MICROCHIP

AN2748

SAMA5D2 Linux® Secure Boot

Introduction

The SAMA5D2 family of MPUs are designed to be booted in one of two different modes – Normal Boot, and Secure Boot.

- Normal Boot mode is used to load an unencrypted/unsigned program from external memory at boot time. This mode of operation is fine for many designs, and is preferred for development because there are fewer steps required between making a code modification and running the code.
- Secure Boot mode is used to load encrypted/signed programs at boot time. This mode is used when the design needs to guarantee that the image that is loaded at boot time is authentic, and is authorized to be run on the secure system. Some of the software is also encrypted to keep the contents hidden.

This application note describes how to boot the Linux kernel as a secure application using a SAMA5D2 MPU. Secure boot helps prevent unauthorized software from being booted on the SAMA5 MPU.

This application note was written for a SAMA5D2-RevC Xplained board, but can be tailored to any SAMA5D2 system.

Reference Documents

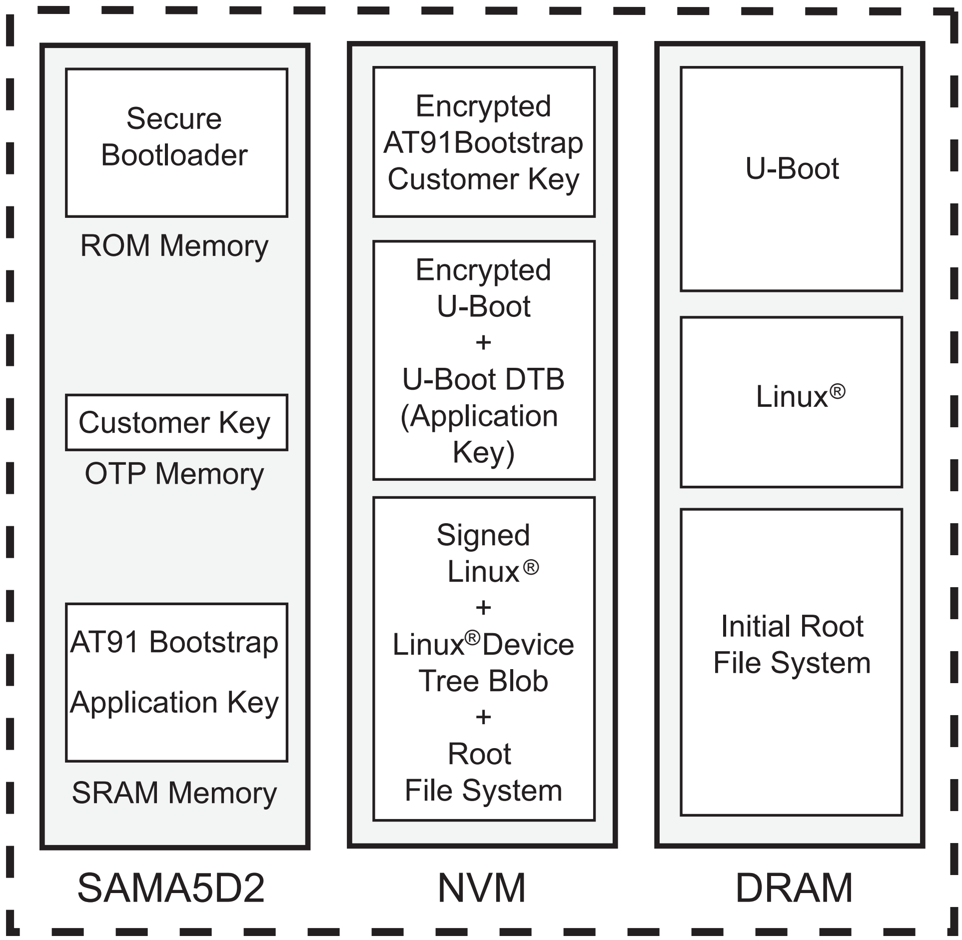
- *SAMA5D2 Series* datasheet (Lit. No. DS60001476). Available on www.microchip.com.
- *SAMA5D2 Series Secure Boot Strategy* application note (AN2435, Lit. No. DS00002435). Available under Non-Disclosure Agreement (NDA) from your local Microchip sales office.
- *SAMA5D2C (Rev. C) Xplained Ultra Evaluation Kit* user's guide (Lit. No. DS50002691). Available on www.microchip.com.
- Secure-sam-ba-cipher-3.2 readme
- Secure-sam-ba-loader-3.2 readme
- AT91Bootstrap source code
- U-Boot documentation (in ./doc/uImage.FIT directory)

Table of Contents

1. Software Components of the System

- ROM code (first stage loader)
- AT91bootstrap bootloader (second stage loader)
- U-Boot bootloader (optional third stage loader)
- Linux kernel/device tree blob
- Root file system

Figure 1-1. Boot Components



1.1 ROM Code

The boot ROM code begins execution when the processor comes out of reset. Before loading a boot image, the ROM configures:

- Arm® supervisor stack
- PLLA (Using 12 MHz fast RC clock as input)
- Processor Clock (PCK) and Master Clock (MCK)

The ROM code determines the boot sequence by reading a bit in the Boot Configuration Word in the fuse area. It then loads the second stage bootloader from non-volatile storage into the on-chip SRAM. The details of ROM code configuration and operation are found in Application Note *SAMA5D2 Series Secure Boot Strategy* (see Reference Documents).

1.2 AT91bootstrap Bootloader

The second stage bootloader, AT91bootstrap, is responsible for initializing the SDRAM, and loading either a third stage bootloader, or the Linux kernel and device tree blob. The second stage bootloader can reside in one of the following NVM locations:

- SDMMC (1 and 0)
- NAND Flash
- Serial Flash (0 and 1)
- QSPI Flash (0 and 1)

1.3 U-Boot Bootloader

U-Boot is a very powerful and flexible bootloader that can load applications from a wide variety of sources. A third stage bootloader, such as U-Boot, can be used if more features are required to boot an image than are provided by AT91bootstrap, such as booting from a network, or booting from a device or filesystem that is not supported in the second stage bootloader. Similar to the Linux kernel, U-Boot uses a Flattened Device Tree to configure features such as GPIO, serial ports, and other hardware devices used during the bootload process.

1.4 Linux Kernel

The Linux kernel is the main component of a Linux system. It handles most of the critical functions of a computer system such as system resource management, scheduling, processor interrupts, virtual memory, and device drivers. The kernel also has several different interfaces to normal “user” programs.

The Linux kernel is loaded by the AT91bootstrap second stage loader, or by the U-Boot third-stage loader into SDRAM.

1.5 Device Tree Blob

The Linux kernel uses a Flattened Device Tree to describe the hardware components of the system it is running on. During boot, a device tree binary (dtb) file is loaded into memory by AT91bootstrap or U-Boot prior to kernel execution.

1.6 Root File System

A root file system is required by the Linux kernel. The root file system contains all the necessary libraries, programs, utilities, and device nodes that the Linux system requires for normal use. This application note shows the protection of an initial root filesystem by including the Linux initRAM filesystem into the kernel image.

2. Secure Boot Tasks

Now that all the pieces of secure boot have been introduced, we can describe the activities necessary to enable a secure boot system.

1. Prepare cryptographic material. Keys and certificates need to be generated and placed in the proper locations.
2. Prepare the software to be loaded (Linux kernel, Linux Device Tree, and bootloader programs).
3. Second stage bootloader (AT91bootstrap) is encrypted and signed.
4. Third stage bootloader (U-Boot) is configured, public keys stored, encrypted and signed.
5. Linux kernel and device tree are packaged into a signed FIT file and properly signed.
6. The SAMA5D MPU is set to Secure Boot mode.
7. Keys are loaded into the MPU.
8. The Secure Boot mode and keys can be permanently programmed into device fuses if desired.

3. Cryptography Usage in Secure Boot

3.1 Encryption

For the Linux secure boot environment, two programs are encrypted: the AT91bootstrap program, and U-Boot. The AT91bootstrap program is encrypted to prevent access to the keys that are used to authenticate U-Boot. If AT91bootstrap was not encrypted, it would be fairly easy to forge the next stage of software in the boot process. This is because AT91bootstrap uses a symmetric key to perform the authentication.

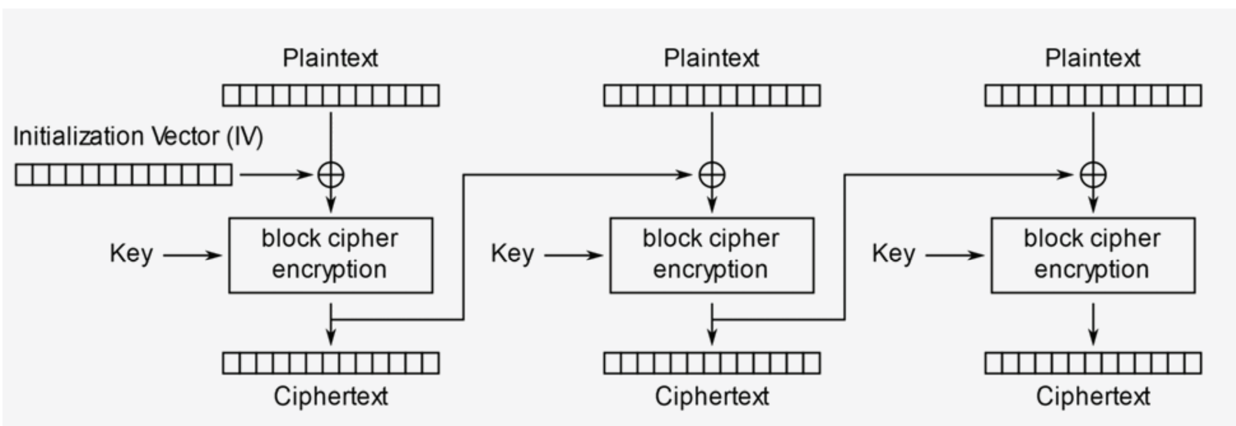
The algorithm used for encryption of both the AT91bootstrap and U-Boot is AES. AES encryption is a symmetric algorithm, which means that the same key is used for encryption as well as decryption. AES keys can be 128, 192 or 256 bits long. The key must be shared between the system that encrypts the data as well as the system that decrypts the data.

AES is a block cipher, which means that it operates on a block of data rather than a data stream. The size of a block is 128 bits, regardless of the size of the key being used. Stream ciphering can use AES block-level encryption when combined with another mechanism such as chaining.

Cipher Block Chaining (CBC) mode is used by the secure boot ROM code as well as by the AT91bootstrap to encrypt/decrypt the firmware that runs on the system.

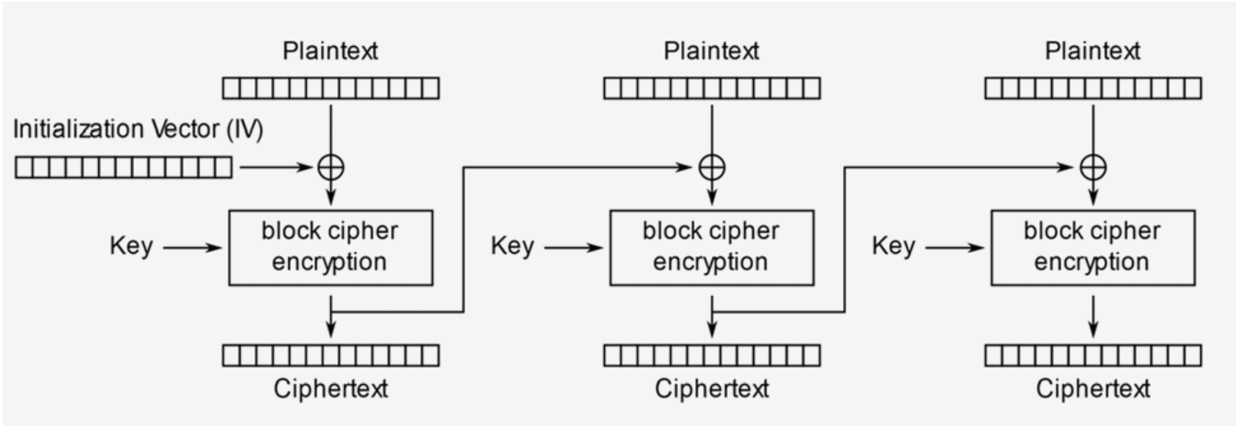
The image is encrypted using AES-CBC mode.

Figure 3-1. Cipher Block Chaining Mode Encryption



The boot ROM and the bootloader use AES-CBC decryption, which is described below.

Figure 3-2. Cipher Block Chaining Mode Decryption



3.2 Authentication

An important feature of secure boot systems is the ability to verify that the image to be booted is from a trusted source, and has not been tampered with or corrupted in some way. Authentication and integrity are determined by appending a Message Authentication Code (MAC) to the end of an image. Appending a MAC to an image is also known as “signing” an image. The ROM code checks the signature of the AT91bootstrap program to make sure it is authentic. In turn, the AT91bootstrap program checks the signature of the U-Boot to make sure it is authentic. U-Boot in turn, checks the signature of the Linux image, the Linux Device Tree, and the root filesystem.

A MAC can be created by one of several methods. Two common methods are 1) HMAC – Hash-based Message Authentication Code, and 2) CMAC – Cipher-based Message Authentication Code. SAMA5Dx MPUs use either the AES-CMAC algorithm, or a RSA + SHA-256 HMAC to perform this check. The AT91bootstrap program uses AES-CMAC to authenticate U-Boot.

4. Development Flow

The flow of this application note will be presented in a somewhat reversed order – starting from a completely open, unencrypted, unsigned platform; then securing pieces “upstream” from the running operating system back to the ROM code. This allows the addition of security features one layer at a time, and prevents having to get everything working together at the beginning.

This application note relies on the Buildroot tool to manage the process of building a working SDCard image that can be run on the SAMA5D2 development board.

4.1 Create a Working SDCard Image

The first step is to checkout Buildroot and build a default image.

```
$ git clone https://git.buildroot.net/buildroot
$ cd buildroot
$ git checkout 2018.02.x
$ make atmel_sama5d2_xplained_mmc_defconfig
$ make menuconfig
```

Modify the version of U-Boot that is used. Go to Bootloaders→Uboot Version (Custom), then select 2018.03.

```
$ make
```

This creates a working SDCard image for the SAMA5D2 Xplained board. Copy the image to an SDCard and make sure it works on the SAMA5D2_Xplained board.

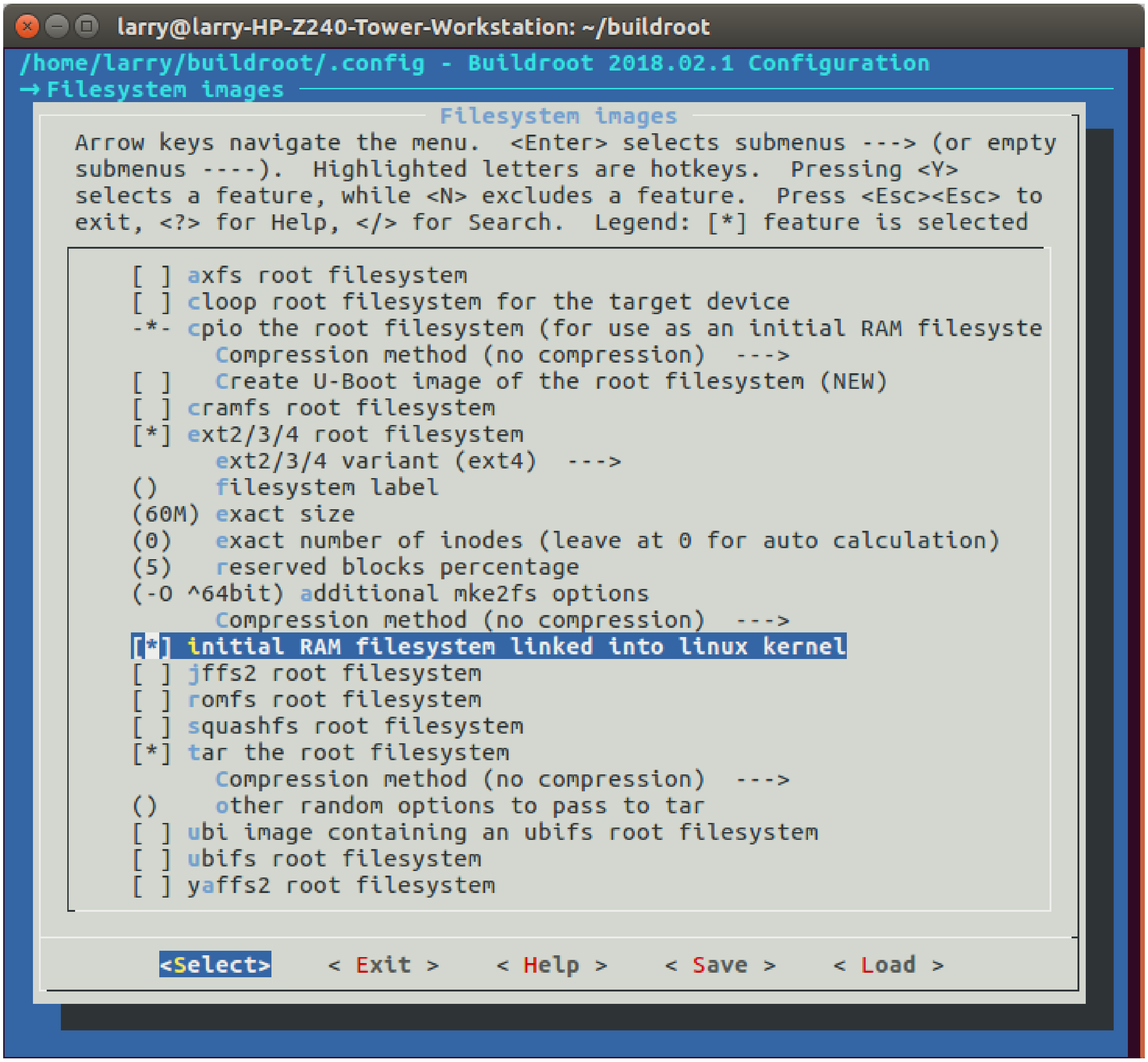
The name of the image is output/images/sdcard.img

4.2 Add Initial RAM Filesystem

One of the easiest ways to authenticate a root filesystem is to include the root filesystem into the Linux kernel binary. Buildroot has a configuration option for this.

```
$ make menuconfig
```

Go to Filesystem Images and select “initial RAM filesystem linked into kernel”.



The Linux kernel needs to be reconfigured manually before the initial RAM filesystem will work properly. Build the image and test it again to make sure the root filesystem works.

```
$ make linux-reconfigure
```

```
$ make
```

Now that we have a working Linux image, we need to enable U-Boot's Verified Boot mechanism.

5. U-Boot Verified Boot

In the U-Boot documentation, there is a description of Verified Boot. Verified Boot is the process that U-Boot uses to verify that an image is correct and is allowed to run on the platform. Using Verified Boot requires:

1. Special configuration
2. Creating keys and certificates
3. Storing the public key in the U-Boot control DTB
4. Creating a FIT image
5. Signing the FIT image

This application note uses the “signed configuration” methodology of Verified Boot. The FIT file contains two images that are hashed using SHA256, and a configuration that is hashed using SHA256 and encrypted using RSA encryption.

5.1 U-Boot FIT Images

Flattened Image Tree (FIT) files are special instances of Flattened Device Tree (FDT) files. Instead of describing hardware, U-Boot FIT files contain files and metadata in the form of nodes and properties that are used to boot applications such as Linux. The FIT files for Verified Boot contain a Linux kernel, Linux device tree, configuration data, and hashes that U-Boot uses to authenticate the data contained within the FIT file. U-Boot uses the “bootm” command to verify and boot an image in the FIT format.

5.2 Configuring U-Boot

Since U-Boot is extremely flexible, there are many features that should be carefully evaluated to help ensure security. One configuration setting, CONFIG_FIT_SIGNATURE, allows image signature checking and is the heart of U-Boot’s Verified Boot methodology. Without the CONFIG_FIT_SIGNATURE setting, it is not possible to check, or even generate, a properly signed FIT file.

Buildroot provides an easy mechanism to configure packages using the syntax:

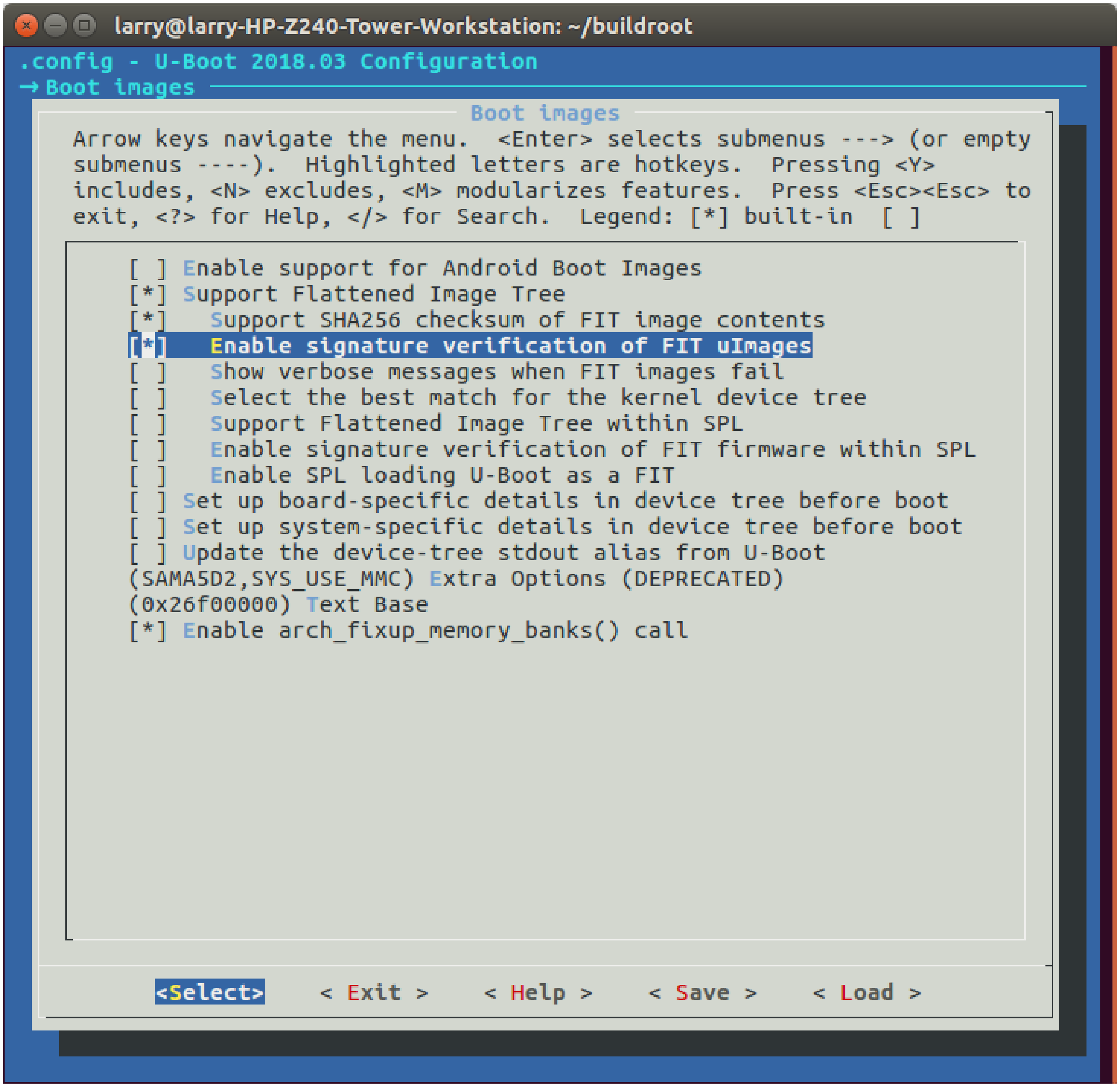
```
$ make <package name>-menuconfig
```

To run menuconfig for U-Boot simply type the command:

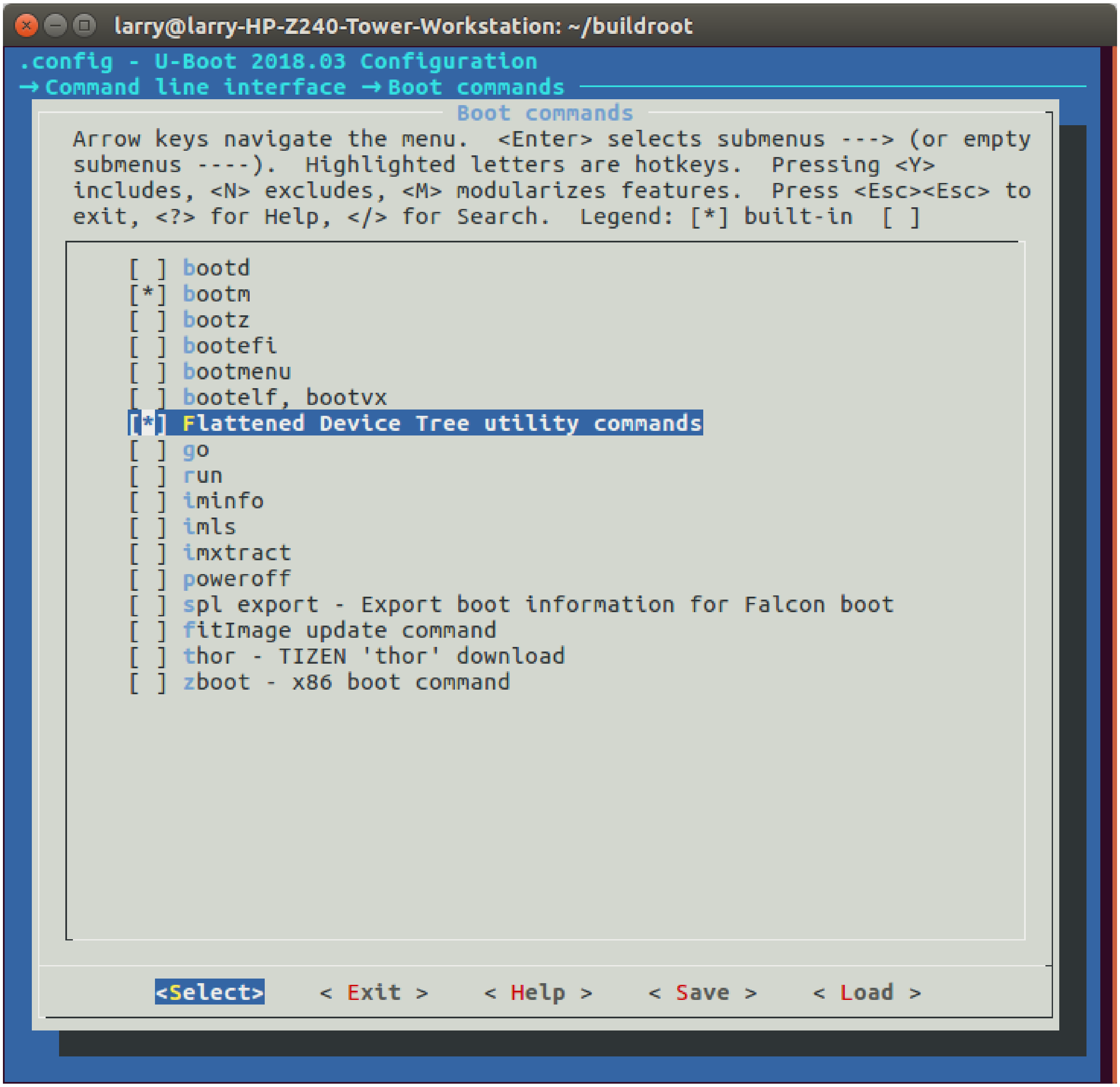
```
$ make uboot-menuconfig
```

The following menuconfig screenshots show the various parameters that will be modified for secure boot.

CONFIG_FIT_SIGNATURE is set by selecting “Boot images”->“Enable signature verification of FIT uImages”.



Under “Command Line Interface”→”boot commands”, make sure bootm command is enabled as well as “Flattened Device Tree utility commands”.



U-Boot is now able to perform checking of FIT images that are signed, but at this point, there is still some information missing. U-Boot has the default control DTB that does not contain the public key required to check the signature of images. As seen below, running without the correct U-Boot control DTB produces a message such as “Verifying Hash Integrity … OK”. Although this message looks correct, there is no signature testing. The hash is correct, but for secure boot, there needs to be an RSA step also. The following section describes how to get the correct keys into the U-Boot control DTB.

```
=> bootm 0x20000000
## Loading kernel from FIT Image at 20000000 ...
   Using 'conf@1' configuration
   Verifying Hash Integrity ... OK
   Trying 'kernel@1' kernel subimage
     Description:  unavailable
     Type:         Kernel Image
     Compression:  uncompressed
```



```
     Data Start:   0x200000c4
     Data Size:    4938272 Bytes = 4.7 MiB
     Architecture: ARM
     OS:           Linux
     Load Address: 0x21000000
     Entry Point:  0x21000000
     Hash algo:    sha256
     Hash value:   9edcd903bc251f78c1eda0d32f175140a172121d1a472d3f50f549b758f8a78a
   Verifying Hash Integrity ... sha256+ OK
## Loading fdt from FIT Image at 20000000 ...
   Using 'conf@1' configuration
   Trying 'fdt@1' fdt subimage
     Description:  unavailable
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x204b5bbc
     Data Size:    31955 Bytes = 31.2 KiB
     Architecture: ARM
     Hash algo:    sha256
     Hash value:   9aa95c7aed6509bb294feb21dd904293620ef9e860f5ad6255771538945e4c39
   Verifying Hash Integrity ... sha256+ OK
   Booting using the fdt blob at 0x204b5bbc
   Loading Kernel Image ... OK
   Loading Device Tree to 3f95e000, end 3f968cd2 ... OK
Starting kernel ...
```

5.3 Creating RSA Signing Credentials

This application note will not go into the details of Public Key Infrastructure (PKI), but will introduce and describe the components necessary to perform a secure boot. When performing PKI operations, there are several important components that must be obtained or constructed ahead of time. These are:

- RSA Root key and certificate – used to sign “signing” certificates
- RSA Signing key and certificate – used to sign images

Although not enforced by the tools used by U-Boot for signing images, certificates for signing code should be different than certificates used to sign certificates. For this application note, we will generate a single root CA certificate, and a single code-signing certificate. Generally, a root CA certificate is created on a machine that is not on a network. The two pieces created are:

- CA private key
- CA Certificate

Private keys should never be shared. With a private key, anyone could create and sign a certificate that is forged, but can be used as authentic.

5.4 OpenSSL Configuration File

Before creating the certificates that will be used to sign the code images, the default OpenSSL configuration file should be modified. The modifications are to change the default values for the “distinguished name” found in certificates, and also to create “key usage” values for the code signing certificate.

The procedure is as follows:

1. Create a directory to store the configuration file, keys, and certificates.

```
$ mkdir keys
$ cd keys
```

2. Copy the OpenSSL default configuration:

```
$ cp /etc/ssl/openssl.cnf .
```

3. Edit the configuration file (openssl.cnf) to fill in the req_distinguished_name section with values that represent your organization:

```
[ req_distinguished_name ]
countryName         = Country Name (2 letter code)
countryName_default     = US
countryName_min         = 2
countryName_max         = 2
stateOrProvinceName     = State or Province Name (full name)
stateOrProvinceName_default = Arizona
localityName            = Locality Name (eg, city)
localityName_default    = Chandler
0.organizationName      = Organization Name (eg, company)
0.organizationName_default  = Microchip Technology
```

4. Create a section named “code_sign” with the following fields:

```
[ code_sign ]
basicConstraints=CA:FALSE
keyUsage = digitalSignature
extendedKeyUsage = codeSigning
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer
```

5.5 Create a CA Certificate and Key

The following command can be used to create a CA certificate that is valid for 10 years:

```
$ openssl req -x509 -newkey rsa:4096 -keyout cacert.key -out cacert.crt -days 3652 -sha256 -
config openssl.cnf
```

This command will prompt for the distinguished name fields as well as asking for a password to use to encrypt the private key. Since the configuration file was modified to include most of the distinguished name default values, pressing enter is fine for most items. Be sure to enter an appropriate string for the “Common Name” field. In the example below, the Common Name is “Test CA”

Here is some sample output of the req command. Notice that the PEM pass phrase and Common Name are inputs.

```
$ openssl req -x509 -newkey rsa:4096 -keyout cacert.key -out cacert.crt -days 3652 -sha256 -
config openssl.cnf
Generating a 4096 bit RSA private key
.............................................................++
................................++
writing new private key to 'cacert.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:
State or Province Name (full name) [Arizona]:
Locality Name (eg, city) [Chandler]:
Organization Name (eg, company) [Microchip Technology]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:Test CA
Email Address []:
```

The options used by the “req” command are as follows:

Table 5-1. req Command Options

Option	Description
`-x509`	A self-signed CA certificate is created instead of a certificate request.
`-newkey rsa:4096`	Create a RSA 4096-bit private key.
`-keyout cacert.key`	Write the CA private key to “cacert.key”.
`-out cacert.crt`	Write the CA certificate to “cacert.crt”.
`-days 3652`	The certificate is valid for 3652 days.
`-sha256`	Use SHA256 hash for the signature algorithm.
`-config openssl.cnf`	Use “openssl.cnf” as the configuration file.

5.6 Create a Certificate Request and Private Key

```
openssl req -nodes -newkey rsa:4096 -keyout samkey.key -out samkey.csr -sha256 -config
openssl.cnf
```

This command does not have the -x509 option, therefore it will create a certificate-signing-request (CSR) and not a self-signed certificate. This command also specifies the -nodes option that causes the private key to be unencrypted. U-Boot currently does not support password-protected RSA keys. This will probably change in the future.

```
$ openssl req -nodes -newkey rsa:4096 -keyout samkey.key -out samkey.csr -sha256 -config
openssl.cnf
Generating a 4096 bit RSA private key
...........................................................++
..............................++
writing new private key to 'samkey.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:
State or Province Name (full name) [Arizona]:
Locality Name (eg, city) [Chandler]:
Organization Name (eg, company) [Microchip Technology]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:U-Boot Image Signing
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

5.7 Sign the Certificate Request

```
openssl x509 -req -in samkey.csr -days 365 -CA cacert.crt -CAkey cacert.key -CAcreateserial -
out samkey.crt -extfile openssl.cnf -extensions code_sign
```

This command uses the previously created CA certificate to sign the samkey.csr certificate request. U-Boot requires the names of the key and certificate to be identical, except the filename extension of the key must be .key, and the certificate .crt. The signer must input the pass phrase of the CA private key to successfully sign the certificate request.

```
$ openssl x509 -req -in samkey.csr -days 365 -CA cacert.crt -CAkey cacert.key -CAcreateserial
-out samkey.crt -extfile openssl.cnf -extensions code_sign
Signature ok
subject=C = US, ST = Arizona, L = Chandler, O = Microchip Technology, OU = MPU32
Applications, CN = U-Boot Image Signing
Getting CA Private Key
Enter pass phrase for cacert.key:
```

5.8 Check the Signing Certificate

If you wish to check the CA-signed certificate, the “x509” command can be used to print the certificate in human-readable form.

```
$ openssl x509 -in samkey.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            81:9b:fb:48:3b:da:22:94
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, ST = Arizona, L = Chandler, O = Microchip Technology, CN = Test CA
        Validity
            Not Before: Apr 30 03:26:41 2018 GMT
            Not After : Apr 30 03:26:41 2019 GMT
        Subject: C = US, ST = Arizona, L = Chandler, O = Microchip Technology, CN = U-Boot
Image Signing
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:ed:04:19:ea:79:01:f8:ef:74:ff:5c:01:20:50:
                    b3:f0:6f:03:10:62:66:9c:59:95:c6:ff:4c:5d:be:
                    33:c5:3c:36:28:c7:44:0c:95:91:66:e8:0e:5d:81:
                    7f:11:95:f6:c8:37:0d:5e:30:2c:0f:d5:84:0f:14:
                    b2:b3:2b:72:b0:65:6f:15:97:31:38:26:88:15:34:
                    d4:4e:20:7c:94:1c:35:9b:ec:62:06:5d:7f:6f:db:
                    93:12:b5:e9:5c:fd:45:a8:9c:5a:10:a7:37:09:2b:
                    d1:35:19:d3:5e:8e:92:4e:9a:53:a5:ad:11:41:e0:
                    06:4b:37:7e:77:c8:82:81:7b:ee:5f:be:71:4f:18:
                    66:d4:fe:2a:74:55:ef:ab:dc:df:80:44:6c:c8:f8:
                    5f:5c:da:76:04:08:68:7c:4b:ec:68:06:81:a0:e8:
                    f5:6e:57:01:38:66:43:9c:e2:4a:d2:d2:ed:f6:64:
                    3c:99:04:f4:af:8b:46:bd:90:45:52:41:1a:45:f5:
                    48:77:28:b0:a8:9f:00:e2:5b:8a:00:d4:01:8c:6f:
                    09:ea:58:a1:2e:73:9d:d2:44:b9:e8:b7:d7:9b:5e:
                    24:a5:2d:86:65:2f:66:ec:41:83:5a:18:e5:47:d3:
                    d1:9a:0d:5d:b9:cd:c2:bd:71:a1:39:ec:5b:88:ad:
                    1b:f7:99:7f:0b:de:89:59:c5:26:9c:de:00:8f:41:
                    76:0b:c2:76:8d:31:84:7f:1b:c4:6f:f7:2d:45:45:
                    12:80:cb:54:ae:f7:65:15:55:dc:d8:c0:9d:74:27:
                    81:00:4a:f2:a9:31:9f:4b:fc:01:16:9a:50:df:96:
                    4d:26:30:f4:4c:11:6e:02:f8:cb:8a:80:37:81:83:
                    84:81:ac:91:46:97:ea:a2:9f:46:36:56:4c:6d:c9:
                    c6:f8:42:e2:e9:90:97:49:bf:9a:5d:21:2d:50:74:
                    bf:ef:a0:53:c4:7b:4a:e6:18:6c:1d:77:ee:94:fc:
                    4c:c5:ae:06:20:32:52:b5:0c:0c:2d:a8:56:76:1b:
                    b8:0c:d6:81:b3:1e:26:1a:f8:69:3f:97:a8:c9:61:
```

```
                     c0:66:55:cf:b4:de:67:9e:c5:ad:a7:88:16:3b:c6:
                     e0:47:82:01:92:e4:1f:b1:d3:dd:7f:70:1a:ab:47:
                     9f:50:98:21:3e:92:fe:45:d4:b0:fd:6f:01:dc:01:
                     d5:f9:9e:57:5e:e1:17:70:c9:42:05:8c:c8:bf:8f:
                     26:fd:19:e7:9e:89:cf:f4:25:6f:40:d8:37:4d:0f:
                     61:5c:5b:58:9e:16:d1:5b:43:d3:c4:3e:81:b1:1c:
                     43:73:b2:11:fe:a2:6f:a1:f5:a9:f5:73:33:99:02:
                     86:ff:57
                 Exponent: 65537 (0x10001)
         X509v3 extensions:
             X509v3 Basic Constraints:
                 CA:FALSE
             X509v3 Key Usage:
                 Digital Signature
             X509v3 Extended Key Usage:
                 Code Signing
             X509v3 Subject Key Identifier:
                 1E:C2:6A:E1:57:03:C2:DC:BB:DD:34:1E:CD:65:49:0C:56:FC:B7:3E
             X509v3 Authority Key Identifier:
                 keyid:BC:C6:9B:7C:56:98:B7:CB:37:34:24:BE:A8:CD:B1:BD:9D:66:B7:AF
     Signature Algorithm: sha256WithRSAEncryption
          cd:ba:53:1f:31:b0:3c:51:d3:0f:dc:87:e4:d0:b5:1c:55:98:
          21:8b:6e:34:29:44:a2:24:14:08:26:0f:d8:74:42:94:44:09:
          b7:64:15:ee:5d:b4:6c:5e:dc:7e:5c:0a:c1:00:17:4f:4c:c1:
          4a:93:b9:61:21:a6:55:a4:c8:c8:f9:75:c8:29:f3:4f:cf:0b:
          d0:60:95:0f:de:27:21:ee:5e:68:8d:af:17:6b:ff:a8:06:9f:
          3e:31:05:3d:e4:d3:a8:db:97:5f:d7:75:47:94:ea:31:f5:3b:
          87:03:a9:47:cc:36:4e:38:0a:9b:c7:4e:3a:a4:18:3e:42:2f:
          75:60:3d:07:66:89:85:7c:45:b4:a8:a4:c8:87:34:45:dd:99:
          21:37:78:5f:06:f5:f7:f0:e5:5b:eb:7b:ea:59:8a:a6:0d:3d:
          18:ab:3b:fb:71:49:2b:9d:7d:2c:99:3d:42:06:91:4a:5f:42:
          37:c0:5b:ba:df:0b:df:47:80:e5:c8:cd:9c:52:94:a4:cf:c1:
          aa:11:e9:db:83:e2:2b:e9:42:ae:3d:c5:20:ad:2c:22:c8:15:
          68:7d:3a:e8:69:21:a3:bb:f0:6e:8e:8a:e3:a8:f4:11:20:7d:
          eb:e3:23:92:b6:27:28:f6:c1:da:e9:ab:1e:71:95:75:f4:7a:
          52:c2:91:b2:56:da:f0:74:c4:3a:53:3d:77:d9:65:d9:9c:1f:
          8e:2a:90:0f:c9:8a:c8:ab:7b:4f:58:d2:71:49:a5:e8:cb:16:
          53:56:a1:a3:73:d1:1b:3a:8d:e2:9c:ef:26:b5:75:9b:17:3c:
          1c:70:a8:6d:d4:cb:37:0e:4a:a2:dc:5e:15:38:6d:05:96:d8:
          6e:8d:8a:13:86:de:a6:fc:1b:65:f8:85:e5:3b:92:1a:61:cc:
          b2:8f:50:2a:73:ba:8b:36:85:6a:19:98:8a:3f:0b:ca:69:e5:
          59:33:02:c1:75:e3:82:ed:d4:c7:03:da:ab:94:c9:24:ec:00:
          2c:96:a1:56:0e:f7:50:9a:1b:68:c0:83:f7:7f:85:d8:b0:81:
          2e:24:58:c9:6a:c8:c4:3e:85:4c:7f:9e:6c:fa:f6:4e:bd:12:
          37:08:6b:91:92:50:78:8f:09:db:cb:f4:ce:a8:fe:5d:33:0e:
          a6:0f:5f:7b:bf:f4:99:96:f2:54:f7:14:54:a8:88:b0:f4:3d:
          c3:85:21:ee:ef:13:53:64:cd:a8:fd:14:da:6b:ee:cf:57:56:
          43:8f:26:75:aa:fc:75:03:3f:b9:80:60:70:2c:5c:b7:1d:b1:
          b4:82:df:cb:4d:0f:92:5d:80:30:63:49:a4:1a:78:ef:ce:9c:
          ff:f6:cf:6e:59:7a:1e:58
```

5.9 FIT Template

FIT files are created by the mkimage tool using a DTS-syntax input file. This is because the mkimage tool relies on the Device-Tree-Compiler tool as part of the process to create the signed FIT file. The U-Boot version of DTC has the “/incbin/” reserved word that allows an entire binary image to be included as the value of a property in the final DTB. This is how the Linux kernel and Linux DTB are placed into the FIT file.

In the Buildroot base directory, create a file named “linux.its” with the following content:

```
/dts-v1/;
/ {
    description = "Linux kernel image with one or more FDT blobs";
    #address-cells = <1>;
    images {
        kernel@1 {
            data = /incbin/("output/images/zImage");
            type = "kernel";
            arch = "arm";
```

```
            os = "linux";
            compression = "none";
            load = <0x21000000>;
            entry = <0x21000000>;
            hash@1 {
                algo = "sha256";
            };
        };
        fdt@1 {
            data = /incbin/("output/images/at91-sama5d2_xplained.dtb");
            type = "flat_dt";
            arch = "arm";
            compression = "none";
            fdt-version = <1>;
            hash@1 {
                algo = "sha256";
            };
        };
    };
    configurations {
        default = "conf@1";
        conf@1 {
            kernel = "kernel@1";
            fdt = "fdt@1";
            signature@1 {
                algo = "sha256,rsa4096";
                key-name-hint = "samkey";
                sign-images = "fdt", "kernel";
            };
        };
    };
};
```

5.10 Public Key Extraction

The public key needs to be stored into the U-Boot Control DTB. In order to fit into the U-Boot build process nicely, we will take the extracted public key and place the public key into the source DTS file rather than modify the control DTB directly. This step only has to be performed when a new signing certificate is created. With the public key in the DTS file, whenever a U-Boot build is performed, the correct public key is written to the U-Boot binary. This is true even if a “clean” build is performed.

When a new signing certificate is created, the following process can be performed to get the public key into the control device tree source.

An intermediate file, pubkey.dtb, will be used as a holding place for the public key, and then it will be merged into the appropriate dts file.

1. Create a DTS file that is nothing more than a shell to create an empty DTB file. The file in this example, named “pubkey.dts”, should look like this:

```
/dts-v1/;
/ {
};
```

2. Compile the DTS file to get an empty DTB:

```
$ dtc -O dtb pubkey.dts > pubkey.dtb
```

3. Run the mkimage tool located in the U-Boot build directory to create a FIT image, and also extract the public key. Make sure that the key and certificate for signing are in the same directory. In this case, the directory name is “keys”. Two files should exist in the “keys” directory: samkey.crt and

samkey.key. The file samkey.crt is the certificate, and the file samkey.key is the private key used to sign the FIT file.

```
$ output/build/uboot-2018.03/tools/mkimage -f linux.its -k keys -r -K pubkey.dtb
linux.itb
```

4. Uncompile the DTB file and overwrite pubkey.dts:

```
$ dtc -I dtb pubkey.dtb > pubkey.dts
```

5. Edit the U-Boot DTS source file that is used as the control DTB, and insert pubkey.dts into the structure. In this case we will edit “output/build/uboot-2018.03/arch/arm/dts/at91-sama5d2_xplained.dts. Be sure to just copy the “signature” node.
The top lines of at91-sama5d2_xplained.dts before editing:

```
/dts-v1/;
#include "sama5d2.dtsi"
#include "sama5d2-pinfunc.h"
/ {
    model = "Atmel SAMA5D2 Xplained";
    compatible = "atmel,sama5d2-xplained", "atmel,sama5d2", "atmel,sama5";
    chosen {
        u-boot,dm-pre-reloc;
        stdout-path = &uart1;
    };
    ahb {
        usb1: ohci@00400000 {
            num-ports = <3>;
            atmel,vbus-gpio = <&pioA 42 0>;
            pinctrl-names = "default";
            pinctrl-0 = <&pinctrl_usb_default>;
            status = "okay";
        };
```

The top lines of at91-sama5d2_xplained.dts after including the signature node from pubkey.dts:

```
/dts-v1/;
#include "sama5d2.dtsi"
#include "sama5d2-pinfunc.h"
/ {
    model = "Atmel SAMA5D2 Xplained";
    compatible = "atmel,sama5d2-xplained", "atmel,sama5d2", "atmel,sama5";
    signature {
        key-samkey {
            required = "conf";
            algo = "sha256,rsa4096";
            rsa,r-squared = <0xdbfec24b 0xa5ea2b1a 0x68400641 0x7326b903 0x56304c98
0x506293aa 0xd8f2b88c 0xff588c74 0x7ff0a71b 0xfe7d3a5d 0x22f305d2 0x9de61df7 0x5caf73fb
0x383e5828 0xc6b78b09 0x7f8bea2b 0xae04338d 0xd2aabc16 0xe755d903 0x4edd0d2e 0x81d12585
0x410a695d 0x191a5d10 0xc4215770 0x6b35edfa 0x94b6cca 0xb6d4f75e 0x12b8c3ac 0x38fa2bc0
0xeb198a6c 0x5063d6ce 0xdda0ad2b 0xfd31ab15 0xb5a2de0b 0xb9e9f906 0x84d7ab9d 0xe3d481f3
0xd0d66125 0x4f4500b6 0x35a90317 0xc27509f1 0x950266e1 0xecaad051 0x4ef9e854 0x4bafc3ad
0x977d78e4 0xdb28e6f9 0x6e3cd28d 0x8626bb81 0x79018bef 0xf952fcc3 0x143d6e9f 0x472cc55f
0xbd793e29 0xd1973294 0x30b01413 0xa06a8d5b 0x59b87ecb 0x36fe25df 0x8c0240ee 0x9683c6e2
0xac4fab5b 0xa0a41089 0x77a89400 0x563c9cdb 0xf18e1984 0xbaf14849 0xfc0ee81a 0x63e90647
0x608656a3 0xf4d8a777 0xc9c0d23b 0x180c4dd6 0xefba6909 0x5dbd1bfd 0x6b1e1f50 0x8b4a764d
0x99495e21 0x7db3b75e 0x948b4ca8 0xf37dda85 0x352ca2f8 0x43ac5883 0xc0e745f0 0x6fc585b0
0x60e6f9ec 0x49457b4f 0x251948cf 0x48aef777 0xd2f8d3b3 0x6c880548 0xb147d8c3 0x353589be
0x3030a4fa 0x3a0de758 0xdc3c58aa 0x463f4d12 0x3c858599 0xcf0a8e24 0x1d47a6bc 0xc456ebde
0x8f774b0c 0xc79324e7 0xb329cdca 0xd72bf02a 0xdc992be8 0xf6f75455 0x8d7add99 0x85cb9ef5
0xd026d74 0x81da0447 0x1ad13f10 0xaafb6612 0xd4f288ea 0xdb95ff5 0xe3a4786f 0xa746913a
0x8a64781d 0x4f300d3b 0xccf8dd37 0xdb7dc010 0x54098c99 0x3f1e5ea2 0xffc98e8f 0x833f7267
0x740f3d0 0x9b981910 0xd8d558b5>;
            rsa,modulus = <0xe4d61420 0xe402426a 0xb7e8a8e 0x7fed6cf3 0x2d105d06 0x58070696
0x9d12b3d3 0x1a549c06 0xf863d237 0x4ed637b3 0xd1e9172c 0xe9a08063 0x8c69eeda 0x2c31e009
0x75651b8e 0x39773276 0x4ae7d620 0x26ad6d5c 0x94f0368a 0x877fb943 0x8ac20c66 0x527ed3d
0xc1a3d0 0xf86bda13 0xaeb60f4b 0xbfbfd887 0x3fd6c20a 0x3212150a 0x68537bd 0xaf00b12e
0x39941c70 0x359d2602 0xbb2ca80e 0xd3be6c23 0x2c6929cc 0xf26f0249 0x7c93878d 0x435eebf5
0xc76479b1 0x960228cd 0x69d9a190 0x8de8d691 0xe93bebe3 0x708c5dfe 0x2d2013d6 0xfdb2c2f9
0xc8f46fb6 0x27ecd502 0x62737dc5 0xb2797bfd 0x8415de8 0xc17d85b7 0xd4e94000 0xada5f467
0x9b16c556 0x29e74ade 0x34b62261 0x61532487 0x3f7563b8 0x1c8bb911 0x6c92201a 0xb3892cd7
```

```
0x82fd1935 0x76cb4571 0x96edc77f 0x12e0cd53 0xacf61aa9 0x8192b68e 0x722e4231 0xdfb6fb75
0x581502eb 0xdaff4c38 0x2109510 0x9beb7e5e 0x7b639e13 0xc3caa133 0xc4bc8884 0x7535ca33
0x3276a627 0x943cdbc7 0xeb3f2ca0 0x39005a10 0x33323f82 0xfe16ea62 0x414286 0xc26e39e8
0xf72f18f 0x78b1c4ac 0x5e8230f 0xa785d73b 0xf60ba5d1 0x416fa25c 0x6395387c 0xa836b42
0xb9649d5c 0x92b4e 0x871a656c 0x5e970991 0x545b3df7 0xd19fb193 0x84b55c49 0x388111a5
0xbd027fe 0xcb39dd8a 0xd4f5856 0xd3434afb 0xc74df655 0xeec9d781 0x6d7c28f6 0xde2f5d14
0xb266c3b 0xb9f322cb 0x24e4de6c 0x4952433b 0xf1011ad7 0xda4c2c38 0x8a92543f 0xf4db6ab7
0xe4bd1c5b 0x6c2bcc2f 0xd4e59af1 0x988456dd 0xd1176b8f 0x7da860fa 0xdf1b7530 0x240ef9f
0x40e63e 0x6f1fb023>;
            rsa,exponent = <0x0 0x10001>;
            rsa,n0-inverse = <0x55718075>;
            rsa,num-bits = <0x1000>;
            key-name-hint = "samkey";
        };
    };
```

When U-Boot is re-built, this device tree will be used as the control DTB and is appended to the U-Boot executable to create a single binary file.

For a rebuild of U-Boot, use the following command:

```
$ make uboot-rebuild
$ make
```

5.11 Test the New Image

Copy the image, output/images/sdcard.img, to an SDCard, then copy the file “linux.itb” to the FAT partition of the SDCard.

Boot into U-Boot’s command shell by pressing enter within 3 seconds after pressing the Reset button of the SAMA5D2 board. Use the “fatload” command (shown below) to load the “linux.itb” FIT file into SDRAM. Then use the “bootm” command (shown below) to perform the signature check and boot the FIT image. Notice that the “Verifying Hash Integrity” message shows “rsa4096:samkey+”. This indicates that the FIT signature was indeed checked, and found to be valid.

```
Hit any key to stop autoboot:  0
=> fatload mmc 1:1 0x20000000 linux.itb
4975090 bytes read in 307 ms (15.5 MiB/s)
=> bootm 0x20000000
## Loading kernel from FIT Image at 20000000 ...
   Using 'conf@1' configuration
   Verifying Hash Integrity ... sha256,rsa4096:samkey+ OK
   Trying 'kernel@1' kernel subimage
     Description:  unavailable
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0x200000c4
     Data Size:    4937736 Bytes = 4.7 MiB
     Architecture: ARM
     OS:           Linux
     Load Address: 0x21000000
     Entry Point:  0x21000000
     Hash algo:    sha256
     Hash value:   aaf680f3fbfc09d0a102a75959bce1c423f8b096b125ae54ca3f98c08b8caa1e
   Verifying Hash Integrity ... sha256+ OK
## Loading fdt from FIT Image at 20000000 ...
   Using 'conf@1' configuration
   Trying 'fdt@1' fdt subimage
     Description:  unavailable
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x204b59a4
     Data Size:    33480 Bytes = 32.7 KiB
     Architecture: ARM
     Hash algo:    sha256
     Hash value:   4e338db79f27d05fc666aaffce73c5e03562562da7b59913eec2b374aaef4b07
   Verifying Hash Integrity ... sha256+ OK
   Booting using the fdt blob at 0x204b59a4
```



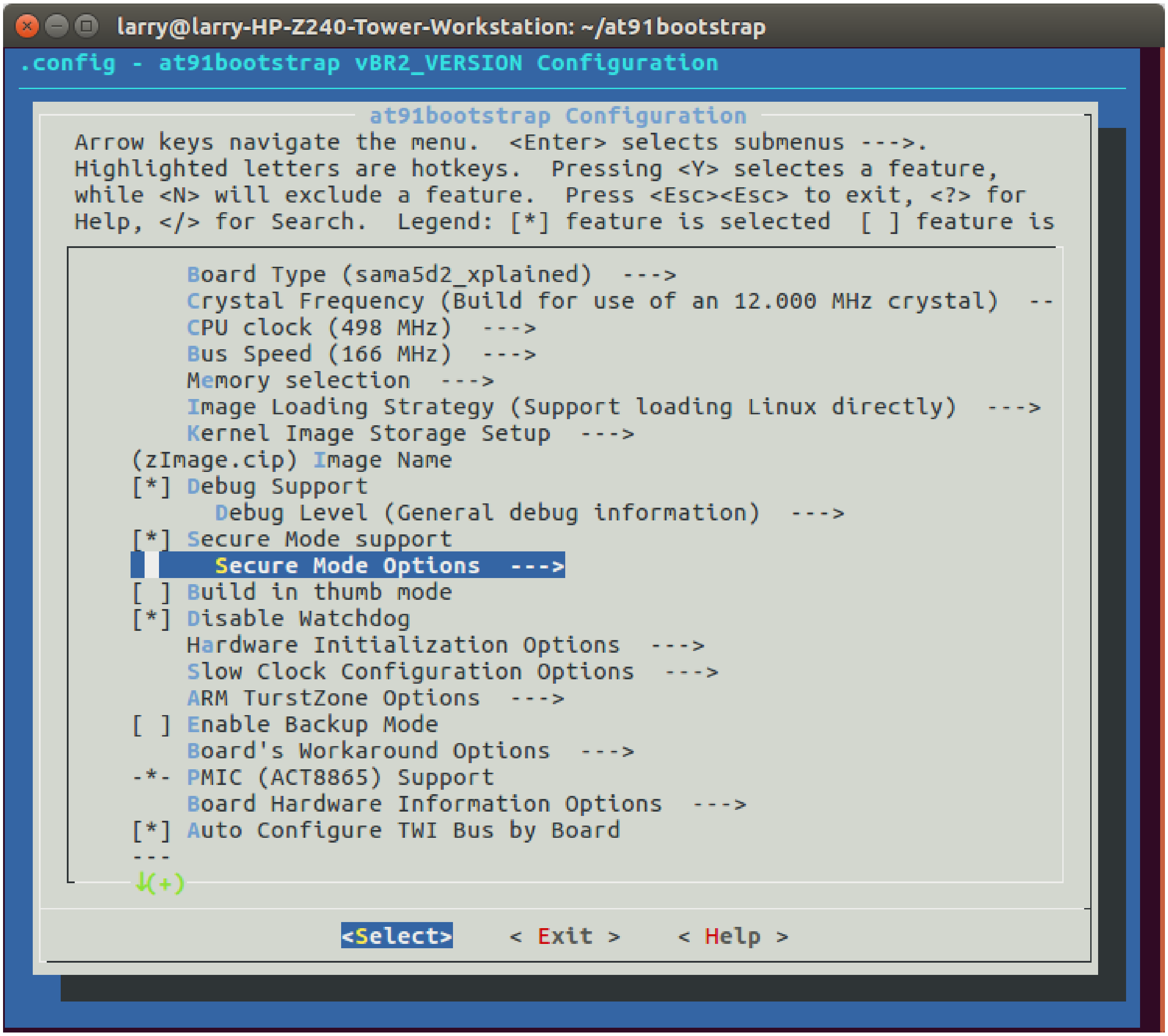
```
   Loading Kernel Image ... OK
   Loading Device Tree to 3f95d000, end 3f9682c7 ... OK
Starting kernel ...
```

6. AT91bootstrap Configuration

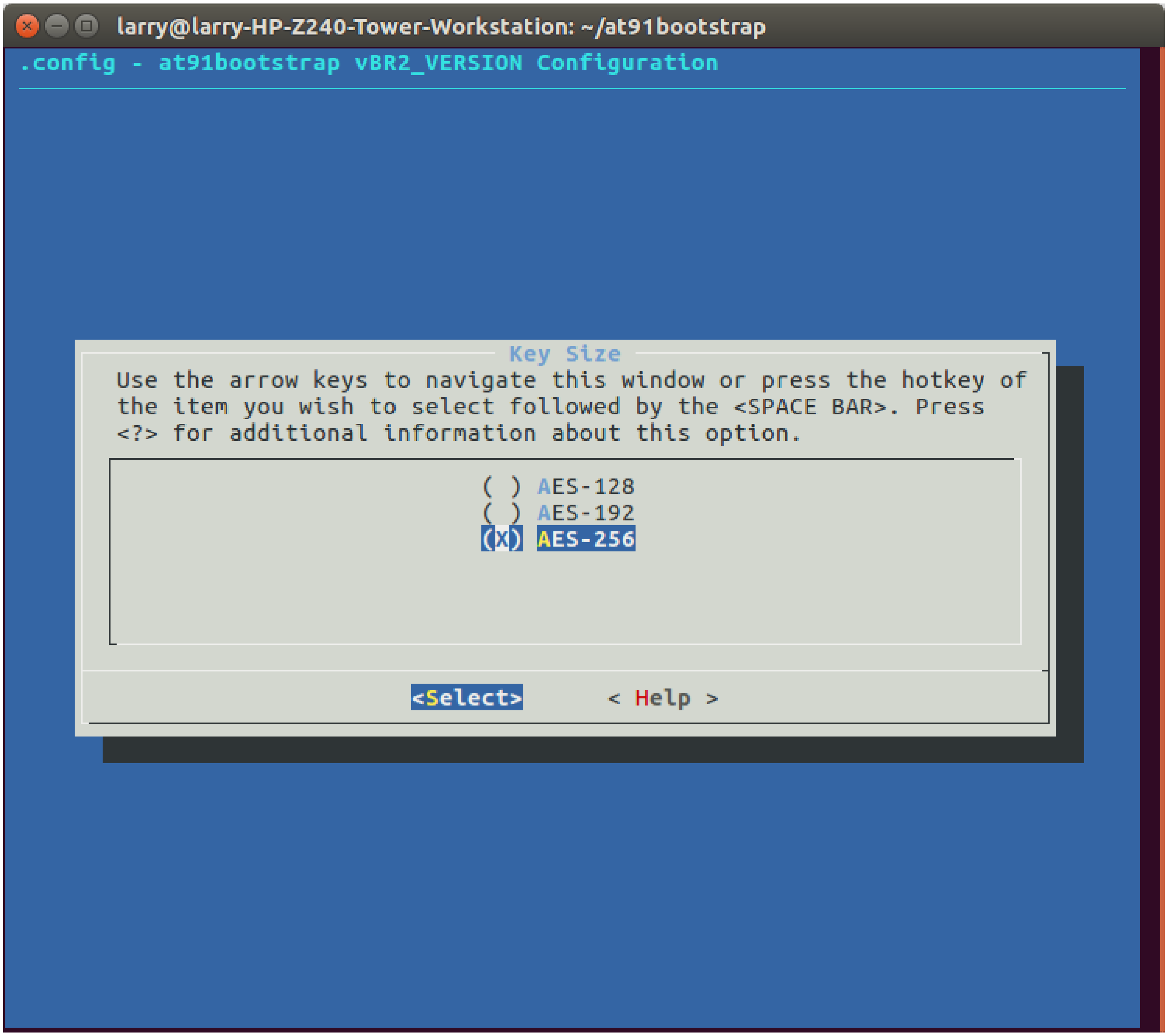
Now that U-Boot has been made to boot only signed images, the AT91bootstrap program must be configured for secure boot. To configure the AT91bootstrap program, run the commands:

```
$ make at91bootstrap3-menuconfig
```

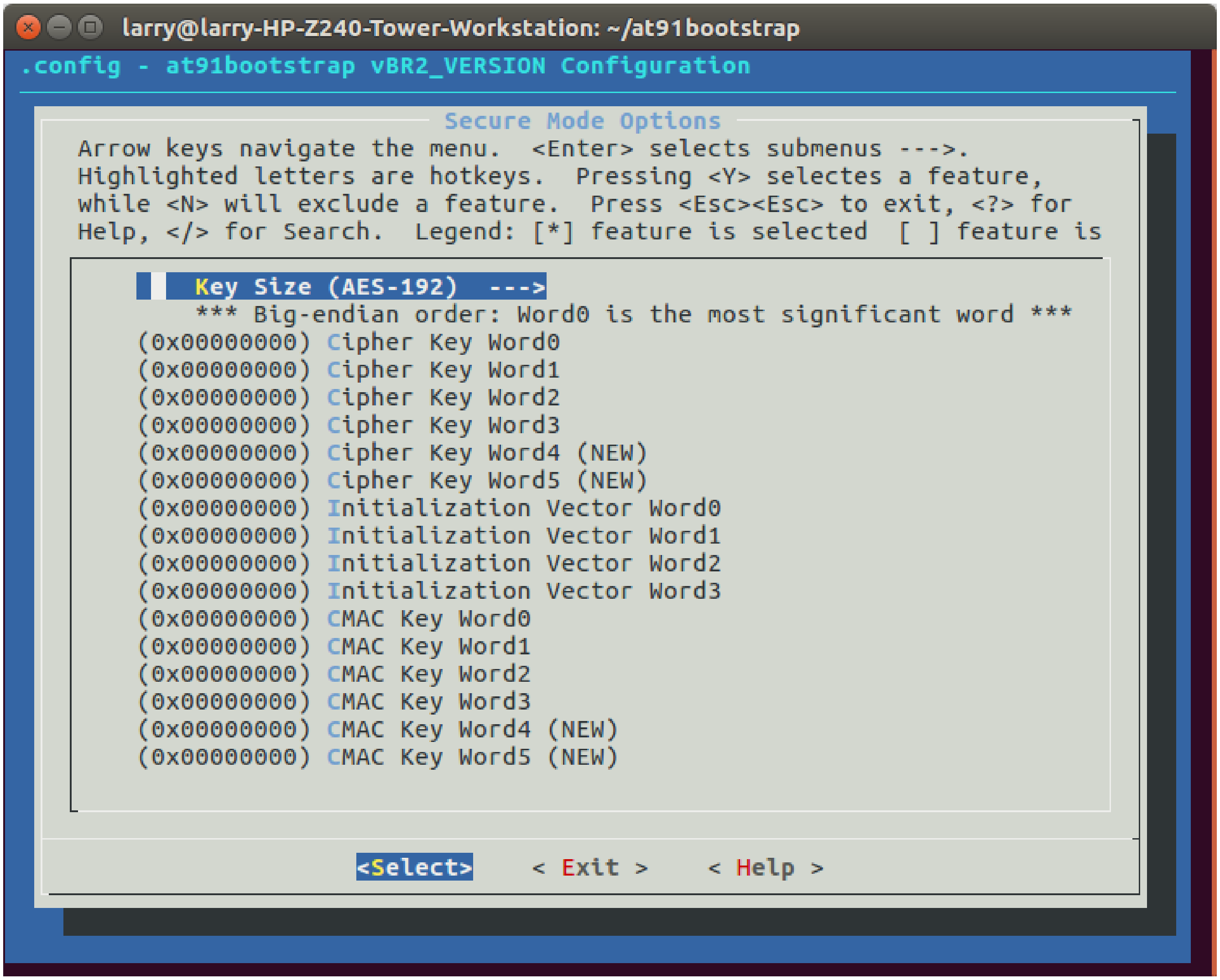
Enable “Secure Mode support”, and be sure to choose a key size that matches the boot configuration that will be used in the system. For all key sizes, the initialization vector remains constant (128 bits, 4 entries).



Key size options for encryption and authentication are shown in the following picture.

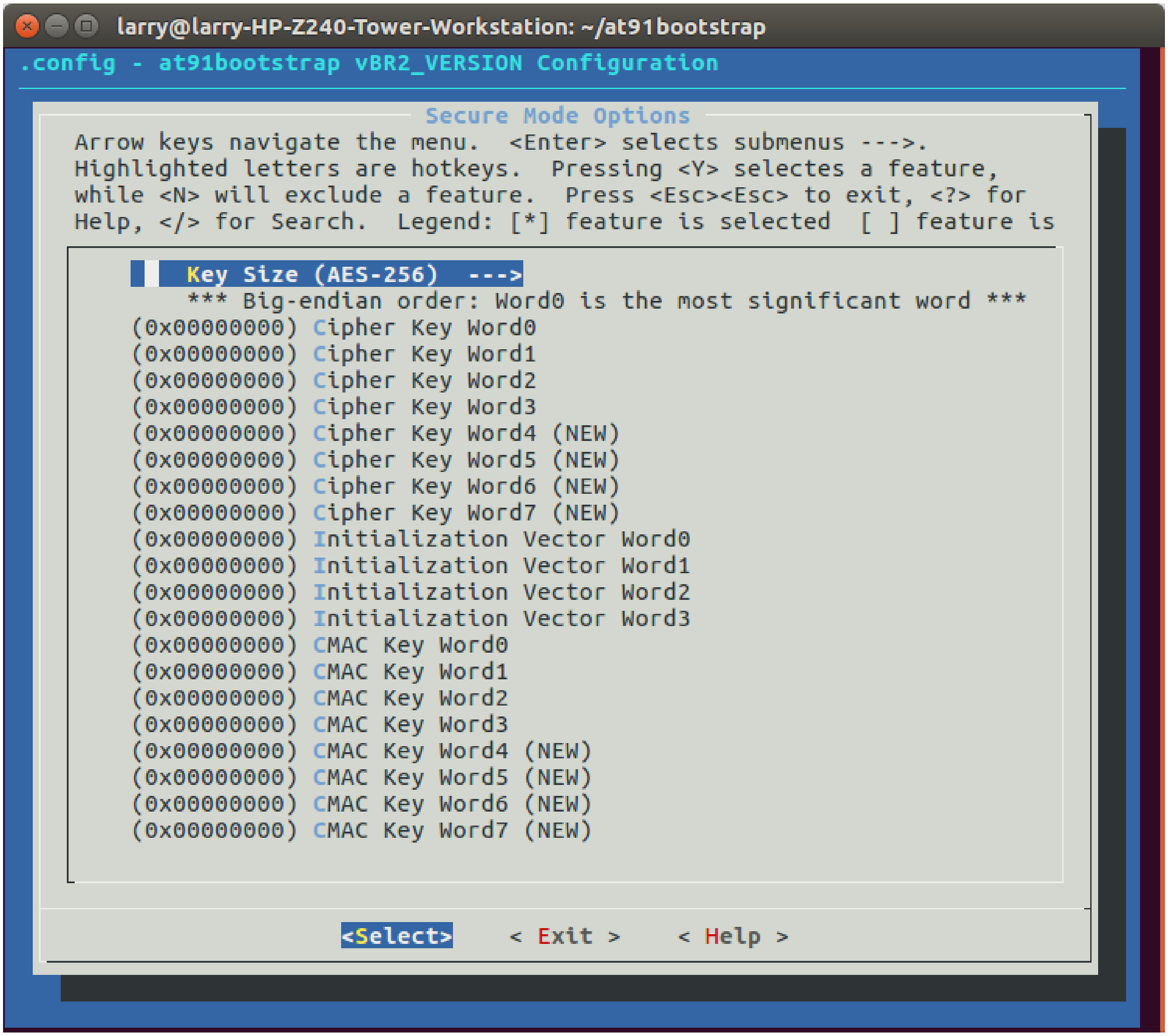


For 128-bit keys, the cipher key as well as the CMAC key are both 128 bits.



When 192-bit keys are selected, both the cipher key and CMAC key have an additional 64 bits to be entered.

When 256-bit keys are selected, both the cipher key and CMAC key are 8 words long.



Make sure that the Linux image to be booted is named zImage.cip in the AT91bootstrap configuration, in order to match the name that will be used for the application encryption step.

6.1 Building AT91bootstrap

After configuring AT91bootstrap, run ‘make’ to build the boot.bin file:

```
$ make
```

This will make the unencrypted boot file that can boot an encrypted U-Boot. It is a good idea to test the bootloader before putting the SAMA5 into Secure Boot mode.

6.2 Secure SAM-BA Tools

The tools necessary to encrypt and sign applications are available from Microchip under NDA.

Contact your local Microchip sales office for more information.

The secure SAM-BA tools consist of two parts.

1. Secure sam-ba cipher – utility to perform various operations during development and preparation of secure boot keys, AT91bootstrap program, and application programs
2. Secure sam-ba loader – utility to communicate with end platforms to configure the device, load keys, hashes, AT91bootstrap programs, and applications

6.3 License Request

Before running the secure SAM-BA tools to perform cryptographic operations, the host machine that will perform the signing must have a valid license file. The license file is restricted to a single host and it is checked every time secure-sam-ba-cipher is run.

Secure SAM-BA cipher is used to generate the license request, and must be run from the host that will be used to perform secure SAM-BA cipher operations such as signing bootstrap programs and applications.

This license request must be validated by Microchip and a license file is then sent back to the requester.

Example:

The following command line will generate a license request in file request.txt.

```
secure-sam-ba-cipher.py request-license -o request.txt
```

6.4 Encrypting/Signing U-Boot

In addition to a valid license file, the secure SAM-BA “application” command requires an application key. Unlike the single license file that is used on the signing machine, there is no limit to the number of application keys that can be used. It is important to note that the application key sizes and values must exactly match the sizes and values configured in the AT91bootstrap program.

6.5 Application Key File Format

The file app.key must contain application keys in hexadecimal using the following format:

```
KEY=0000000000000000000000000000000000000000000000000000000000000000
IV_KEY=00000000000000000000000000000000
CMAC_KEY=0000000000000000000000000000000000000000000000000000000000000000
```

The length of the AES key and AES CMAC key can be:

- 128 bits (16 bytes, 32 hexadecimal digits)
- 192 bits (24 bytes, 48 hexadecimal digits)
- 256 bits (32 bytes, 64 hexadecimal digits)

AES IV is always 128 bits (16 bytes, 32 hexadecimal digits).

6.6 Application Encrypt/Sign Example

In this example, the AT91bootstrap program is configured to securely boot a U-Boot image named “uboot.cip” using 128-bit AES. The AT91bootstrap program must be configured with the correct key size option and make sure that the key values in the AT91bootstrap configuration file match the key file used by the secure SAM-BA "application" command. This example assumes that the secure SAM-BA license file has already been requested, and a valid license file received back from Microchip.

6.7 Application Key File

The key file in this example is named zImage.key and is formatted to use 128-bit keys.

```
$ cat uboot.key
KEY=1234567811111111abcdef01ffffffff
IV_KEY=00000001000000020000000300000004
CMAC_KEY=876543211111111110fedcbaffffffff
```

6.8 AT91bootstrap .config file

The .config file for AT91bootstrap is configured with values that match the uboot.key file.

```
CONFIG_SECURE=y
#
# Secure Mode Options
#
CONFIG_AES_KEY_SIZE_128=y
# CONFIG_AES_KEY_SIZE_192 is not set
# CONFIG_AES_KEY_SIZE_256 is not set
#
# Big-endian order: Word0 is the most significant word
#
CONFIG_AES_CIPHER_KEY_WORD0=0x12345678
CONFIG_AES_CIPHER_KEY_WORD1=0x11111111
CONFIG_AES_CIPHER_KEY_WORD2=0xabcdef01
CONFIG_AES_CIPHER_KEY_WORD3=0xffffffff
CONFIG_AES_IV_WORD0=0x00000001
CONFIG_AES_IV_WORD1=0x00000002
CONFIG_AES_IV_WORD2=0x00000003
CONFIG_AES_IV_WORD3=0x00000004
CONFIG_AES_CMAC_KEY_WORD0=0x87654321
CONFIG_AES_CMAC_KEY_WORD1=0x11111111
CONFIG_AES_CMAC_KEY_WORD2=0x10fedcba
CONFIG_AES_CMAC_KEY_WORD3=0xffffffff
```

6.9 Run secure-sam-ba-cipher "application" Command

The "application" command takes the application key file and the application as input, and creates an encrypted and signed output file.

```
$ secure-sam-ba-cipher.py application -l license_D2.txt -k uboot.key -i uboot.bin -o uboot.cip
```

6.10 Test the Application

Replace the AT91bootstrap program on the SDCard with the new AT91bootstrap program that is configured to boot a secure U-Boot image.

Copy the following files to the FAT partition of the SDCard:

- boot.bin – the new AT91bootstrap program
- uboot.cip – the new encrypted and signed U-Boot image that matches the configuration in boot.bin

Place the SDCard into the SAMA5 development board and make sure the image boots.

6.10.1 Securing the AT91bootstrap program

After the AT91bootstrap program is correctly booting the uboot.cip file, a secure AT91bootstrap binary named boot.cip can be created from this boot.bin. Similar to the way that the uboot.cip file needs an Application Key, the AT91bootstrap program needs a key that will be used by the ROM code to securely authenticate and decrypt the AT91bootstrap image. The key that the ROM code uses to decrypt and authenticate the AT91bootstrap program is derived from a key called “customer key”.

6.11 AT91bootstrap Key File Format

The file cust.key must contain a customer key in hexadecimal using the following format:

```
KEY_CUST=0000000000000000000000000000000000000000000000000000
```

The customer key length for the SAMA5D2 is 256 bits.

6.12 Run secure-sam-ba-cipher “customer-key” Command

The “customer-key” command takes the AT91bootstrap key file as input and creates an encrypted and signed output file that will eventually be loaded into the SAMA5 MPU in a separate step.

```
secure-sam-ba-cipher.py customer-key -d sama5d2x -l license_D2.txt -k cust.key -o customer-key.cip
```

6.13 Run secure-sam-ba-cipher “bootstrap” Command

The “bootstrap” command takes the bootstrap key (customer key) file and the bootstrap binary as input, and creates an encrypted and signed output file.

```
$ secure-sam-ba-cipher.py bootstrap -d sama5d2x -l license_D2.txt -k cust.key -i boot.bin -o boot.cip
```

6.14 Test the AT91bootstrap Program

The boot.cip file is the encrypted and signed version of boot.bin that was previously tested in Non Secure mode.

Copy the boot.cip file to the FAT partition of the SDCard.

6.15 Provision the Board using secure-sam-ba-loader

Previously, the secure-sam-ba-cipher program was used to prepare the following encrypted/signed files:

- zImage.cip - encrypted and signed Linux image
- boot.cip – encrypted and signed AT91bootstrap program
- customer-key.cip – encrypted and signed AT91bootstrap key

The zImage.cip file and the boot.cip file are copied to the boot media - in our case the SDCard. The customer-key.cip file is stored in the SAMA5 MPU and should not be stored on the boot media.

The SAMA5 should be placed into Secure mode using the secure-sam-ba-loader utility. Before running the command, make sure the end device is running the SAM-BA Monitor, and make sure a USB cable is connected to the host properly. Note the device name that is enumerated when the USB cable is plugged in. In this case, the console is /dev/ttyACM0, and the SAM-BA interface is /dev/ttyACM1.

After seeing the RomBOOT prompt, use secure-sam-ba-loader to place the MPU into Secure Boot mode:

```
$ secure-sam-ba-loader.py secure-mode -d sama5d2x -p /dev/ttyACM1
```

After running the secure-sam-ba-loader “secure-mode” command, the prompt should be the following after reset:

```
Secure Boot Mode
```

The next step is to load the customer key into the SAMA5 MPU.

```
$ secure-sam-ba-loader.py customer-key -d sama5d2x -p /dev/ttyACM1 -i customer-key.cip
```

6.16 Test the Image

Place the SDCard into the board, and press Reset. The “Secure Boot Mode” prompt should appear on the console followed by the running of AT91bootstrap, and then Linux. At this point, all testing has been performed using one of the MPU's backup registers (BUREG0). If power is removed from the device, all Secure mode settings will be undone.

6.17 Burn Fuses

After the AT91bootstrap program is working correctly in Secure mode, the secure-mode bit can be permanently set in the device fuses with the following command:

```
$ secure-sam-ba-loader.py secure-mode --fuse -d sama5d2x -p /dev/ttyACM1
Connecting to serial port /dev/ttyACM1...
Connected to /dev/ttyACM1.
Enabling secure mode on sama5d2x...
Secure mode successfully enabled, please power cycle the device.
```

The AT91bootstrap key can then be permanently programmed into fuses with the following:

```
$ secure-sam-ba-loader.py customer-key -d sama5d2x -p /dev/ttyACM1 --fuse -i customer-key.cip
```

Now the device is permanently in Secure Boot mode.

Be aware that fuse programming is a permanent operation and cannot be undone.

7. Further Steps

This application note showed the process to follow to securely boot a SAMA5 MPU system. The settings used for encrypting and signing were limited to 128-bit AES with AES CMAC. There are other modes of authentication that can be used, and are further described in the secure boot package that is delivered under NDA.

8. Revision History

8.1 Rev. A - 06/2018

This is the initial released version of this application note.

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com/. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at http://www.microchip.com/. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, AnyRate, AVR, AVR logo, AVR Freaks, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, Heldo, JukeBlox, KeeLoq, Kleer, LANCheck, LINK MD, maXStylus, maXTouch, MediaLB, megaAVR, MOST, MOST logo, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, PIC32 logo, Prochip Designer, QTouch, SAM-BA, SpyNIC, SST, SST Logo, SuperFlash, tinyAVR, UNI/O, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

ClockWorks, The Embedded Control Solutions Company, EtherSynch, Hyper Speed Control, HyperLight Load, IntelliMOS, mTouch, Precision Edge, and Quiet-Wire are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, INICnet, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, memBrain, Mindi, MiWi, motorBench, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

ISBN: 978-1-5224-3268-5

Quality Management System Certified by DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-67-3636 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-7289-7561 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820