ConnectCore 6

U-Boot Customizations

Reference Manual

Revision history—90001422

Revision	Date	Description
A	August, 2014	Initial release
B	October, 2014	Added SBCv2 support; added carrier board version support; disable autoscript by default; console is UART4 on SBC; added known issues.
C	June, 2015	Update from RAM support; eMMC-less variants support; corrected link on page 5.
D	September, 2017	Updated branding and made editorial enhancements.

Trademarks and copyright

Disclaimers

Information in this document is subject to change without notice and does not represent a commitment on the part of Digi International. Digi provides this document “as is,” without warranty of any kind, expressed or implied, including, but not limited to, the implied warranties of fitness or merchantability for a particular purpose. Digi may make improvements and/or changes in this manual or in the product(s) and/or the program(s) described in this manual at any time.

Warranty

To view product warranty information, go to the following website:

www.digi.com/howtobuy/terms

Send comments

Documentation feedback: To provide feedback on this document, send your comments to techcomm@digi.com.

Customer support

Digi Technical Support: Digi offers multiple technical support plans and service packages to help our customers get the most out of their Digi product. For information on Technical Support plans and pricing, contact us at +1 952.912.3444 or visit us at www.digi.com/support.

Contents

Updating the boot loader

Recovering a bricked device

Known issues

About the ConnectCore 6 U-Boot

The ConnectCore 6 platform uses U-boot (Universal Bootloader) as its primary boot loader. U- Boot is an open-source project. Standard documentation about commands, environment variables, Flattened Device Tree support and more can be found at www.denx.de/wiki/U-Boot/Documentation.

This guide only describes specific changes done by Digi to the standard U-Boot.

Version

U-Boot for ConnectCore 6 is based on standard U-Boot v2013.04 plus Freescale patches at git.freescale.com/git/cgit.cgi/imx/uboot-imx.git.

Supported interfaces

The ConnectCore 6 module supports the following interfaces:

- UART4 for console (115200/8/N/1)
- 1GB DDR3
- 4GB eMMC for storage (on USDHC4)
- uSD card (on USDHC2)
- Gigabit Ethernet on Micrel PHY KSZ9031 (RGMII)
- I2C multi-port
- OTP bits (e-fuses)

Digi custom commands

In addition to the standard U-Boot commands, Digi provides custom U-Boot commands to perform common embedded platforms operations more efficiently.

Direct boot: 'dboot'

The **dboot** command simplifies operating system bootup by performing the following operations:

- Downloads the required files (kernel, flattened device tree, init ramdisk) from the specified media to RAM.
- Sets the **bootargs** variable with the boot command line required for the operating system.
- Boots the system.

Syntax

```
=> help dboot
dboot - Digi modules boot command
Usage:
dboot <os> [source] [extra-args...]
 Description: Boots <os> via <source>
 Arguments:
   - os:           a partition name or one of the reserved names:
                   linux|android
   - [source]:     tftp|nfs|mmc
   - [extra-args]: extra arguments depending on 'source'
      source=tftp|nfs -> [filename]
       - filename: file to transfer (if not provided, filename will
                   will be taken from variable '<partition>_file')
      source=mmc -> [device:part] [filesystem] [filename]
       - device:part: number of device and partition
       - filesystem: fat (default)|ext4
       - filename: file to transfer (if not provided, filename will
will be taken from variable '<partition>_file')
```

Examples

- Boot Linux from TFTP using kernel image **uImage-test**:

```
=>  dboot linux tftp uImage-test
```

- Boot Android from FAT partition 1 on eMMC (kernel image name taken by default from variable **$uimage**):

```
=>  dboot android mmc
```

The behavior of the dboot command is highly customizable through its parameters and also using the following environment variables:

Addresses

- **$loadaddr**: this is the RAM address where the kernel image is downloaded to.
- **$fdt_addr**: this is the RAM address where the flattened device tree image is downloaded to.
- **$initrd_addr**: this is the RAM address where the init ram disk image is downloaded to.

Image file names

- **$uimage**: this is the kernel image filename
- **$fdt_file**: this is the flattened device tree image filename
- **$initrd_file**: this is the init ramdisk image filename

Modifiers

- **$boot_fdt**: can be set to:
  - **yes**: flattened device tree is required. If it cannot be downloaded **dboot** command will fail.
  - **try**: **dboot** will try to load a flattened device tree file but will continue to boot the kernel without it if not available.
  - **no**: flattened device tree not required. **dboot** will not try to download it.
- **$boot_initrd**: can be set to:
  - **yes**: init ramdisk is required. If it cannot be downloaded **dboot** command will fail.
  - **try**: **dboot** will try to load an init ramdisk file but will continue to boot the kernel without it if not available.
  - **no**: init ramdisk not required. **dboot** will not try to download it.
- **$ip_dyn**: can be set to
  - **yes**: will append 'ip=dhcp' to the kernel $booargs when booting for fetching dynamic address when booting from TFTP/NFS.
  - **no**: will append 'ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}' to the kernel $booargs for using the static IP address configured in U-Boot, when booting from TFTP/NFS.

Source media

- **$mmcdev**: this is the default device index when reading files from the MMC (when no device index is passed as parameter).
- **$mmcpart**: this is the default partition index when reading files from the MMC (when no partition index is passed as parameter).

Boot arguments

- **$bootargs_android**: contains common Android boot arguments (independent of the media where Android is booting from).
- **$bootargs_mmc_android**: script that sets the **bootargs** variable with Android boot arguments when booting from the MMC.
- **$bootargs_tftp_android/$bootargs_nfs_android**: script that sets the **bootargs** variable with Android boot arguments when booting from TFTP and NFS.
- **$bootargs_mmc_linux**: script that sets the **bootargs** variable with Linux boot arguments when booting from the MMC.

- **$bootargs_tftp_linux/$bootargs_nfs_linux**: script that sets the **bootargs** variable with Linux boot arguments when booting from TFTP and NFS.
- **$rootpath**: this is the NFS root path to use in the **bootargs** when booting from TFTP/NFS.
- **$mmcroot**: this is the root path to use in the **bootargs** when booting from MMC.
- **$video_args:** the contents of this variable are appended to the **bootargs** variable when booting Android OS. The variable is a placeholder for the video arguments to pass to Android OS (Linux OS defines the video via the Device Tree).
- **$bootargs_once**: this is a special variable that is appended to the **bootargs**. It can only be set locally with the equals sign not with **setenv** command:

```
=>  bootargs_once="<arguments>"
```

This means the variable is not saved across reboots and will pass boot arguments only once.

- **$extra_bootargs**: this variable is a placeholder for appending any customized boot arguments at the end of the cmdline.

Firmware update: 'update'

The **update** command simplifies the task of updating a partition of the storage media by performing the following operations:

- Download the required firmware image files from the specified media to RAM.
- Retrieve and check information (offset, size) of the partition to update.
- Write the firmware image to the partition.
- Read back and verify the written firmware image.
- The syntax of the **update** command is:

```
=> help update
update - Digi modules update command

Usage:
update <partition>  [source] [extra-args...]
 Description: updates (raw writes) <partition> in $mmcdev via <source>
 Arguments:
   - partition:    a partition index, a GUID partition name, or one
                   of the reserved names: uboot
   - [source]:     tftp|nfs|mmc|ram
   - [extra-args]: extra arguments depending on 'source'

      source=tftp|nfs -> [filename]
       - filename: file to transfer (if not provided, filename will
                   will be taken from variable '<partition>_file')

      source=mmc -> [device:part] [filesystem] [filename]
       - device:part: number of device and partition
       - filesystem: fat (default)|ext4
       - filename: file to transfer (if not provided, filename will
                   will be taken from variable '<partition>_file')
```

```
source=ram -> [image_address] [image_size]
 - image_address: address of image in RAM
                  ($loadaddr if not provided)
 - image_size: size of image in RAM
               ($filesize if not provided)
```

Examples

- Update partition named *system* with file *test.img* downloaded from TFTP:

```
=>  update system tftp test.img
```

- Update the boot loader on the eMMC of the module using default filename (in variable *uboot_file*) that is stored on a FAT partition on the uSD card:

```
=>  update uboot mmc 1:1 fat
```

The behavior of **update** command is customizable through its parameters and also using the following environment variables:

Addresses

- **$loadaddr**: this is the RAM address where the firmware image file to use for the update is downloaded to.
- **$verifyaddr**: this is the RAM address where the firmware image is read back for verification. Normally, you should not set this variable, and U-Boot automatically sets it to a RAM address halfway through the available RAM, starting at $loadaddr, to maximize the size of firmware that can be transferred to RAM and verified during the update process.

Image file names

- **$uboot_file**: this is the default image filename to use for updating the boot loader (if no filename is passed as parameter).
- • **$<partition-name>_file**: a variable of this form will contain the default filename to use for updating the partition of that specific name. E.g. for a partition named *system* the variable containing the default filename should be called **system_file**.

Source media

- **$mmcdev**: this is the default device index when reading files from the MMC (when no device index is passed as parameter). This variable is also used as target MMC device index to write the firmware to.
- **$mmcpart**: this is the default partition index when reading files from the MMC (when no partition index is passed as parameter).

On-the-fly (OTF) update mechanism

The standard **update** command first transfers the file completely from a media into the RAM memory. Then it writes the file from RAM into the MMC partition. Occasionally, the firmware files used to update a certain partition are very large, maybe larger than the available RAM of the platform. In that scenario, we must use a mechanism that transfers the file from the media to RAM in chunks. After transferring one chunk, the chunk is written to the storage media verified, and the RAM memory is reused to transfer the next chunk. We call this mechanism **on-the-fly update**.

The on-the-fly update mechanism can be activated by defining the variable otf-update to yes, and then using the update command normally. For example:

```
=>  setenv otf-update yes
```

The on-the-fly update mechanism can only work when updating from TFTP source and is only recommended to update very large images. Notice that if there is any problem during the transmission of the file, the partition will be left partially written (as opposed to the standard update where if the transmission fails, the partition remains untouched).

For security reasons, the on-the-fly update mechanism is automatically disabled when updating the boot loader.

If the variable **otf-update** is undefined, U-Boot may automatically activate the on-the-fly update mechanism if the size of the partition to be updated is larger than the available RAM to hold the firmware.

WARNING! The on-the-fly update mechanism will pause the TFTP transfer while it writes each chunk to the storage media, and will resume it after reading back and verifying the written data, for this reason it may not work with all TFTP servers.

Check that your TFTP server has a large enough timeout so that it does not cancel the transfer due to not receiving the client's ACKs in time while the device is writing. Also, disable any mechanism that sends packets without waiting for the client's ACKs (such as anticipation window) that might not work with on-the-fly update mechanism.

Update files within partition

The **updatefile** command simplifies the update of files in a partition filesystem. While the standard **update** command writes raw data to the storage media, the **updatefile** command uses U-Boot file system support to write files to a formatted partition. Specifically it does the following:

- Download the required files from the specified media to RAM.
- Write the file to the storage media partition filesystem.

The syntax of the **updatefile** command is:

```
=> help updatefile
updatefile - Digi modules updatefile command

Usage:
updatefile <partition>  [source] [extra-args...]
 Description: updates/writes a file in <partition> in $mmcdev via
              <source>
 Arguments:
   - partition:    a partition index or a GUID partition name where
                   to upload the file
   - [source]:     tftp|nfs|mmc|ram
```

```
   - [extra-args]: extra arguments depending on 'source'

      source=tftp|nfs -> [source_file] [targetfile] [target_fs]
       - source_file: file to transfer
       - target_file: target filename
       - target_fs: fat (default)

      source=mmc -> [device:part] [filesystem] [source_file] [target_file]
[target_fs]
       - device:part: number of device and partition
       - filesystem: fat (default)|ext4
       - source_file: file to transfer
       - target_file: target filename
       - target_fs: fat (default)

      source=ram -> [image_address] [image_size] [targetfile] [target_fs]
       - image_address: address of image in RAM
                        ($loadaddr if not provided)
       - image_size: size of image in RAM
                     ($filesize if not provided)
       - target_file: target filename
       - target_fs: fat (default)
```

Examples:

Update file **newkernel.bin** from TFTP to linux FAT partition:

```
=> updatefile linux tftp newkernel.bin
```

Update file **newkernel.bin** from uSD card FAT partition 3 to linux FAT partition and save it with filename **my-uImage**:

```
=> updatefile linux mmc 1:3 fat newkernel.bin my-uImage fat
```

Limitations:

- Only FAT filesystem is supported
- Files can be overwritten but cannot be deleted
- Filesystem support does not check for available space
- Writing a file on a filesystem that does not have space for it may corrupt the entire filesystem.

MMC extended CSD register: mmc ecsd

eMMC v4.4 specification defines a 512-byte Extended CSD register that contains parameters of the eMMC. The mmc command has been extended with subcommand ecsd to access the fields of this register. Available operations are:

```
=>  help mmc
mmc - MMC sub system

Usage:
mmc read addr blk# cnt mmc write addr blk# cnt mmc erase blk# cnt
mmc rescan
mmc part - lists available partition on current mmc device
mmc dev [dev] [part] - show or set current mmc device [partition] mmc list -
```

```
lists available devices
  mmc ecsd dump - dump ECSD values
  mmc ecsd read offset - read ECSD value at offset
  mmc ecsd write offset value - write ECSD value at offset
```

eMMC partitioning

eMMC v4.4 specification defines four physical (hardware) partitions:

- Boot 1
- Boot 2
- RPMB
- User Data

The size of the Boot and RPMB partitions is fixed in the chip and can be determined by reading information at the Extended CSD register. They are small partitions to hold a boot loader or secure parameters. The main storage is located at the User Data area.

ConnectCore 6 eMMC partitions are used for the following:

- **Boot 1**: holds the U-Boot boot loader.
- **Boot 2**: holds the U-Boot environment and its redundant copy.
- **RPMB**: not used.
- **User Data**: holds the operating system (divided in logical partitions).

The User Data area can be partitioned as any standard block device. In U-Boot, the command gpt allows you to write a GUID partition table (GPT) by passing a string with the partition attributes. GPT standard supersedes old MBR partition table allowing to define multiple partitions with names and universal unique identifiers (UUID).

There are two scripts (as environment variables) that use the gpt command to partition the internal eMMC with a default partition table for Linux or Android operating systems:

- $partition_mmc_linux
- $partition_mmc_android

You can view the partition table details by printing the contents of each variable with printenv.

If you want to modify the number, start offset, or size of the partitions you can edit the values of variables $parts_linux or $parts_android.

WARNING! These scripts partition the MMC device pointed to by variable $mmcdev. Never run these commands to partition a bootable microSD card. A Bootable microSD stores U-Boot on sector 2 and the GUID partition table uses the first 34 sectors, overwriting U-Boot and making the microSD card non-bootable. U-Boot in bootable microSD cards can only coexist with a DOS partition table.

Environment variables

There are several environment variables worth mentioning:

Variable	Description	Flags
ethaddr	MAC address of the first wired Ethernet interface	change-default
wlanaddr	MAC address of the WLAN interface	change-default
btaddr	MAC address of the Bluetooth interface	change-default

These variables are protected and will not be overwritten by **setenv** or **env default** command (unless manually forced with -f option).

On variants with eMMC, the MAC addresses are programmed by Digi during manufacturing and saved in the U-Boot environment on the eMMC.

On variants without eMMC, the MAC addresses are not programmed. You should program the MAC addresses of the available interfaces and save the environment on your boot storage media. The printed CC6 label contains the MAC addresses we assign to the unit.

```
=> setenv ethaddr 00:40:9D:AA:AA:AA
=> setenv wlanaddr 00:40:9D:BB:BB:BB
=> setenv btaddr 00:40:9D:CC:CC:CC
=> saveenv
```

After saving the environment, you cannot overwrite the MAC addresses using setenv or env default (unless forced with the -f option).

The following image shows the CC6 module label with the MAC addresses.

One-time programmable (OTP) bits (e-fuses)

The ConnectCore 6 has one-time programmable (OTP) bits. Most of the OTP bits have dedicated functionality and some are free to use. U-Boot supports access to the OTP bits through the **fuse** command.

See the CPU Reference Manual for information about the OTP bits.

WARNING! Programming the OTP bits is an irreversible operation that could potentially brick your module. Use the fuse command with extreme care.

HWID and MAC address OTP words

Digi saves MAC addresses in U-Boot environment variables (ethaddr, wlanaddr, btaddr). The two OTP words originally meant for the MAC address are reserved for Digi HWID, a unique identifier of the module.

Carrier board version

Since the ConnectCore 6 is an SMD module, customers can design their own carrier boards. Carrier boards often suffer redesigns and it is useful for the software to be able to determine the version of the carrier board it is running on. This allows the user to have conditional code depending on the board's version (like enabling different GPIOs or using different IOMUX).

Digi U-Boot has built-in support to program and query the carrier board version number on the OTP bits.

Coding the carrier board version on the OTP bits

First, your board's include file should define:

```
#define CONFIG_HAS_CARRIERBOARD_VERSION
```

Second, you must define which OTP bits you will use to code the carrier board version. This is done by defining the following constants on your board's include file:

```
#define CONFIG_CARRIERBOARD_VERSION_BANK
#define CONFIG_CARRIERBOARD_VERSION_WORD
#define CONFIG_CARRIERBOARD_VERSION_MASK
#define CONFIG_CARRIERBOARD_VERSION_OFFSET
```

For the SBC carrier board, Digi uses the lower 4 bits of the OTP General Purpose 1 register (GP1) which corresponds to Bank 4, Word 6 with a mask 0xf (four bits) and an offset of 0 (lower four bits). Not counting the value of 0x0, these four bits allow you to code up to 15 board versions.

You can use this layout for your own carrier board or a different one.

Querying the carrier board for conditional code

ConnectCore 6 module common source code defines function **get_carrierboard_version()** which can be used to create conditional code basing on the carrier board version, for example:

```
switch (get_carrierboard_version()) {
case 1:
       /* Code for carrier board version 1 */
       break;
case 2:
       /* Code for carrier board version 2 */
       break;
default:
        /* Code for other (or undefined) carrier board version */
        break;
}
```

Auto bootscript

The bootscript is a script that is automatically executed when the boot loader starts, before U-Boot's default boot command **bootcmd**.

The bootscript allows the user to execute a set of predefined U-Boot commands automatically, before proceeding with normal boot. This is especially useful for production environments and targets which don't have an available serial port for showing the U-Boot monitor.

Bootscript process

The bootscript works in the following way:

1. U-Boot checks the variable **bootscript**. If it exists U-Boot tries to download the filename referred by the variable from the TFTP server IP address defined at variable **$serverip** (by default 192.168.42.1). If the bootscript file is successfully downloaded, it is executed.
2. If any of the commands in the bootscript fails, the rest of the script is cancelled.
3. When the bootscript has been fully executed (or cancelled) U-Boot continues normal execution.

   To cancel the automatic bootscript download, erase the **$bootscript** variable by setting it to nothing:

```
=>  setenv bootscript
```

```
=>    saveenv
```

WARNING! On the ConnectCore 6, the variable **bootscript** is undefined by default. For more information, please refer to the PHY autonegotiation delay section.

Creating a bootscript

To create a bootscript file create a plain text file with the sequence of U-Boot commands. It is recommended that the last command erases the variable bootscript to avoid the bootscript from executing a second time.

For example, create a file called myscript.txt with the following contents:

```
setenv company digi
setenv bootdelay 1
printenv company
setenv bootscript
saveenv
```

This script creates a new variable called **company** with value **digi** and sets the bootdelay to one second. Finally it erases the variable **bootscript** so that U-Boot doesn't try to execute the bootscript in the future, and saves the changes.

1. Execute the **mkimage** tool (provided with U-Boot in the **tools** subdirectory) with the file above as input file. Syntax is: mkimage -A arm -T script -n "Bootscript" -C none -d <input_file> <output_file>

   The name of the output file must be in the form **<platformname>-boot.scr**, where <platformname> must be replaced with your target's platform name.

For example, to create the bootscript from the text file above and for a ConnectCore 6 SBC platform, go to the U-Boot directory and execute:

```
tools/mkimage -A arm -T script -n "Bootscript" -C none -d myscript.txt
ccimx6sbc-boot.scr
```

Updating the boot loader

It is possible to update the boot loader in the storage media using the **update** command. For example, to update from TFTP using the default filename (in variable **uboot_file**):

```
=> update uboot tftp
```

WARNING! Writing an invalid boot loader file may lead to the target not booting.

Digi will release U-Boot updates from time to time to fix problems or add new functionality.

Much of the custom functionality added to U-Boot depends on environment variables and scripts that may have new values in newly released versions.

Generally, after upgrading U-Boot, it is recommended to reset the environment to its defaults using the following command.

```
=> env default -a
```

This will reset the whole environment, with the exception of protected variables (like the MAC addresses).

After resetting the environment, you may need to adjust your manufacturing or boot scripts to accommodate to changes in the default environment, like new or modified scripts, variables and default filenames

Recovering a bricked device

If for any reason the bootloader has been erased from the storage media (or written with an invalid image) and the target does not boot, we can boot the target from an alternative source, like the uSD card.

The ConnectCore 6 Getting Started Guide documentation (available on http://www.digi.com/products/wireless-wired-embedded-solutions/solutions-on-module/connectcore/connectcore-imx6#docs) contains instructions to create a full bootable uSD card with a complete Linux or Android operating system. The following chapter will only show you how to create a bootable uSD card if you have a U- Boot boot loader binary image file (for example: u-boot-ccimx6sbc.imx).

Note The ROM loader of the CPU expects to find U-Boot on sector 2 of the block device (offset of 1024 bytes).

Create a bootable microSD card from a U-Boot image

You might need root/Administrator permissions.

WARNING! The following procedure will destroy existing data in the uSD card.

Using Linux

1. Insert the uSD card to your computer and check the node Linux assigns to it (**/dev/[sdcard]**) using dmesg:

```
  dmesg
  [1413652.901270] sd 41:0:0:0: [sdc] 7744512 512-byte logical blocks:
(3.96 GB/3.69
  GiB)
  [1413652.903140] sd 41:0:0:0: [sdc] No Caching mode page present
  [1413652.903144] sd 41:0:0:0: [sdc] Assuming drive cache: write
through
  [1413652.905638] sd 41:0:0:0: [sdc] No Caching mode page present
  [1413652.905642] sd 41:0:0:0: [sdc] Assuming drive cache: write
through
  [1413652.915154] sdc: sdc1
```

2. Do not mount any partitions the card might contain (or unmount any partition if automatically mounted) as you will be writing to the entire block device.
3. Write the U-Boot image file to the uSD card with this command:

```
  sudo dd if=[path/filename.imx] of=/dev/[sdcard] bs=512 seek=2
oflag=sync
```

where

- **[path/filename.imx]** must be substituted with the path and filename to the U-Boot image.
- **/dev/[sdcard]** must be substituted with the device node assigned by Linux to your uSD card.

WARNING! An incorrect device node here might destroy all data on your computer hard drive).

The uSD card is now ready.

Using Windows

1. Download and uncompress the **dd** application for Windows from www.chrysocome.net/dd
2. Open a Command Prompt console in Windows.
3. Insert the uSD card to your computer.

4. Change to the directory where the **dd** application was uncompressed and run it to list the available drives and identify the device node assigned to your SD card (follow the **dd** documentation on the web page):

```
dd --list
```

5. Copy the U-Boot image file to the same directory where the **dd** program is.
6. Write the U-Boot image file to the uSD card with this command:

```
dd if=[filename.imx] \\.\Volume{UUID}\ bs=512 seek=2
```

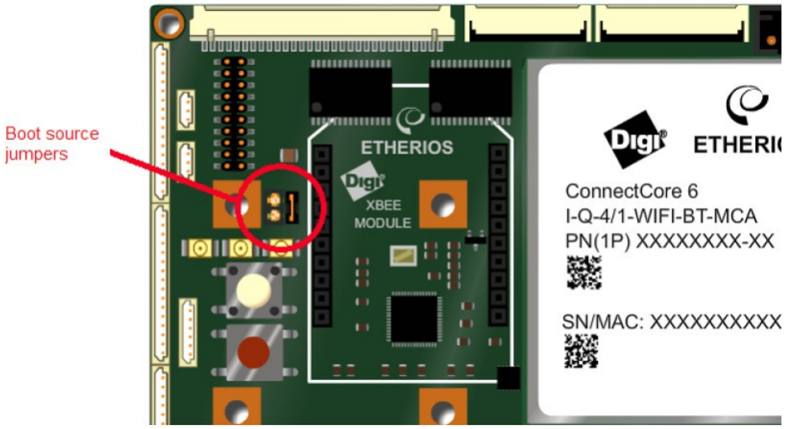
where

- **[filename.imx]** must be substituted with the filename of the U-Boot binary image.
- **UUID** must be substituted by the identifier given by Windows to the uSD card on the --list command.

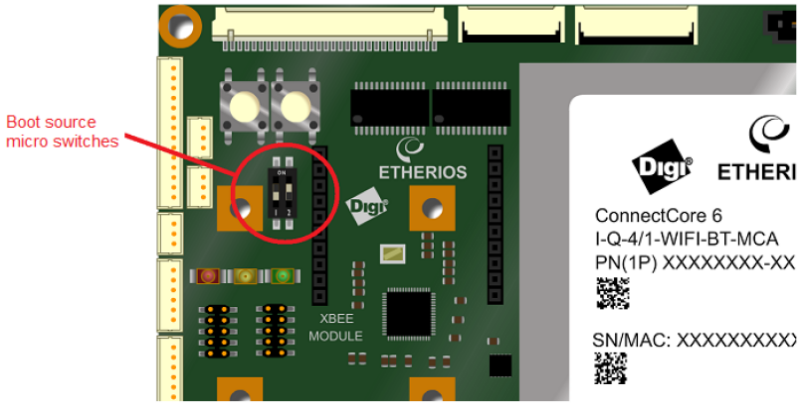
The uSD card is now ready.

Boot from a uSD card

1. Insert the uSD card on the uSD card holder.
2. Change the boot source configuration to boot from the uSD card:
    a. If using an SBC version1 (30014752-01), set the boot source jumpers: J4: open, J5: closed.



    b. If using an SBC version2 (30014752-02), set the boot source microswitches: SW3.1: OFF, SW3.2: ON.



3. Power up the board.

    When booted from uSD card, U-Boot reads/writes the environment from the uSD card as well, at an offset of 0x1C0000 bytes.

WARNING! Such a microSD can hold a DOS partition table but not a GPT partition table. The first partition should start at an offset of 2MiB minimum, to avoid overwriting U-Boot or its environment.

Known issues

PHY autonegotiation delay

The Micrel PHY KSZ9031 on the SBC carrier board may take between 5 and 6 seconds to autonegotiate with Gigabit switches. To avoid this long delay during boot, the variable bootscript is undefined by default and the auto bootscript feature does not run.

To speed up the PHY's autonegotiation, you can:

- Use a 10/100 switch (not Gigabit).
- Force the Micrel PHY to work as master during master/slave negotiation by setting variable **phy_mode** to **master**.