```
/*
 * Copyright (c) 2014 The Chromium OS Authors.
 *
 * SPDX-License-Identifier:     GPL-2.0+
 */
```

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 chosen vendor and board names are also 'sandbox', so there is a single
board in board/sandbox/sandbox.

CONFIG_SANDBOX_BIG_ENDIAN should be defined when running on big-endian
machines.

Note that standalone/API support is not available at present.

Basic Operation

To run sandbox U-Boot use something like:

```
   make sandbox_config all
   ./u-boot
```

Note:
   If you get errors about 'sdl-config: Command not found' you may need to
   install libsdl1.2-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_config 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.

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).

As mentioned above, -l causes the LCD emulation window to be shown.

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.

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 shall will normally process and
swallow quotes. When -c is used, U-Boot exists after the command is complete,
but you can force it to go to interactive mode instead with -i.


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)
- Keyboard (Chrome OS)
- LCD
- Serial (for console only)
- Sound (incomplete - see sandbox_sdl_sound_init() for details)
- SPI
- SPI flash
- TPM (Trusted Platform Module)

Notable omissions are networking and I2C.

A wide range of commands is implemented. Filesystems which use a block
device are supported.

Also sandbox uses generic board (CONFIG_SYS_GENERIC_BOARD) and supports
driver model (CONFIG_DM) and associated commands.


SPI Emulation
-------------

Sandbox supports SPI and SPI flash emulation.

This is controlled by the spi_sf argument, the format of which is:

   bus:cs:device:file
```

```
   bus    - SPI bus number
   cs     - SPI chip select number
   device - SPI device emulation name
   file   - File on disk containing the data

For example:

 dd if=/dev/zero of=spi.bin bs=1M count=4
 ./u-boot --spi_sf 0:0:M25P16:spi.bin

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).

Drivers are connected to a particular bus/cs using sandbox's state
structure (see the 'spi' member). A set of operations must be provided
for each driver.


Configuration settings for the curious are:

CONFIG_SANDBOX_SPI_MAX_BUS
        The maximum number of SPI buses supported by the driver (default 1).

CONFIG_SANDBOX_SPI_MAX_CS
        The maximum number of chip selects supported by the driver
        (default 10).

CONFIG_SPI_IDLE_VAL
        The idle value on the SPI bus


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.


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
     - test/dm/test-dm.sh to run these.
  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).

It would be useful to have a central script to run all of these.

--
Simon Glass <sjg@chromium.org>
Updated 22-Mar-14
```