -D st-image-userfs-openstlinux-weston-stm32mp1.ext4
```

- EMMC (mmc1):

```
$> dfu-util -d 0483:5720 -a 10 -D tf-a-stm32mp157c-ev1-trusted.stm32
$> dfu-util -d 0483:5720 -a 11 -D tf-a-stm32mp157c-ev1-trusted.stm32
$> dfu-util -d 0483:5720 -a 12 -D u-boot-stm32mp157c-ev1-trusted.img
$> dfu-util -d 0483:5720 -a 13 -D st-image-bootfs-openstlinux-weston-stm32mp1.ext4
$> dfu-util -d 0483:5720 -a 14 -D st-image-vendorfs-openstlinux-weston-stm32mp1.
↪ext4
$> dfu-util -d 0483:5720 -a 15 -D st-image-weston-openstlinux-weston-stm32mp1.ext4
$> dfu-util -d 0483:5720 -a 16 -D st-image-userfs-openstlinux-weston-stm32mp1.ext4
```

- you can also dump the OTP and the PMIC NVM with:

```
$> dfu-util -d 0483:5720 -a 19 -U otp.bin
$> dfu-util -d 0483:5720 -a 20 -U pmic.bin
```

When the board is booting for nor0 or nand0, only the MTD partition on the boot devices are available, for example:

- NOR (nor0 = alt 20) & NAND (nand0 = alt 26)

```
$> dfu-util -d 0483:5720 -a 21 -D tf-a-stm32mp157c-ev1-trusted.stm32
$> dfu-util -d 0483:5720 -a 22 -D tf-a-stm32mp157c-ev1-trusted.stm32
```

(continues on next page)

(continued from previous page)

```
$> dfu-util -d 0483:5720 -a 23 -D u-boot-stm32mp157c-ev1-trusted.img
$> dfu-util -d 0483:5720 -a 27 -D st-image-weston-openstlinux-weston-stm32mp1_nand_
↪4_256_multivolume.ubi
```

- NAND (nand0 = alt 21):

```
$> dfu-util -d 0483:5720 -a 22 -D tf-a-stm32mp157c-ev1-trusted.stm32
$> dfu-util -d 0483:5720 -a 23 -D u-boot-stm32mp157c-ev1-trusted.img
$> dfu-util -d 0483:5720 -a 24 -D u-boot-stm32mp157c-ev1-trusted.img
$> dfu-util -d 0483:5720 -a 25 -D st-image-weston-openstlinux-weston-stm32mp1_nand_
↪4_256_multivolume.ubi
```

7.1.16 TBS

TBS2910 Matrix ARM miniPC

Building

To build u-boot for the TBS2910 Matrix ARM miniPC, you can use the following procedure:

First add the ARM toolchain to your PATH

Then setup the ARCH and cross compilation environment variables.

When this is done you can then build u-boot for the TBS2910 Matrix ARM miniPC with the following commands:

```
make mrproper
make tbs2910_defconfig
make
```

Once the build is complete, you can find the resulting image as u-boot.imx in the current directory.

UART

The UART voltage is at 3.3V and its settings are 115200bps 8N1

BOOT/UPDATE boot switch:

The BOOT/UPDATE switch (SW11) is connected to the BOOT_MODE0 and BOOT_MODE1 SoC pins. It has “BOOT” and “UPDATE” markings both on the PCB and on the plastic case.

When set to the “UPDATE” position, the SoC will use the “Boot From Fuses” configuration, and since BT_FUSE_SEL is 0, this makes the SOC jump to serial downloader.

When set in the “BOOT” position, the SoC will use the “Internal boot” configuration, and since BT_FUSE_SEL is 0, it will then use the GPIO pins for the boot configuration.

SW6 binary DIP switch array on the PCB revision 2.1:

On that PCB revision, SW6 has 8 positions.

Switching a position to ON sets the corresponding register to 1.

See the following table for a correspondence between the switch positions and registers:

Switch position	Register
1	BOOT_CFG2[3]
2	BOOT_CFG2[4]
3	BOOT_CFG2[5]
4	BOOT_CFG2[6]
5	BOOT_CFG1[4]
6	BOOT_CFG1[5]
7	BOOT_CFG1[6]
8	BOOT_CFG1[7]

For example:

- To boot from the eMMC: 1:ON , 2:ON, 3:ON, 4:OFF, 5:OFF, 6:ON, 7:ON, 8:OFF
- To boot from the microSD slot: 1: ON, 2: OFF, 3: OFF, 4: OFF, 5:OFF, 6:OFF, 7:ON, 8:OFF
- To boot from the SD slot: 1: OFF, 2: ON, 3: OFF, 4: OFF, 5:OFF, 6:OFF, 7:ON, 8:OFF
- To boot from SATA: 1: OFF, 2: OFF, 3: OFF, 4: OFF, 5:OFF, 6:ON, 7:OFF, 8:OFF

You can refer to the BOOT_CFG registers in the I.MX6Q reference manual for additional details.

SW6 binary DIP switch array on the PCB revision 2.3:

On that PCB revision, SW6 has only 4 positions.

Switching a position to ON sets the corresponding register to 1.

See the following table for a correspondence between the switch positions and registers:

Switch position	Register
1	BOOT_CFG2[3]
2	BOOT_CFG2[4]
3	BOOT_CFG2[5]
4	BOOT_CFG1[5]

For example:

- To boot from the eMMC: 1:ON, 2:ON, 3:ON, 4:ON
- To boot from the microSD slot: 1:ON, 2:OFF, 3:OFF, 4:OFF
- To boot from the SD slot: 1:OFF, 2:ON, 3:OFF, 4:OFF

You can refer to the BOOT_CFG registers in the I.MX6Q reference manual for additional details.

Loading u-boot from USB:

If you need to load u-boot from USB, you can use the following instructions:

First build imx_usb_loader, as we will need it to load u-boot from USB. This can be done with the following commands:

```
git clone git://github.com/boundarydevices/imx_usb_loader.git
cd imx_usb_loader
make
```

This will create the resulting imx_usb binary.

When this is done, you can copy the u-boot.imx image that you built earlier in in the imx_usb_loader directory.

You will then need to power off the TBS2910 Matrix ARM miniPC and make sure that the boot switch is set to “UPDATE”

Once this is done you can connect an USB cable between the computer that will run imx_usb and the TBS2910 Matrix ARM miniPC.

If you also need to access the u-boot console, you will also need to connect an UART cable between the computer running imx_usb and the TBS2910 Matrix ARM miniPC.

Once everything is connected you can finally power on the TBS2910 Matrix ARM miniPC. The SoC will then jump to the serial download and wait for you.

Finlay, you can load u-boot through USB with with the following command:

```
sudo ./imx_usb -v u-boot.imx
```

The u-boot boot messages will then appear in the serial console.

Install u-boot on the eMMC:

To install u-boot on the eMMC, you first need to boot the TBS2910 Matrix ARM miniPC.

Once booted, you can flash u-boot.imx to mmcblk0boot0 with the following commands:

```
sudo echo 0 >/sys/block/mmcblk0boot0/force_ro
sudo dd if=u-boot.imx of=/dev/mmcblk0boot0 bs=1k seek=1; sync
```

Note that the eMMC card node may vary, so adjust this as needed.

Once the new u-boot version is installed, to boot on it you then need to power off the TBS2910 Matrix ARM miniPC.

Once it is off, you need make sure that the boot switch is set to “BOOT” and that the SW6 switch is set to boot on the eMMC as described in the previous sections.

If you also need to access the u-boot console, you will also need to connect an UART cable between the computer running imx_usb and the TBS2910 Matrix ARM miniPC.

You can then power up the TBS2910 Matrix ARM miniPC and U-Boot messages will appear in the serial console.

Booting a distribution:

When booting on the TBS2910 Matrix ARM miniPC, by default U-Boot will first try to boot from hardcoded offsets from the start of the eMMC. This is for compatibility with the stock GNU/Linux distribution.

If that fails it will then try to boot from several interfaces using ‘distro_bootcmd’: It will first try to boot from the microSD slot, then the SD slot, then the internal eMMC, then the SATA interface and finally the USB interface. For more information on how to configure your distribution to boot, see ‘README.distro’.

Links:

- https://www.tbsdtv.com/download/document/tbs2910/TBS2910-Matrix-ARM-mini-PC-SCH_rev2.1.pdf - The schematics for the revision 2.1 of the TBS2910 Matrix ARM miniPC.
- https://cache.freescale.com/files/32bit/doc/ref_manual/IMX6DQRM.pdf - The SoC reference manual for additional details on the BOOT_CFG registers.

7.1.17 Toradex

Apalis iMX8QM V1.0B Module

Quick Start

- Build the ARM trusted firmware binary
- Get scfw_tcm.bin and ahab-container.img
- Build U-Boot
- Load U-Boot binary using uuu
- Flash U-Boot binary into the eMMC
- Boot

Get and Build the ARM Trusted Firmware

```
$ git clone -b imx_4.14.78_1.0.0_ga https://source.codeaurora.org/external/imx/imx-atf
$ cd imx-atf/
$ make PLAT=imx8qm bl31
```

Get scfw_tcm.bin and ahab-container.img

```
$ wget https://github.com/toradex/meta-fsl-bsp-release/blob/toradex-sumo-4.14.78-1.0.0_
↪ga-bringup/imx/meta-bsp/recipes-
  bsp/imx-sc-firmware/files/mx8qm-apalis-scfw-tcm.bin?raw=true
$ mv mx8qm-apalis-scfw-tcm.bin\?raw\=true mx8qm-apalis-scfw-tcm.bin
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.0.bin
$ chmod +x firmware-imx-8.0.bin
$ ./firmware-imx-8.0.bin
```

Copy the following binaries to the U-Boot folder:

```
$ cp imx-atf/build/imx8qm/release/bl31.bin .
$ cp u-boot/u-boot.bin .
```

Copy the following firmware to the U-Boot folder:

```
$ cp firmware-imx-8.0/firmware/seco/ahab-container.img .
```

Build U-Boot

```
$ make apalis-imx8_defconfig
$ make u-boot-dtb.imx
```

Load the U-Boot Binary Using UUU

Get the latest version of the universal update utility (uuu) aka `mfgtools 3.0`:

https://community.nxp.com/external-link.jspa?url=https%3A%2F%2Fgithub.com%2FNXPmicro%2Fmfgtools%2Freleases

Put the module into USB recovery aka serial downloader mode, connect USB device to your host and execute uuu:

```
sudo ./uuu u-boot/u-boot-dtb.imx
```

Flash the U-Boot Binary into the eMMC

Burn the `u-boot-dtb.imx` binary to the primary eMMC hardware boot area partition and boot:

```
load mmc 1:1 $loadaddr u-boot-dtb.imx
setexpr blkcnt ${filesize} + 0x1ff && setexpr blkcnt ${blkcnt} / 0x200
mmc dev 0 1
mmc write ${loadaddr} 0x0 ${blkcnt}
```

Colibri iMX7

Quick Start

- Build U-Boot
- NAND IMX image adjustments before flashing
- Flashing manually U-Boot to eMMC
- Flashing manually U-Boot to NAND
- Using `update_uboot` script

Build U-Boot

```
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make colibri_imx7_emmc_defconfig # For NAND: colibri_imx7_defconfig
$ make
```

After build succeeds, you will obtain final `u-boot-dtb.imx` IMX specific image, ready for flashing (but check next section for additional adjustments).

Final IMX program image includes (section `6.6.7` from IMX7DRM):

- **Image vector table** (IVT) for BootROM
- **Boot data** -indicates the program image location, program image size in bytes, and the plugin flag.
- **Device configuration data**
- **User image**: U-Boot image (`u-boot-dtb.bin`)

IMX image adjustments prior to flashing

1. U-Boot for both Colibri iMX7 NAND and eMMC versions is built with HABv4 support (AN4581.pdf) enabled by default, which requires to generate a proper Command Sequence File (CSF) by srktool from NXP (not included in the U-Boot tree, check additional details in introduction_habv4.txt) and concatenate it to the final `u-boot-dtb.imx`.

2. In case if you don't want to generate a proper CSF (for any reason), you still need to pad the IMX image so i has the same size as specified in in **Boot Data** section of IMX image. To obtain this value, run:

```
$ od -X -N 0x30 u-boot-dtb.imx
0000000    402000d1 87800000 00000000 877ff42c
0000020    877ff420 877ff400 878a5000 00000000
                    ^^^^^^^^
0000040    877ff000 000a8060 00000000 40b401d2
           ^^^^^^^^ ^^^^^^^^
```

Where:

- `877ff400` - IVT self address
- `877ff000` - Program image address
- `000a8060` - Program image size

To calculate the padding:

- IVT offset = `0x877ff400` - `0x877ff000` = `0x400`
- Program image size = `0xa8060` - `0x400` = `0xa7c60`

and then pad the image:

```
$ objcopy -I binary -O binary --pad-to 0xa7c60 --gap-fill=0x00 \
    u-boot-dtb.imx u-boot-dtb.imx.zero-padded
```

3. Also, according to requirement from `6.6.7.1`, the final image should have `0x400` offset for initial IVT table.

For eMMC setup we handle this by flashing it to `0x400`, howewer for NAND setup we adjust the image prior to flashing, adding padding in the beginning of the image.

```
$ dd if=u-boot-dtb.imx.zero-padded of=u-boot-dtb.imx.ready bs=1024 seek=1
```

Flash U-Boot IMX image to eMMC

Flash the `u-boot-dtb.imx.zero-padded` binary to the primary eMMC hardware boot area partition:

```
=> load mmc 1:1 $loadaddr u-boot-dtb.imx.zero-padded
=> setexpr blkcnt ${filesize} + 0x1ff && setexpr blkcnt ${blkcnt} / 0x200
=> mmc dev 0 1
=> mmc write ${loadaddr} 0x2 ${blkcnt}
```

Flash U-Boot IMX image to NAND

```
=> load mmc 1:1 $loadaddr u-boot-dtb.imx.ready
=> nand erase.part u-boot1
=> nand write ${loadaddr} u-boot1 ${filesize}
=> nand erase.part u-boot2
=> nand write ${loadaddr} u-boot2 ${filesize}
```

Using update_uboot script

You can also usb U-Boot env update_uboot script, which wraps all eMMC/NAND specific command invocation:

```
=> load mmc 1:1 $loadaddr u-boot-dtb.imx.ready
=> run update_uboot
```

Colibri iMX8QXP V1.0B Module

Quick Start

- Build the ARM trusted firmware binary
- Get scfw_tcm.bin and ahab-container.img
- Build U-Boot
- Load U-Boot binary using uuu
- Flash U-Boot binary into the eMMC
- Boot

Get and Build the ARM Trusted Firmware

```
$ git clone -b imx_4.14.78_1.0.0_ga https://source.codeaurora.org/external/imx/imx-atf
$ cd imx-atf/
$ make PLAT=imx8qxp bl31
```

Get scfw_tcm.bin and ahab-container.img

```
$ wget https://github.com/toradex/meta-fsl-bsp-release/blob/
       toradex-sumo-4.14.78-1.0.0_ga-bringup/imx/meta-bsp/recipes-
       bsp/imx-sc-firmware/files/mx8qx-colibri-scfw-tcm.bin?raw=true
$ mv mx8qx-colibri-scfw-tcm.bin\?raw\=true mx8qx-colibri-scfw-tcm.bin
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.0.bin
$ chmod +x firmware-imx-8.0.bin
$ ./firmware-imx-8.0.bin
```

Copy the following binaries to the U-Boot folder:

```
$ cp imx-atf/build/imx8qxp/release/bl31.bin .
$ cp u-boot/u-boot.bin .
```

Copy the following firmware to the U-Boot folder:

```
$ cp firmware-imx-8.0/firmware/seco/ahab-container.img .
```

Build U-Boot

```
$ make colibri-imx8x_defconfig
$ make u-boot-dtb.imx
```

Load the U-Boot Binary Using UUU

Get the latest version of the universal update utility (uuu) aka `mfgtools 3.0`:

https://community.nxp.com/external-link.jspa?url=https%3A%2F%2Fgithub.com%2FNXPmicro%2Fmfgtools%2Freleases

Put the module into USB recovery aka serial downloader mode, connect USB device to your host and execute uuu:

```
sudo ./uuu u-boot/u-boot-dtb.imx
```

Flash the U-Boot Binary into the eMMC

Burn the `u-boot-dtb.imx` binary to the primary eMMC hardware boot area partition:

```
load mmc 1:1 $loadaddr u-boot-dtb.imx
setexpr blkcnt ${filesize} + 0x1ff && setexpr blkcnt ${blkcnt} / 0x200
mmc dev 0 1
mmc write ${loadaddr} 0x0 ${blkcnt}
```

Verdin iMX8M Mini Module

Quick Start

- Build the ARM trusted firmware binary
- Get the DDR firmware
- Build U-Boot
- Flash to eMMC
- Boot

Get and Build the ARM Trusted Firmware (Trusted Firmware A)

```
$ echo "Downloading and building TF-A..."
$ git clone https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git
$ cd trusted-firmware-a
```

Then build ATF (TF-A):

```
$ make PLAT=imx8mm IMX_BOOT_UART_BASE=0x30860000 bl31
$ cp build/imx8mm/release/bl31.bin ../
```

Get the DDR Firmware

```
$ cd ..
$ wget https://www.nxp.com/lgfiles/NMG/MAD/YOCTO/firmware-imx-8.4.1.bin
$ chmod +x firmware-imx-8.4.1.bin
$ ./firmware-imx-8.4.1.bin
$ cp firmware-imx-8.4.1/firmware/ddr/synopsys/lpddr4*.bin ./
```

Build U-Boot

```
$ export CROSS_COMPILE=aarch64-linux-gnu-
$ export ATF_LOAD_ADDR=0x920000
$ make verdin-imx8mm_defconfig
$ make flash.bin
```

Flash to eMMC

```
> tftpboot ${loadaddr} flash.bin
> setexpr blkcnt ${filesize} + 0x1ff && setexpr blkcnt ${blkcnt} / 0x200
> mmc dev 0 1 && mmc write ${loadaddr} 0x2 ${blkcnt}
```

As a convenience, instead of the last two commands one may also use the update U-Boot wrapper:

```
> run update_uboot
```

Boot

ATF, U-Boot proper and u-boot.dtb images are packed into FIT image, which is loaded and parsed by SPL.

Boot sequence is:

- SPL —> ATF (TF-A) —> U-Boot proper

Output:

```
U-Boot SPL 2020.01-00187-gd411d164e5 (Jan 26 2020 - 04:47:26 +0100)
Normal Boot
Trying to boot from MMC1

U-Boot 2020.01-00187-gd411d164e5 (Jan 26 2020 - 04:47:26 +0100)

CPU:   Freescale i.MX8MMQ rev1.0 at 0 MHz
Reset cause: POR
DRAM:  2 GiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1, FSL_SDHC: 2
Loading Environment from MMC... OK
In:    serial
Out:   serial
Err:   serial
Model: Toradex Verdin iMX8M Mini Quad 2GB Wi-Fi / BT IT V1.0A, Serial:
Net:   eth0: ethernet@30be0000
Hit any key to stop autoboot:  0
Verdin iMX8MM #
```

7.1.18 XenGuestARM64

Xen guest ARM64 board

This board specification

This board is to be run as a virtual Xen [1] guest with U-boot as its primary bootloader. Xen is a type 1 hypervisor that allows multiple operating systems to run simultaneously on a single physical server. Xen is capable of running virtual machines in both full virtualization and para-virtualization (PV) modes. Xen runs virtual machines, which are called “domains”.

Paravirtualized drivers are a special type of device drivers that are used in a guest system in the Xen domain and perform I/O operations using a special interface provided by the virtualization system and the host system.

Xen support for U-boot is implemented by introducing a new Xen guest ARM64 board and porting essential drivers from MiniOS [3] as well as some of the work previously done by NXP [4]:

- PV block device frontend driver with XenStore based device enumeration and UCLASS_PVBLOCK class;
- PV serial console device frontend driver;
- Xen hypervisor support with minimal set of the essential headers adapted from the Linux kernel;
- Xen grant table support;
- Xen event channel support in polling mode;
- XenBus support;
- dynamic RAM size as defined in the device tree instead of the statically defined values;
- position-independent pre-relocation code is used as we cannot statically define any start addresses at compile time which is up to Xen to choose at run-time;
- new defconfig introduced: xenguest_arm64_defconfig.

Board limitations

1. U-boot runs without MMU enabled at the early stages. According to Xen on ARM ABI (xen/include/public/arch-arm.h): all memory which is shared with other entities in the system (including the hypervisor and other guests) must reside in memory which is mapped as Normal Inner Write-Back Outer Write-Back Inner-Shareable. Thus, page attributes must be equally set for all the entities working with that page. Before MMU is set up the data cache is turned off and pages are seen by the vCPU and Xen in different ways - cacheable by Xen and non-cacheable by vCPU. So it means that manual data cache maintenance is required at the early stages.
2. No serial console until MMU is up. Because data cache maintenance is required until the MMU setup the early/debug serial console is not implemented. Therefore, we do not have usual prints like U-boot’s banner etc. until the serial driver is initialized.
3. Single RAM bank supported. If a Xen guest is given much memory it is possible that Xen allocates two memory banks for it. The first one is allocated under 4GB address space and in some cases may represent the whole guest’s memory. It is assumed that U-boot most likely won’t require high memory bank for its work andlaunching OS, so it is enough to take the first one.

Board default configuration

One can select the configuration as follows:

- make xenguest_arm64_defconfig

[1] - https://xenproject.org/

[2] - https://wiki.xenproject.org/wiki/Paravirtualization_(PV)

[3] - https://wiki.xenproject.org/wiki/Mini-OS

[4] - https://source.codeaurora.org/external/imx/uboot-imx/tree/?h=imx_v2018.03_4.14.98_2.0.0_ga

7.1.19 Xilinx

U-Boot device tree bindings

All the device tree bindings used in U-Boot are specified in Linux kernel. Please refer dt bindings from below specified paths in Linux kernel.

- **ata**
  - Documentation/devicetree/bindings/ata/ahci-ceva.txt
- **clock**
  - Documentation/devicetree/bindings/clock/xlnx,zynqmp-clk.txt
- **firmware**
  - Documentation/devicetree/bindings/firmware/xilinx/xlnx,zynqmp-firmware.txt
- **fpga**
  - Documentation/devicetree/bindings/fpga/xlnx,zynqmp-pcap-fpga.txt
- **gpio**
  - Documentation/devicetree/bindings/gpio/gpio-xilinx.txt
  - Documentation/devicetree/bindings/gpio/gpio-zynq.txt
- **i2c**
  - Documentation/devicetree/bindings/i2c/xlnx,xps-iic-2.00.a.yaml
  - Documentation/devicetree/bindings/i2c/cdns,i2c-r1p10.yaml
- **mmc**
  - Documentation/devicetree/bindings/mmc/arasan,sdhci.yaml
- **net**
  - Documentation/devicetree/bindings/net/macb.txt
  - Documentation/devicetree/bindings/net/xilinx_axienet.txt
  - Documentation/devicetree/bindings/net/xilinx_emaclite.txt
- **nvmem**
  - Documentation/devicetree/bindings/nvmem/xlnx,zynqmp-nvmem.txt
- **power**
  - Documentation/devicetree/bindings/power/reset/xlnx,zynqmp-power.txt
- **serial**
  - Documentation/devicetree/bindings/serial/cdns,uart.txt
  - Documentation/devicetree/bindings/serial/xlnx,opb-uartlite.txt
- **spi**
  - Documentation/devicetree/bindings/spi/spi-cadence.txt
  - Documentation/devicetree/bindings/spi/spi-xilinx.txt
  - Documentation/devicetree/bindings/spi/spi-zynqmp-qspi.txt
  - Documentation/devicetree/bindings/spi/spi-zynq-qspi.txt
- **usb**
  - Documentation/devicetree/bindings/usb/dwc3-xilinx.txt

  - Documentation/devicetree/bindings/usb/dwc3.txt
  - Documentation/devicetree/bindings/usb/ci-hdrc-usb2.txt
- **wdt**
  - Documentation/devicetree/bindings/watchdog/of-xilinx-wdt.txt

ZYNQ

About this

This document describes the information about Xilinx Zynq U-Boot - like supported boards, ML status and TODO list.

Zynq boards

Xilinx Zynq-7000 All Programmable SoCs enable extensive system level differentiation, integration, and flexibility through hardware, software, and I/O programmability.

- zc702 (single qspi, gem0, mmc) [1]
- zc706 (dual parallel qspi, gem0, mmc) [2]
- zed (single qspi, gem0, mmc) [3]
- microzed (single qspi, gem0, mmc) [4]
- **zc770**
  - zc770-xm010 (single qspi, gem0, mmc)
  - zc770-xm011 (8 or 16 bit nand)
  - zc770-xm012 (nor)
  - zc770-xm013 (dual parallel qspi, gem1)

Building

configure and build for zc702 board:

```
$ export DEVICE_TREE=zynq-zc702
$ make xilinx_zynq_virt_defconfig
$ make
```

Bootmode

Zynq has a facility to read the bootmode from the slcr bootmode register once user is setting through jumpers on the board - see page no:1546 on [5]

All possible bootmode values are defined in Table 6-2:Boot_Mode MIO Pins on [5].

board_late_init() will read the bootmode values using slcr bootmode register at runtime and assign the modeboot variable to specific bootmode string which is intern used in autoboot.

SLCR bootmode register Bit[3:0] values

```
#define ZYNQ_BM_NOR          0x02
#define ZYNQ_BM_SD           0x05
#define ZYNQ_BM_JTAG         0x0
```

"modeboot" variable can assign any of "norboot", "sdboot" or "jtagboot" bootmode strings at runtime.

Flashing

SD Card

To write an image that boots from a SD card first create a FAT32 partition and a FAT32 filesystem on the SD card:

```
sudo fdisk /dev/sdx
sudo mkfs.vfat -F 32 /dev/sdx1
```

Mount the SD card and copy the SPL and U-Boot to the root directory of the SD card:

```
sudo mount -t vfat /dev/sdx1 /mnt
sudo cp spl/boot.bin /mnt
sudo cp u-boot.img /mnt
```

Mainline status

- Added basic board configurations support.
- Added zynq u-boot bsp code - arch/arm/mach-zynq
- Added zynq boards named - zc70x, zed, microzed, zc770_xm010/xm011/xm012/xm013
- Added zynq drivers:

  serial drivers/serial/serial_zynq.c

  net drivers/net/zynq_gem.c

  mmc drivers/mmc/zynq_sdhci.c

  spi drivers/spi/zynq_spi.c

  qspi drivers/spi/zynq_qspi.c

  i2c drivers/i2c/zynq_i2c.c

  nand drivers/mtd/nand/raw/zynq_nand.c
- Done proper cleanups on board configurations
- Added basic FDT support for zynq boards
- d-cache support for zynq_gem.c
- [1] http://www.xilinx.com/products/boards-and-kits/EK-Z7-ZC702-G.htm
- [2] http://www.xilinx.com/products/boards-and-kits/EK-Z7-ZC706-G.htm
- [3] http://zedboard.org/product/zedboard
- [4] http://zedboard.org/product/microzed
- [5] http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

ZYNQMP

About this

This document describes the information about Xilinx Zynq UltraScale+ MPSOC U-Boot support. Core support is available in arch/arm/mach-zynqmp folder.

ZynqMP boards

- zcu100 (ultra96 v1), zcu102, zcu104, zcu106 - Evaluation boards
- zc1232 - Characterization boards
- zcu111, zcu208, zcu216 - RFSOC evaluation boards
- zcu1254, zcu1275, zcu1285 - RFSOC characterization boards
- a2197 - System Controller on Versal boards
- mini - Mini U-Boot running out of OCM
- **zc1751 - Characterization Processor boards**
    - zc1751-xm015-dc1
    - zc1751-xm016-dc2
    - zc1751-xm017-dc3
    - zc1751-xm018-dc4
    - zc1751-xm019-dc5

Building

Configure and build for zcu102 board:

```
$ source arm64 toolchain
$ export DEVICE_TREE=zynqmp-zcu102-revA
$ make xilinx_zynqmp_virt_defconfig
$ make
```

U-Boot SPL flow

For getting U-Boot SPL flow up and running it is necessary to do some additional steps because booting device requires external images which are not the part of U-Boot repository.

PMU firmware

The Platform Management Unit (PMU) RAM can be loaded with a firmware (PMU Firmware) at run-time and can be used to extend or customize the functionality of PMU. The PMU firmware is the part of boot image (boot.bin) and it is automatically loaded by BootROM. boot.bin can be directly generated by mkimage tool as the part of make. If you want to create boot.bin with PMU Firmware include please point CONFIG_PMUFW_INIT_FILE to PMU firmware binary. For example::

```
CONFIG_PMUFW_INIT_FILE="<path>/pmu.bin"
```

If you see below message you need to load PMU Firmware:

```
PMUFW is not found - Please load it!
```

The second external blob is PMU Configuration object which is object which is passed from U-Boot SPL to PMU Firmware for initial system configuration. PMU configuration object is the part of U-Boot SPL image. For pointing to this object please use CONFIG_ZYNQMP_SPL_PM_CFG_OBJ_FILE symbol. For example::

```
CONFIG_ZYNQMP_SPL_PM_CFG_OBJ_FILE="<path>/pmu_obj.bin"
```

PMU configuration object

Object can be obtain in several ways. The easiest way is to take pm_cfg_obj.c from SDK/Vitis design and build it::

```
$ git clone https://github.com/Xilinx/embeddedsw.git
$ export EMBEDDED_SW=$PWD/embeddedsw
$ gcc -c pm_cfg_obj.c -I ${EMBEDDED_SW}/lib/bsp/standalone/src/common/ -I ${EMBEDDED_
↪SW}/lib/sw_services/xilpm/src/zynqmp/client/common/
$ objcopy -O binary pm_cfg_obj.o pmu_obj.bin
```

The second way is to use tools/zynqmp_pm_cfg_obj_convert.py. For more information about this tool please run it with -h parameter.

The third way is to extract it from Xilinx FSBL elf file. Object is starting at XPm_ConfigObject symbol.

Arm Trusted Firmware (ATF)

U-Boot itself can run from EL3 to EL1. Without ATF U-Boot runs in EL3. Boot flow is U-Boot SPL->U-Boot in EL3. When ATF is used U-Boot normally runs in EL2. Boot flow is U-Boot SPL->ATF->U-Boot in EL2. As the part of build process u-boot.itb is generated. When BL31 shell variable is present u-boot.itb is generated with ATF included. You can point to it by::

```
$ export BL31=<path>/bl31.bin
```

Flashing

SD Card

To write an image that boots from a SD card first create a FAT32 partition and a FAT32 filesystem on the SD card:

```
sudo fdisk /dev/sdx
sudo mkfs.vfat -F 32 /dev/sdx1
```

Mount the SD card and copy the SPL and U-Boot to the root directory of the SD card:

```
sudo mount -t vfat /dev/sdx1 /mnt
sudo cp spl/boot.bin /mnt
sudo cp u-boot.itb /mnt
```

ZYNQMP-R5

About this

This document describes the information about Xilinx Zynq UltraScale+ MPSOC U-Boot Cortex R5 support.

ZynqMP R5 boards

- zynqmp-r5 - U-Boot running on RPU Cortex-R5

Building

configure and build armv7 toolchain:

```
$ make xilinx_zynqmp_r5_defconfig
$ make
```

Notes

Output fragment is u-boot.

Loading

ZynqMP R5 U-Boot was created for supporting loading OS on RPU. There are two ways how to start U-Boot on R5.

Bootgen

The first way is to use Xilinx FSBL (First stage bootloader) to load u-boot and start it. The following bif can be used for boot image generation via Xilinx bootgen utility:

```
the_ROM_image:
{
      [bootloader,destination_cpu=r5-0] fsbl_rpu.elf
      [destination_cpu=r5-0]u-boot.elf
}
```

Bootgen command for building boot.bin:

```
bootgen -image <bif>.bif -r -w -o i boot.bin
```

U-Boot cpu command

The second way to load U-Boot to Cortex R5 is from U-Boot running on A53 as is visible from the following log:

```
U-Boot SPL 2020.10-rc4-00090-g801b3d5c5757 (Sep 15 2020 - 14:07:24 +0200)
PMUFW:        v1.1
Loading new PMUFW cfg obj (2024 bytes)
EL Level:     EL3
Multiboot:    0
Trying to boot from MMC2
spl: could not initialize mmc. error: -19
Trying to boot from MMC1
spl_load_image_fat_os: error reading image u-boot.bin, err - -2
NOTICE:  ATF running on XCZU7EG/EV/silicon v4/RTL5.1 at 0xfffea000
NOTICE:  BL31: v2.2(release):v2.2-614-ged9dc512fb9c
NOTICE:  BL31: Built : 09:32:09, Mar 13 2020


U-Boot 2020.10-rc4-00090-g801b3d5c5757 (Sep 15 2020 - 14:07:24 +0200)

Model: ZynqMP ZCU104 RevC
```

(continues on next page)

(continued from previous page)

```
Board: Xilinx ZynqMP
DRAM:  2 GiB
PMUFW:       v1.1
EL Level:    EL2
Chip ID:     zu7e
WDT:   Started with servicing (60s timeout)
NAND:  0 MiB
MMC:   mmc@ff170000: 0
Loading Environment from FAT... *** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Bootmode: LVL_SHFT_SD_MODE1
Reset reason: SOFT
Net:
ZYNQ GEM: ff0e0000, mdio bus ff0e0000, phyaddr 12, interface rgmii-id
eth0: ethernet@ff0e0000
Hit any key to stop autoboot:  0
ZynqMP> setenv autoload no
ZynqMP> dhcp
BOOTP broadcast 1
DHCP client bound to address 192.168.0.167 (8 ms)
ZynqMP> tftpboot 20000000 192.168.0.105:u-boot-r5-2.elf
Using ethernet@ff0e0000 device
TFTP from server 192.168.0.105; our IP address is 192.168.0.167
Filename 'u-boot-r5-2.elf'.
Load address: 0x20000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ################
         376 KiB/s
done
Bytes transferred = 2075464 (1fab48 hex)
ZynqMP> setenv autostart no
ZynqMP> bootelf -p 20000000
ZynqMP> cpu 4 release 10000000 lockstep
Using TCM jump trampoline for address 0x10000000
R5 lockstep mode
ZynqMP>
```

Then on second uart you can see U-Boot up and running on R5:

```
U-Boot 2020.10-rc4-00071-g7045622cc9ba (Sep 16 2020 - 13:38:53 +0200)

Model: Xilinx ZynqMP R5
DRAM:  512 MiB
MMC:
In:    serial@ff010000
Out:   serial@ff010000
Err:   serial@ff010000
Net:   No ethernet found.
```

(continues on next page)

(continued from previous page)

```
ZynqMP r5>
```

Please make sure MIO pins for uart are properly configured to see output.

CHAPTER

EIGHT

ANDROID-SPECIFIC DOC

These books provide information about booting the Android OS from U-Boot, manipulating Android images from U-Boot shell and discusses other Android-specific features available in U-Boot.

8.1 Android-specific doc

8.1.1 Android A/B updates

Overview

A/B system updates ensures modern approach for system update. This feature allows one to use two sets (or more) of partitions referred to as slots (normally slot A and slot B). The system runs from the current slot while the partitions in the unused slot can be updated[1].

A/B enablement

The A/B updates support can be activated by specifying next options in your board configuration file:

```
CONFIG_ANDROID_AB=y
CONFIG_CMD_AB_SELECT=y
```

The disk space on target device must be partitioned in a way so that each partition which needs to be updated has two or more instances. The name of each instance must be formed by adding suffixes: _a, _b, _c, etc. For example: `boot_a`, `boot_b`, `system_a`, `system_b`, `vendor_a`, `vendor_b`.

As a result you can use `ab_select` command to ensure A/B boot process in your boot script. This command analyzes and processes A/B metadata stored on a special partition (e.g. `misc`) and determines which slot should be used for booting up.

Command usage

```
ab_select <slot_var_name> <interface> <dev[:part_number|#part_name]>
```

for example:

```
=> ab_select slot_name mmc 1:4
```

or:

```
=> ab_select slot_name mmc 1#misc
```

Result:

[1] https://source.android.com/devices/tech/ota/ab

```
=> printenv slot_name
slot_name=a
```

Based on this slot information, the current boot partition should be defined, and next kernel command line parameters should be generated:

- `androidboot.slot_suffix=`
- `root=`

For example:

```
androidboot.slot_suffix=_a root=/dev/mmcblk1p12
```

A/B metadata is organized according to AOSP reference[2]. On the first system start with A/B enabled, when `misc` partition doesn't contain required data, the default A/B metadata will be created and written to `misc` partition.

References

8.1.2 Android Verified Boot 2.0

This file contains information about the current support of Android Verified Boot 2.0 in U-Boot.

Overview

Verified Boot establishes a chain of trust from the bootloader to system images:

- Provides integrity checking for:
  - Android Boot image: Linux kernel + ramdisk. RAW hashing of the whole partition is done and the hash is compared with the one stored in the VBMeta image
  - `system/vendor` partitions: verifying root hash of dm-verity hashtrees
- Provides capabilities for rollback protection

Integrity of the bootloader (U-Boot BLOB and environment) is out of scope.

For additional details check[1].

AVB using OP-TEE (optional)

If AVB is configured to use OP-TEE (see *Enable on your board*) rollback indexes and device lock state are stored in RPMB. The RPMB partition is managed by OP-TEE (see[2] for details) which is a secure OS leveraging ARM TrustZone.

AVB 2.0 U-Boot shell commands

Provides CLI interface to invoke AVB 2.0 verification + misc. commands for different testing purposes:

---

[2] https://android.googlesource.com/platform/bootable/recovery/+/refs/tags/android-10.0.0_r25/bootloader_message/include/bootloader_message/bootloader_message.h

[1] https://android.googlesource.com/platform/external/avb/+/master/README.md

[2] https://www.op-tee.org/

```
avb init <dev> - initialize avb 2.0 for <dev>
avb verify - run verification process using hash data from vbmeta structure
avb read_rb <num> - read rollback index at location <num>
avb write_rb <num> <rb> - write rollback index <rb> to <num>
avb is_unlocked - returns unlock status of the device
avb get_uuid <partname> - read and print uuid of partition <partname>
avb read_part <partname> <offset> <num> <addr> - read <num> bytes from
partition <partname> to buffer <addr>
avb write_part <partname> <offset> <num> <addr> - write <num> bytes to
<partname> by <offset> using data from <addr>
```

Partitions tampering (example)

Boot or system/vendor (dm-verity metadata section) is tampered:

```
=> avb init 1
=> avb verify
avb_slot_verify.c:175: ERROR: boot: Hash of data does not match digest in
descriptor.
Slot verification result: ERROR_IO
```

Vbmeta partition is tampered:

```
=> avb init 1
=> avb verify
avb_vbmeta_image.c:206: ERROR: Hash does not match!
avb_slot_verify.c:388: ERROR: vbmeta: Error verifying vbmeta image:
HASH_MISMATCH
Slot verification result: ERROR_IO
```

Enable on your board

The following options must be enabled:

```
CONFIG_LIBAVB=y
CONFIG_AVB_VERIFY=y
CONFIG_CMD_AVB=y
```

In addtion optionally if storing rollback indexes in RPMB with help of OP-TEE:

```
CONFIG_TEE=y
CONFIG_OPTEE=y
CONFIG_OPTEE_TA_AVB=y
CONFIG_SUPPORT_EMMC_RPMB=y
```

Then add `avb verify` invocation to your android boot sequence of commands, e.g.:

```
=> avb_verify=avb init $mmcdev; avb verify;
=> if run avb_verify; then                       \
        echo AVB verification OK. Continue boot; \
        set bootargs $bootargs $avb_bootargs;    \
   else                                          \
        echo AVB verification failed;            \
        exit;                                    \
   fi;                                           \
```

(continues on next page)

(continued from previous page)

```
=> emmc_android_boot=                                    \
       echo Trying to boot Android from eMMC ...;        \
       ...                                               \
       run avb_verify;                                   \
       mmc read ${fdtaddr} ${fdt_start} ${fdt_size};     \
       mmc read ${loadaddr} ${boot_start} ${boot_size}; \
            bootm $loadaddr $loadaddr $fdtaddr;          \
```

If partitions you want to verify are slotted (have A/B suffixes), then current slot suffix should be passed to `avb verify` sub-command, e.g.:

```
=> avb verify _a
```

To switch on automatic generation of vbmeta partition in AOSP build, add these lines to device configuration mk file:

```
BOARD_AVB_ENABLE := true
BOARD_AVB_ALGORITHM := SHA512_RSA4096
BOARD_BOOTIMAGE_PARTITION_SIZE := <boot partition size>
```

After flashing U-Boot don't forget to update environment and write new partition table:

```
=> env default -f -a
=> setenv partitions $partitions_android
=> env save
=> gpt write mmc 1 $partitions_android
```

References

8.1.3 Android Bootloader Control Block (BCB)

The purpose behind this file is to:

- give an overview of BCB w/o duplicating public documentation
- describe the main BCB use-cases which concern U-Boot
- reflect current support status in U-Boot
- mention any relevant U-Boot build-time tunables
- precisely exemplify one or more use-cases

Additions and fixes are welcome!

Overview

Bootloader Control Block (BCB) is a well established term/acronym in the Android namespace which refers to a location in a dedicated raw (i.e. FS-unaware) flash (e.g. eMMC) partition, usually called `misc`, which is used as media for exchanging messages between Android userspace (particularly recovery[1]) and an Android-capable bootloader.

On higher level, BCB provides a way to implement a subset of Android Bootloader Requirements[2], amongst which are:

[1] https://android.googlesource.com/platform/bootable/recovery
[2] https://source.android.com/devices/bootloader

- Android-specific bootloader flow[3]
- Get the “reboot reason” (and act accordingly)[4]
- Get/pass a list of commands from/to recovery[1]
- TODO

‘bcb’. Shell command overview

The bcb command provides a CLI to facilitate the development of the requirements enumerated above. Below is the command’s help message:

```
=> bcb
bcb - Load/set/clear/test/dump/store Android BCB fields

Usage:
bcb load  <dev> <part>       - load  BCB from mmc <dev>:<part>
bcb set   <field> <val>      - set   BCB <field> to <val>
bcb clear [<field>]          - clear BCB <field> or all fields
bcb test  <field> <op> <val> - test  BCB <field> against <val>
bcb dump  <field>            - dump  BCB <field>
bcb store                    - store BCB back to mmc

Legend:
<dev>   - MMC device index containing the BCB partition
<part>  - MMC partition index or name containing the BCB
<field> - one of {command,status,recovery,stage,reserved}
<op>    - the binary operator used in 'bcb test':
          '=' returns true if <val> matches the string stored in <field>
          '~' returns true if <val> matches a subset of <field>'s string
<val>   - string/text provided as input to bcb {set,test}
          NOTE: any ':' character in <val> will be replaced by line feed
          during 'bcb set' and used as separator by upper layers
```

‘bcb’. Example of getting reboot reason

```
if bcb load 1 misc; then
    # valid BCB found
    if bcb test command = bootonce-bootloader; then
        bcb clear command; bcb store;
        # do the equivalent of AOSP ${fastbootcmd}
        # i.e. call fastboot
    else if bcb test command = boot-recovery; then
        bcb clear command; bcb store;
        # do the equivalent of AOSP ${recoverycmd}
        # i.e. do anything required for booting into recovery
    else
        # boot Android OS normally
    fi
else
    # corrupted/non-existent BCB
    # report error or boot non-Android OS (platform-specific)
fi
```

[3] https://patchwork.ozlabs.org/patch/746835/ (“[U-Boot,5/6] Initial support for the Android Bootloader flow”)
[4] https://source.android.com/devices/bootloader/boot-reason

Enable on your board

The following Kconfig options must be enabled:

```
CONFIG_PARTITIONS=y
CONFIG_MMC=y
CONFIG_BCB=y
```

8.1.4 Android Boot Image

Overview

Android Boot Image is used to boot Android OS. It usually contains kernel image (like `zImage` file) and ramdisk. Sometimes it can contain additional binaries. This image is built as a part of AOSP (called `boot.img`), and being flashed into `boot` partition on eMMC. Bootloader then reads that image from `boot` partition to RAM and boots the kernel from it. Kernel than starts `init` process from the ramdisk. It should be mentioned that recovery image (`recovery.img`) also has Android Boot Image format.

Android Boot Image format is described at[1]. At the moment it can have one of next image headers:

- v0: it's called *legacy* boot image header; used in devices launched before Android 9; contains kernel image, ramdisk and second stage bootloader (usually unused)
- v1: used in devices launched with Android 9; adds `recovery_dtbo` field, which should be used for non-A/B devices in `recovery.img` (see[2] for details)
- v2: used in devices launched with Android 10; adds `dtb` field, which references payload containing DTB blobs (either concatenated one after the other, or in Android DTBO image format)

v2, v1 and v0 formats are backward compatible.

Android Boot Image format is represented by `struct andr_img_hdr` in U-Boot, and can be seen in `include/android_image.h`. U-Boot supports booting Android Boot Image and also has associated command

Booting

U-Boot is able to boot the Android OS from Android Boot Image using `bootm` command. In order to use Android Boot Image format support, next option should be enabled:

```
CONFIG_ANDROID_BOOT_IMAGE=y
```

Then one can use next `bootm` command call to run Android:

```
=> bootm $loadaddr $loadaddr $fdtaddr
```

where `$loadaddr` - address in RAM where boot image was loaded; `$fdtaddr` - address in RAM where DTB blob was loaded.

And parameters are, correspondingly:

1. Where kernel image is located in RAM
2. Where ramdisk is located in RAM (can be `"-"` if not applicable)
3. Where DTB blob is located in RAM

`bootm` command will figure out that image located in `$loadaddr` has Android Boot Image format, will parse that and boot the kernel from it, providing DTB blob to kernel (from 3rd parameter), passing info about ramdisk to kernel via DTB.

[1] https://source.android.com/devices/bootloader/boot-image-header
[2] https://source.android.com/devices/bootloader/recovery-image

DTB and DTBO blobs

`bootm` command can't just use DTB blob from Android Boot Image (`dtb` field), because:

- there is no DTB area in Android Boot Image before v2
- there may be several DTB blobs in DTB area (e.g. for different SoCs)
- some DTBO blobs may have to be merged in DTB blobs before booting (e.g. for different boards)

So user has to prepare DTB blob manually and provide it in a 3rd parameter of `bootm` command. Next commands can be used to do so:

1. `abootimg`: manipulates Anroid Boot Image, allows one to extract meta-information and payloads from it
2. `adtimg`: manipulates Android DTB/DTBO image[3], allows one to extract DTB/DTBO blobs from it

In order to use those, please enable next config options:

```
CONFIG_CMD_ABOOTIMG=y
CONFIG_CMD_ADTIMG=y
```

For example, let's assume we have next Android partitions on eMMC:

- `boot`: contains Android Boot Image v2 (including DTB blobs)
- `dtbo`: contains DTBO blobs

Then next command sequence can be used to boot Android:

```
=> mmc dev 1

   # Read boot image to RAM (into $loadaddr)
=> part start mmc 1 boot boot_start
=> part size mmc 1 boot boot_size
=> mmc read $loadaddr $boot_start $boot_size

   # Read DTBO image to RAM (into $dtboaddr)
=> part start mmc 1 dtbo dtbo_start
=> part size mmc 1 dtbo dtbo_size
=> mmc read $dtboaddr $dtbo_start $dtbo_size

   # Copy required DTB blob (into $fdtaddr)
=> abootimg get dtb --index=0 dtb0_start dtb0_size
=> cp.b $dtb0_start $fdtaddr $dtb0_size

   # Merge required DTBO blobs into DTB blob
=> fdt addr $fdtaddr 0x100000
=> adtimg addr $dtboaddr
=> adtimg get dt --index=0 $dtbo0_addr
=> fdt apply $dtbo0_addr

   # Boot Android
=> bootm $loadaddr $loadaddr $fdtaddr
```

This sequence should be used for Android 10 boot. Of course, the whole Android boot procedure includes much more actions, like:

- obtaining reboot reason from BCB (see[4])
- implementing recovery boot

[3] https://source.android.com/devices/architecture/dto/partitions
[4] *Android Bootloader Control Block (BCB)*

- implementing fastboot boot
- implementing A/B slotting (see[5])
- implementing AVB2.0 (see[6])

But Android Boot Image booting is the most crucial part in Android boot scheme.

All Android bootloader requirements documentation is available at[7]. Some overview on the whole Android 10 boot process can be found at[8].

C API for working with Android Boot Image format

int **android_image_get_kernel**(const struct andr_img_hdr * *hdr*, int *verify*, ulong * *os_data*, ulong * *os_len*)
processes kernel part of Android boot images

Parameters

const struct andr_img_hdr * hdr Pointer to image header, which is at the start of the image.

int verify Checksum verification flag. Currently unimplemented.

ulong * os_data Pointer to a ulong variable, will hold os data start address.

ulong * os_len Pointer to a ulong variable, will hold os data length.

Description

This function returns the os image's start address and length. Also, it appends the kernel command line to the bootargs env variable.

Return

Zero, os start address and length on success, otherwise on failure.

bool **android_image_get_dtbo**(ulong *hdr_addr*, ulong * *addr*, u32 * *size*)
Get address and size of recovery DTBO image.

Parameters

ulong hdr_addr Boot image header address

ulong * addr If not NULL, will contain address of recovery DTBO image

u32 * size If not NULL, will contain size of recovery DTBO image

Description

Get the address and size of DTBO image in "Recovery DTBO" area of Android Boot Image in RAM. The format of this image is Android DTBO (see corresponding "DTB/DTBO Partitions" AOSP documentation for details). Once the address is obtained from this function, one can use 'adtimg' U-Boot command or android_dt_*() functions to extract desired DTBO blob.

This DTBO (included in boot image) is only needed for non-A/B devices, and it only can be found in recovery image. On A/B devices we can always rely on "dtbo" partition. See "Including DTBO in Recovery for Non-A/B Devices" in AOSP documentation for details.

Return

true on success or false on error.

bool **android_image_get_dtb_img_addr**(ulong *hdr_addr*, ulong * *addr*)
Get the address of DTB area in boot image.

Parameters

[5] *Android A/B updates*
[6] *Android Verified Boot 2.0*
[7] https://source.android.com/devices/bootloader
[8] https://connect.linaro.org/resources/san19/san19-217/

ulong hdr_addr Boot image header address

ulong * addr Will contain the address of DTB area in boot image

Return

true on success or false on fail.

bool **android_image_get_dtb_by_index**(ulong *hdr_addr*, u32 *index*, ulong * *addr*, u32 * *size*)
Get address and size of blob in DTB area.

Parameters

ulong hdr_addr Boot image header address

u32 index Index of desired DTB in DTB area (starting from 0)

ulong * addr If not NULL, will contain address to specified DTB

u32 * size If not NULL, will contain size of specified DTB

Description

Get the address and size of DTB blob by its index in DTB area of Android Boot Image in RAM.

Return

true on success or false on error.

void **android_print_contents**(const struct andr_img_hdr * *hdr*)
prints out the contents of the Android format image

Parameters

const struct andr_img_hdr * hdr pointer to the Android format image header

Description

android_print_contents() formats a multi line Android image contents description. The routine prints out Android image properties

Return

no returned results

bool **android_image_print_dtb_info**(const struct fdt_header * *fdt*, u32 *index*)
Print info for one DTB blob in DTB area.

Parameters

const struct fdt_header * fdt DTB header

u32 index Number of DTB blob in DTB area.

Return

true on success or false on error.

bool **android_image_print_dtb_contents**(ulong *hdr_addr*)
Print info for DTB blobs in DTB area.

Parameters

ulong hdr_addr Boot image header address

Description

DTB payload in Android Boot Image v2+ can be in one of following formats:

1. Concatenated DTB blobs
2. Android DTBO format (see CONFIG_CMD_ADTIMG for details)

This function does next:

1. Prints out the format used in DTB area

2. Iterates over all DTB blobs in DTB area and prints out the info for each blob.

Return

true on success or false on error.

References

8.1.5 FastBoot Version 0.4

The fastboot protocol is a mechanism for communicating with bootloaders over USB. It is designed to be very straightforward to implement, to allow it to be used across a wide range of devices and from hosts running Linux, Windows, or OSX.

Basic Requirements

- Two bulk endpoints (in, out) are required
- Max packet size must be 64 bytes for full-speed and 512 bytes for high-speed USB
- The protocol is entirely host-driven and synchronous (unlike the multi-channel, bi-directional, asynchronous ADB protocol)

Transport and Framing

1. Host sends a command, which is an ascii string in a single packet no greater than 64 bytes.
2. Client response with a single packet no greater than 64 bytes. The first four bytes of the response are “OKAY”, “FAIL”, “DATA”, or “INFO”. Additional bytes may contain an (ascii) informative message.
   a. INFO -> the remaining 60 bytes are an informative message (providing progress or diagnostic messages). They should be displayed and then step #2 repeats
   b. FAIL -> the requested command failed. The remaining 60 bytes of the response (if present) provide a textual failure message to present to the user. Stop.
   c. OKAY -> the requested command completed successfully. Go to #5
   d. DATA -> the requested command is ready for the data phase. A DATA response packet will be 12 bytes long, in the form of DATA00000000 where the 8 digit hexidecimal number represents the total data size to transfer.
3. Data phase. Depending on the command, the host or client will send the indicated amount of data. Short packets are always acceptable and zero-length packets are ignored. This phase continues until the client has sent or received the number of bytes indicated in the “DATA” response above.
4. Client responds with a single packet no greater than 64 bytes. The first four bytes of the response are “OKAY”, “FAIL”, or “INFO”. Similar to #2:
   a. INFO -> display the remaining 60 bytes and return to #4
   b. FAIL -> display the remaining 60 bytes (if present) as a failure reason and consider the command failed. Stop.
   c. OKAY -> success. Go to #5
5. Success. Stop.

Example Session

```
Host:    "getvar:version"        request version variable

Client:  "OKAY0.4"               return version "0.4"

Host:    "getvar:nonexistant"    request some undefined variable

Client:  "OKAY"                  return value ""

Host:    "download:00001234"     request to send 0x1234 bytes of data

Client:  "DATA00001234"          ready to accept data

Host:    < 0x1234 bytes >        send data

Client:  "OKAY"                  success

Host:    "flash:bootloader"      request to flash the data to the bootloader

Client:  "INFOerasing flash"     indicate status / progress
         "INFOwriting flash"
         "OKAY"                  indicate success

Host:    "powerdown"             send a command

Client:  "FAILunknown command"   indicate failure
```

Command Reference

- Command parameters are indicated by printf-style escape sequences.
- Commands are ascii strings and sent without the quotes (which are for illustration only here) and without a trailing 0 byte.
- Commands that begin with a lowercase letter are reserved for this specification. OEM-specific commands should not begin with a lowercase letter, to prevent incompatibilities with future specs.

```
"getvar:%s"            Read a config/version variable from the bootloader.
                       The variable contents will be returned after the
                       OKAY response.

"download:%08x"        Write data to memory which will be later used
                       by "boot", "ramdisk", "flash", etc.  The client
                       will reply with "DATA%08x" if it has enough
                       space in RAM or "FAIL" if not.  The size of
                       the download is remembered.

 "verify:%08x"         Send a digital signature to verify the downloaded
                       data.  Required if the bootloader is "secure"
                       otherwise "flash" and "boot" will be ignored.

 "flash:%s"            Write the previously downloaded image to the
                       named partition (if possible).

 "erase:%s"            Erase the indicated partition (clear to 0xFFs)

 "boot"                The previously downloaded data is a boot.img
                       and should be booted according to the normal
```

(continues on next page)

(continued from previous page)

```
                      procedure for a boot.img

"continue"            Continue booting as normal (if possible)

"reboot"              Reboot the device.

"reboot-bootloader"   Reboot back into the bootloader.
                      Useful for upgrade processes that require upgrading
                      the bootloader and then upgrading other partitions
                      using the new bootloader.

"powerdown"           Power off the device.
```

Client Variables

The `getvar:%s` command is used to read client variables which represent various information about the device and the software on it.

The various currently defined names are:

```
version              Version of FastBoot protocol supported.
                     It should be "0.3" for this document.

version-bootloader   Version string for the Bootloader.

version-baseband     Version string of the Baseband Software

product              Name of the product

serialno             Product serial number

secure               If the value is "yes", this is a secure
                     bootloader requiring a signature before
                     it will install or boot images.
```

Names starting with a lowercase character are reserved by this specification. OEM-specific names should not start with lowercase characters.

8.1.6 Android Fastboot

Overview

The protocol that is used over USB and UDP is described in[1].

The current implementation supports the following standard commands:

- `boot`
- `continue`
- `download`
- `erase` (if enabled)
- `flash` (if enabled)
- `getvar`

[1] *FastBoot Version 0.4*

- `reboot`
- `reboot-bootloader`
- `set_active` (only a stub implementation which always succeeds)

The following OEM commands are supported (if enabled):

- `oem format` - this executes `gpt write mmc %x $partitions`

Support for both eMMC and NAND devices is included.

Client installation

The counterpart to this is the fastboot client which can be found in Android's `platform/system/core` repository in the fastboot folder. It runs on Windows, Linux and OSX. The fastboot client is part of the Android SDK Platform-Tools and can be downloaded from[2].

Board specific

USB configuration

The fastboot gadget relies on the USB download gadget, so the following options must be configured:

```
CONFIG_USB_GADGET_DOWNLOAD
CONFIG_USB_GADGET_VENDOR_NUM
CONFIG_USB_GADGET_PRODUCT_NUM
CONFIG_USB_GADGET_MANUFACTURER
```

NOTE: The `CONFIG_USB_GADGET_VENDOR_NUM` must be one of the numbers supported by the fastboot client. The list of vendor IDs supported can be found in the fastboot client source code.

General configuration

The fastboot protocol requires a large memory buffer for downloads. This buffer should be as large as possible for a platform. The location of the buffer and size are set with `CONFIG_FASTBOOT_BUF_ADDR` and `CONFIG_FASTBOOT_BUF_SIZE`. These may be overridden on the fastboot command line using `-l` and `-s`.

Fastboot environment variables

Partition aliases

Fastboot partition aliases can also be defined for devices where GPT limitations prevent user-friendly partition names such as `boot`, `system` and `cache`. Or, where the actual partition name doesn't match a standard partition name used commonly with fastboot.

The current implementation checks aliases when accessing partitions by name (flash_write and erase functions). To define a partition alias add an environment variable similar to:

```
fastboot_partition_alias_<alias partition name>=<actual partition name>
```

for example:

```
fastboot_partition_alias_boot=LNX
```

[2] https://developer.android.com/studio/releases/platform-tools

Raw partition descriptors

In cases where no partition table is present, a raw partition descriptor can be defined, specifying the offset, size, and optionally the MMC hardware partition number for a given partition name.

This is useful when using fastboot to flash files (e.g. SPL or U-Boot) to a specific offset in the eMMC boot partition, without having to update the entire boot partition.

To define a raw partition descriptor, add an environment variable similar to:

```
fastboot_raw_partition_<raw partition name>=<offset> <size> [mmcpart <num>]
```

for example:

```
fastboot_raw_partition_boot=0x100 0x1f00 mmcpart 1
```

Variable overrides

Variables retrived through `getvar` can be overridden by defining environment variables of the form `fastboot.<variable>`. These are looked up first so can be used to override values which would otherwise be returned. Using this mechanism you can also return types for NAND filesystems, as the fully parameterised variable is looked up, e.g.:

```
fastboot.partition-type:boot=jffs2
```

Boot command

When executing the fastboot `boot` command, if `fastboot_bootcmd` is set then that will be executed in place of `bootm <CONFIG_FASTBOOT_BUF_ADDR>`.

Partition Names

The Fastboot implementation in U-Boot allows to write images into disk partitions. Target partitions are referred on the host computer by their names.

For GPT/EFI the respective partition name is used.

For MBR the partitions are referred by generic names according to the following schema:

```
<device type><device index letter><partition index>
```

Example: `hda3`, `sdb1`, `usbda1`.

The device type is as follows:

- IDE, ATAPI and SATA disks: `hd`
- SCSI disks: `sd`
- USB media: `usbd`
- MMC and SD cards: `mmcsd`
- Disk on chip: `docd`
- other: `xx`

The device index starts from `a` and refers to the interface (e.g. USB controller, SD/MMC controller) or disk index. The partition index starts from `1` and describes the partition number on the particular device.

Writing Partition Table

Fastboot also allows to write the partition table to the media. This can be done by writing the respective partition table image to a special target "gpt" or "mbr". These names can be customized by defining the following configuration options:

```
CONFIG_FASTBOOT_GPT_NAME
CONFIG_FASTBOOT_MBR_NAME
```

In Action

Enter into fastboot by executing the fastboot command in U-Boot for either USB:

```
=> fastboot usb 0
```

or UDP:

```
=> fastboot udp
link up on port 0, speed 100, full duplex
Using ethernet@4a100000 device
Listening for fastboot command on 192.168.0.102
```

On the client side you can fetch the bootloader version for instance:

```
$ fastboot getvar version-bootloader
version-bootloader: U-Boot 2019.07-rc4-00240-g00c9f2a2ec
Finished. Total time: 0.005s
```

or initiate a reboot:

```
$ fastboot reboot
```

and once the client comes back, the board should reset.

You can also specify a kernel image to boot. You have to either specify the an image in Android format *or* pass a binary kernel and let the fastboot client wrap the Android suite around it. On OMAP for instance you take zImage kernel and pass it to the fastboot client:

```
$ fastboot -b 0x80000000 -c "console=ttyO2 earlyprintk root=/dev/ram0 mem=128M" boot␣
↪zImage
creating boot image...
creating boot image - 1847296 bytes
downloading 'boot.img'...
OKAY [  2.766s]
booting...
OKAY [ -0.000s]
finished. total time: 2.766s
```

and on the U-Boot side you should see:

```
Starting download of 1847296 bytes
........................................................
downloading of 1847296 bytes finished
Booting kernel..
## Booting Android Image at 0x81000000 ...
Kernel load addr 0x80008000 size 1801 KiB
Kernel command line: console=ttyO2 earlyprintk root=/dev/ram0 mem=128M
   Loading Kernel Image ... OK
```

(continues on next page)

(continued from previous page)

```
OK

Starting kernel ...
```

References

CHAPTER

NINE

COMMAND LINE

9.1 PStore command

9.1.1 Design

Linux PStore and Ramoops modules (Linux config options PSTORE and PSTORE_RAM) allow to use memory to pass data from the dying breath of a crashing kernel to its successor. This command allows to read those records from U-Boot command line.

Ramoops is an oops/panic logger that writes its logs to RAM before the system crashes. It works by logging oopses and panics in a circular buffer. Ramoops needs a system with persistent RAM so that the content of that area can survive after a restart.

Ramoops uses a predefined memory area to store the dump.

Ramoops parameters can be passed as kernel parameters or through Device Tree, i.e.:

```
ramoops.mem_address=0x30000000 ramoops.mem_size=0x100000 ramoops.record_size=0x2000␣
↪ramoops.console_size=0x2000 memmap=0x100000$0x30000000
```

The same values should be set in U-Boot to be able to retrieve the records. This values can be set at build time in U-Boot configuration file, or at runtime. U-Boot automatically patches the Device Tree to pass the Ramoops parameters to the kernel.

The PStore configuration parameters are:

Name	Default
CMD_PSTORE_MEM_ADDR	
CMD_PSTORE_MEM_SIZE	0x10000
CMD_PSTORE_RECORD_SIZE	0x1000
CMD_PSTORE_CONSOLE_SIZE	0x1000
CMD_PSTORE_FTRACE_SIZE	0x1000
CMD_PSTORE_PMSG_SIZE	0x1000
CMD_PSTORE_ECC_SIZE	0

Records sizes should be a power of 2. The memory size and the record/console size must be non-zero.

Multiple 'dump' records can be stored in the memory reserved for PStore. The memory size has to be larger than the sum of the record sizes, i.e.:

```
MEM_SIZE >= RECORD_SIZE * n + CONSOLE_SIZE + FTRACE_SIZE + PMSG_SIZE
```

9.1.2 Usage

Generate kernel crash

For test purpose, you can generate a kernel crash by setting reboot timeout to 10 seconds and trigger a panic:

```
$ sudo sh -c "echo 1 > /proc/sys/kernel/sysrq"
$ sudo sh -c "echo 10 > /proc/sys/kernel/panic"
$ sudo sh -c "echo c > /proc/sysrq-trigger"
```

Retrieve logs in U-Boot

First of all, unless PStore parameters as been set during U-Boot configuration and match kernel ramoops parameters, it needs to be set using 'pstore set', e.g.:

```
=> pstore set 0x30000000 0x100000 0x2000 0x2000
```

Then all available dumps can be displayed using:

```
=> pstore display
```

Or saved to an existing directory in an Ext2 or Ext4 partition, e.g. on root directory of 1st partition of the 2nd MMC:

```
=> pstore save mmc 1:1 /
```

CHAPTER

TEN

INDICES AND TABLES

- genindex