DESIGNING SECURE IOT DEVICES STARTS WITH A SECURE BOOT

DONNIE GARCIA, *SOLUTIONS ARCHITECT FOR SECURE TRANSACTIONS*, NXP

DIYA SOUBRA, *SENIOR PRODUCT MARKETING MANAGER*, ARM

PUBLIC

PROPER REARING FOR THE IOT EDGE NODE STARTS WITH A SECURE BOOT

Phishing scams perpetrated by re-purposing IoT end nodes is a real threat. A plan for the development, manufacturing and deployment stages of IoT edge nodes must be made. The complexities of life cycle management create a demanding environment where developers must make use of available resources to create the hardware, software, policies and partnerships used to achieve product goals. An essential component is protecting each device power up with a secure and trusted boot. This can be achieved with the right MCU hardware capabilities and ARM® mbed TLS. This webinar will introduce a life cycle management model and detail the steps for how to achieve a secure boot with NXP's ARM Cortex®-M based MCUs with mbed TLS cryptography support. A special guest from ARM will discuss new processors and architectures with ARM TrustZone® for ARMv8-M that will free time and resources for secure designs.

In this webinar, you will learn how to:

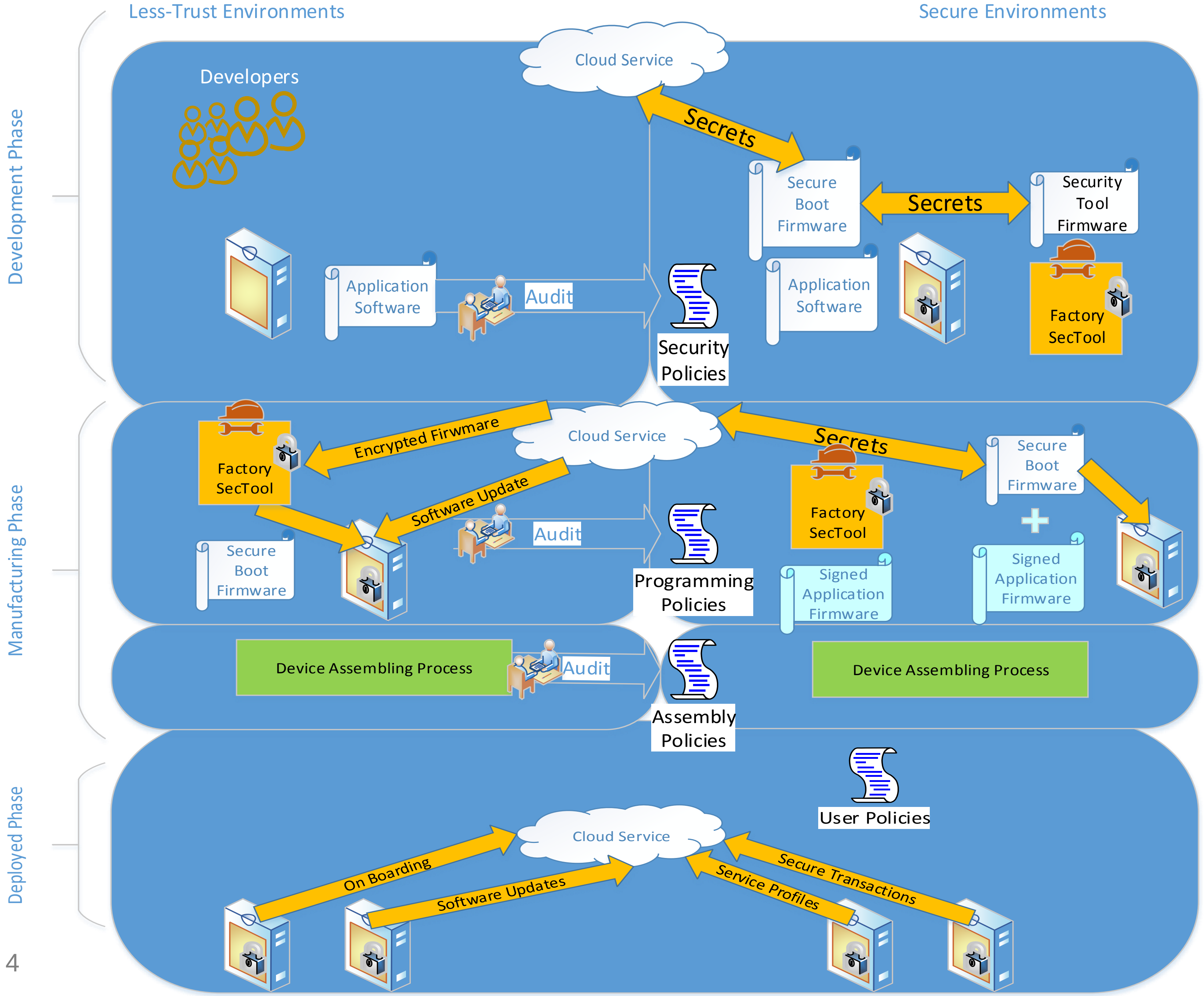
- Manage the life cycle of an IoT edge node from development to deployment
- Leverage hardware and software offerings available with the Kinetis MCU portfolio that can help you protect against attacks
- Ease the burden of secure IoT edge node development using new processors and architectures from ARM

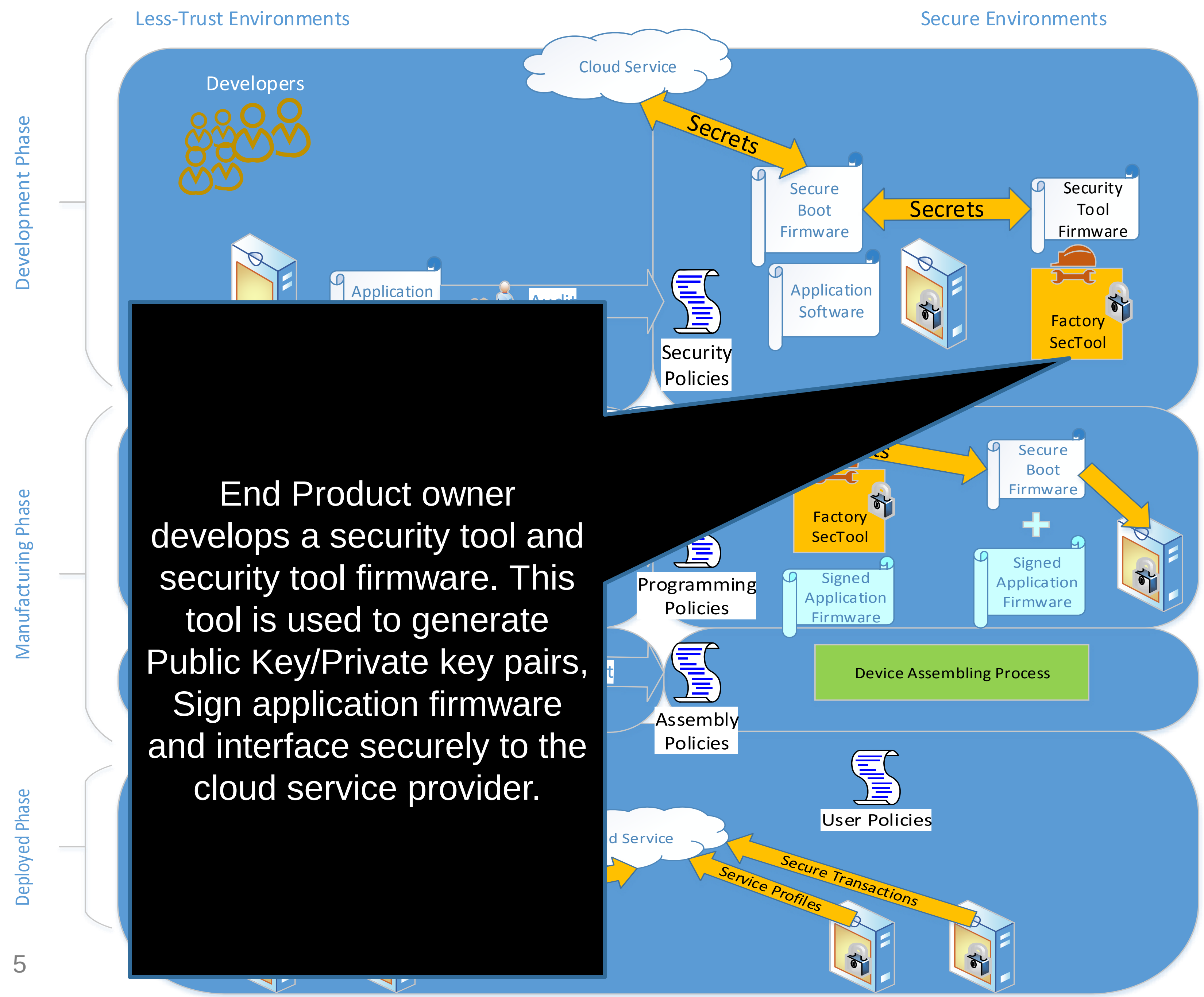
Agenda

- IoT Edge Node Life Cycle Management Model
- Secure Boot Architecture
  - NXP Kinetis MCU solution
    - Kinetis K28F MCU How To:
      - Set Flash Block Protection
      - Set Chip Security Level
    - mbed TLS
      - Adding Relevant Source Code to KBOOT
      - APIs Needed for Key Generation, Signatures and Verification
    - KBOOT Tools
      - Boot Directive file
      - Using ElfToSB
      - Using Blhost
- Portability
  - Moving to Other Targets
- ARMv8-M: What the future will bring
  - New Capabilities to Make Secure Designs Ecosystem and Developer Friendly
  - Improved Developer Productivity and Higher Energy Efficiency

1

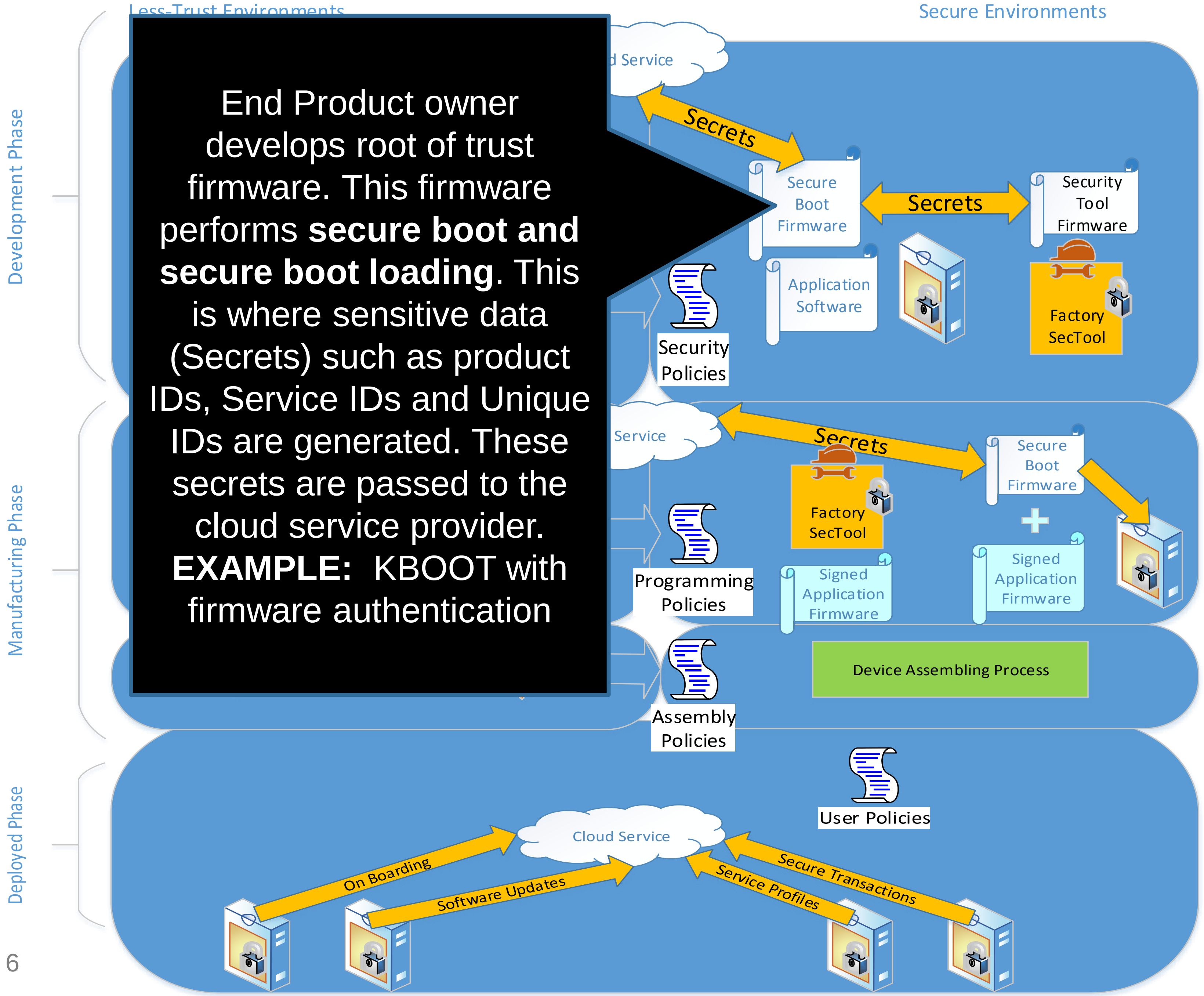
IoT Edge Node Life Cycle Management Model

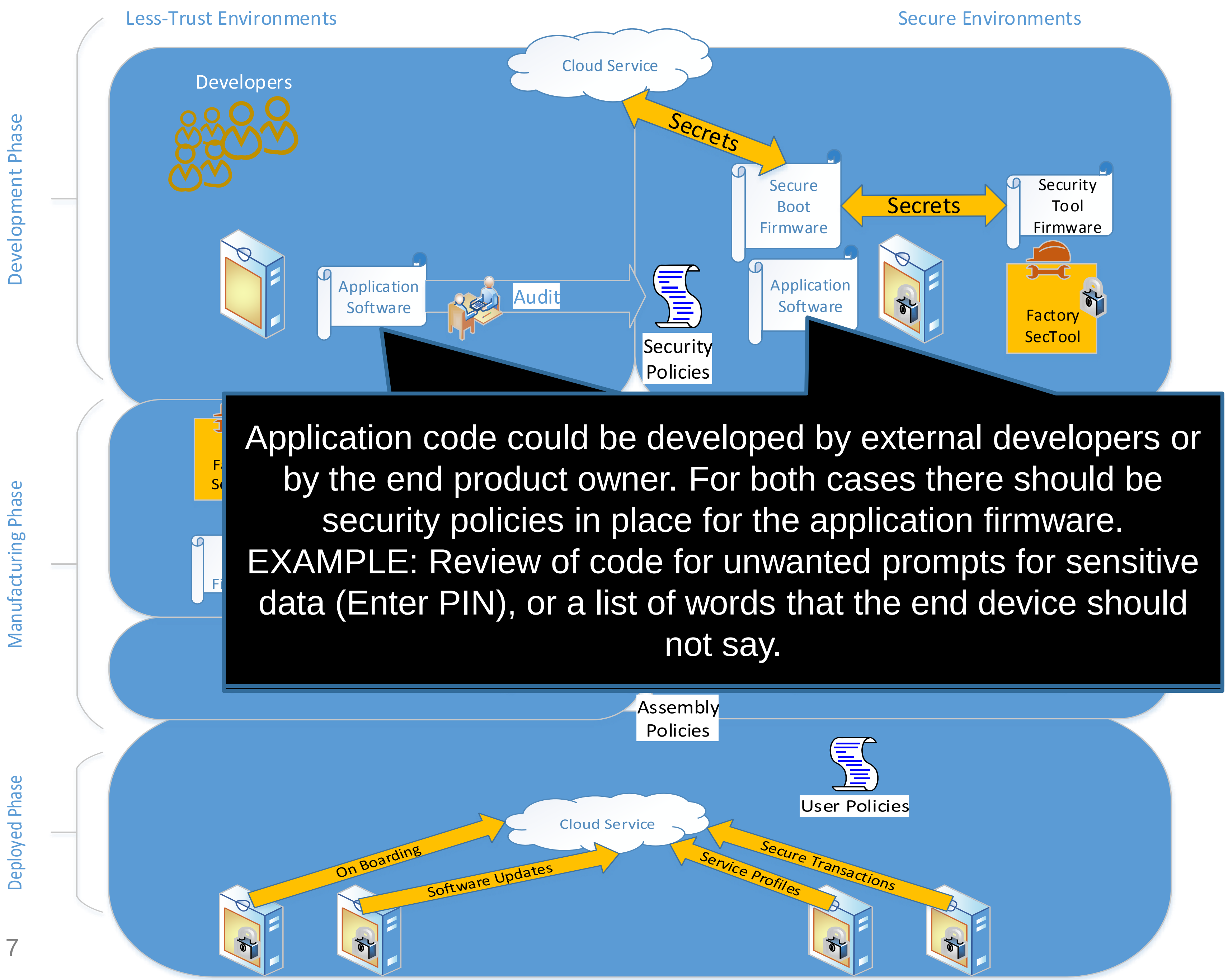

Less-Trust Environments
Secure Environments
Development Phase
Manufacturing Phase
Deployed Phase
Developers
Cloud Service
Secrets
Secure Boot Firmware
Secrets
Security Tool Firmware
Application Software
Audit
Security Policies
Application Software
Factory SecTool
Factory SecTool
Encrypted Firwmare
Cloud Service
Secrets
Secure Boot Firmware
Software Update
Audit
Programming Policies
Factory SecTool
Secure Boot Firmware
Signed Application Firmware
Signed Application Firmware
Device Assembling Process
Audit
Assembly Policies
Device Assembling Process
User Policies
Cloud Service
On Boarding
Software Updates
Service Profiles
Secure Transactions



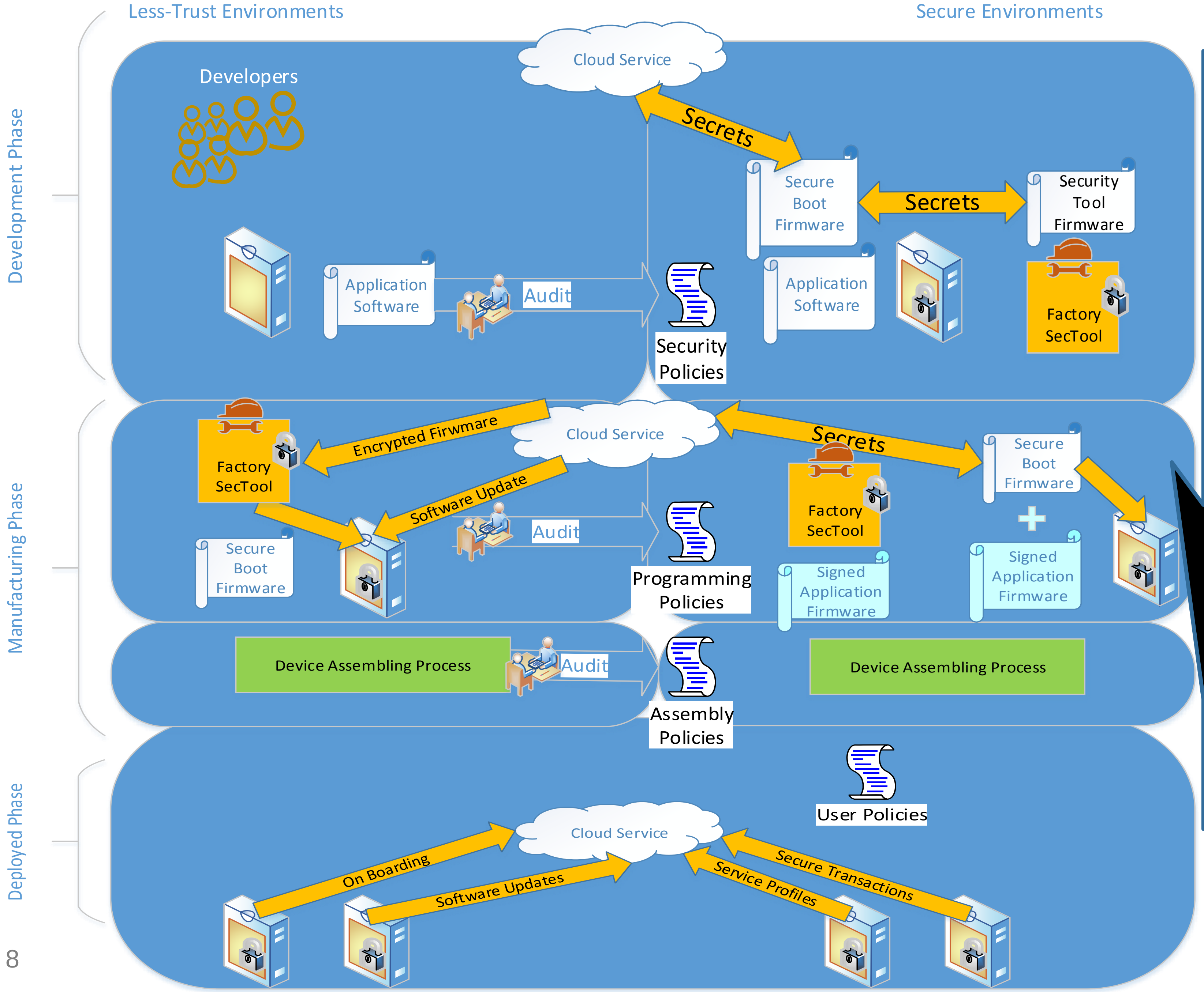
End Product owner develops a security tool and security tool firmware. This tool is used to generate Public Key/Private key pairs, Sign application firmware and interface securely to the cloud service provider.

End Product owner develops root of trust firmware. This firmware performs **secure boot and secure boot loading**. This is where sensitive data (Secrets) such as product IDs, Service IDs and Unique IDs are generated. These secrets are passed to the cloud service provider.
EXAMPLE: KBOOT with firmware authentication



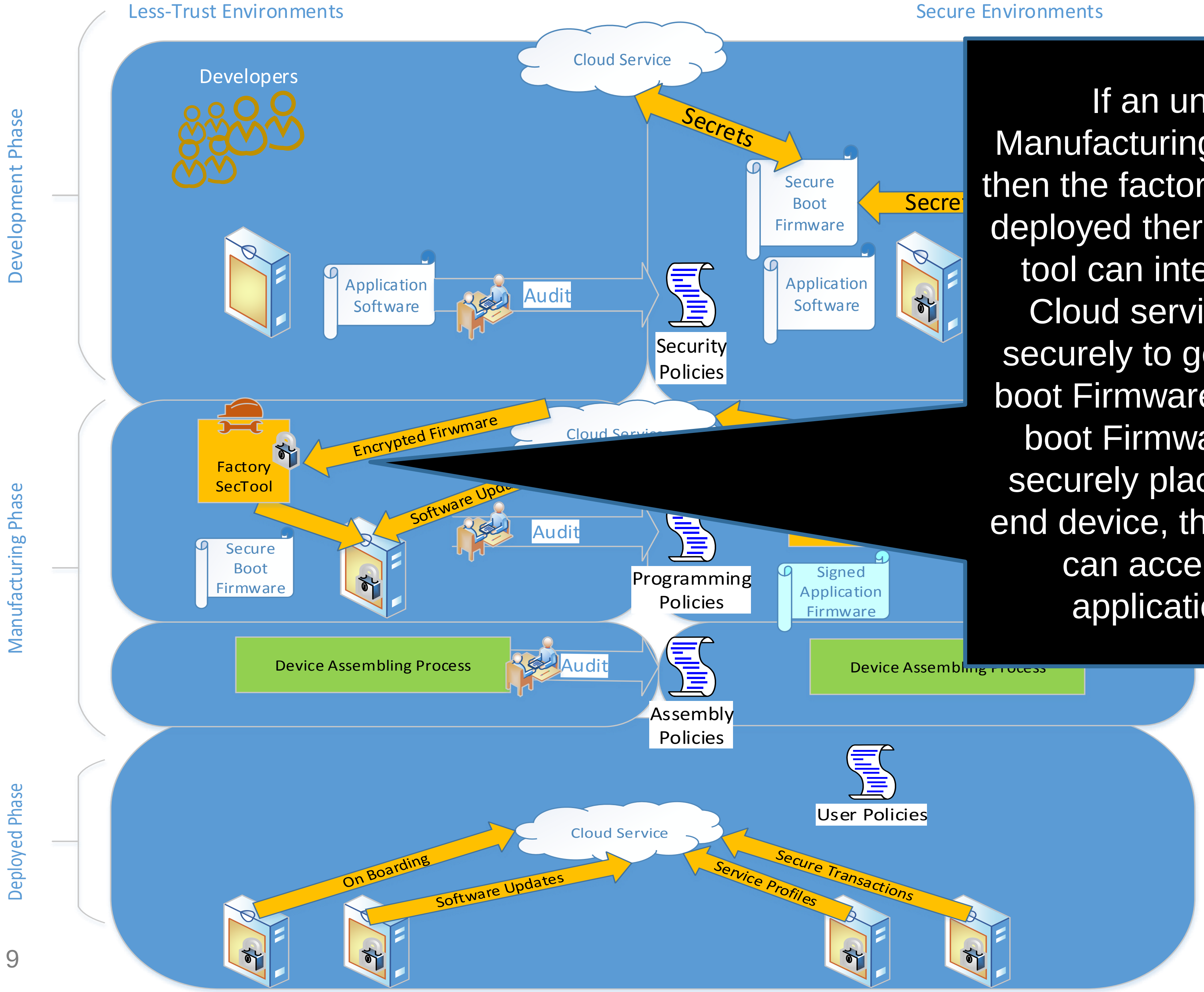


Application code could be developed by external developers or by the end product owner. For both cases there should be security policies in place for the application firmware. EXAMPLE: Review of code for unwanted prompts for sensitive data (Enter PIN), or a list of words that the end device should not say.

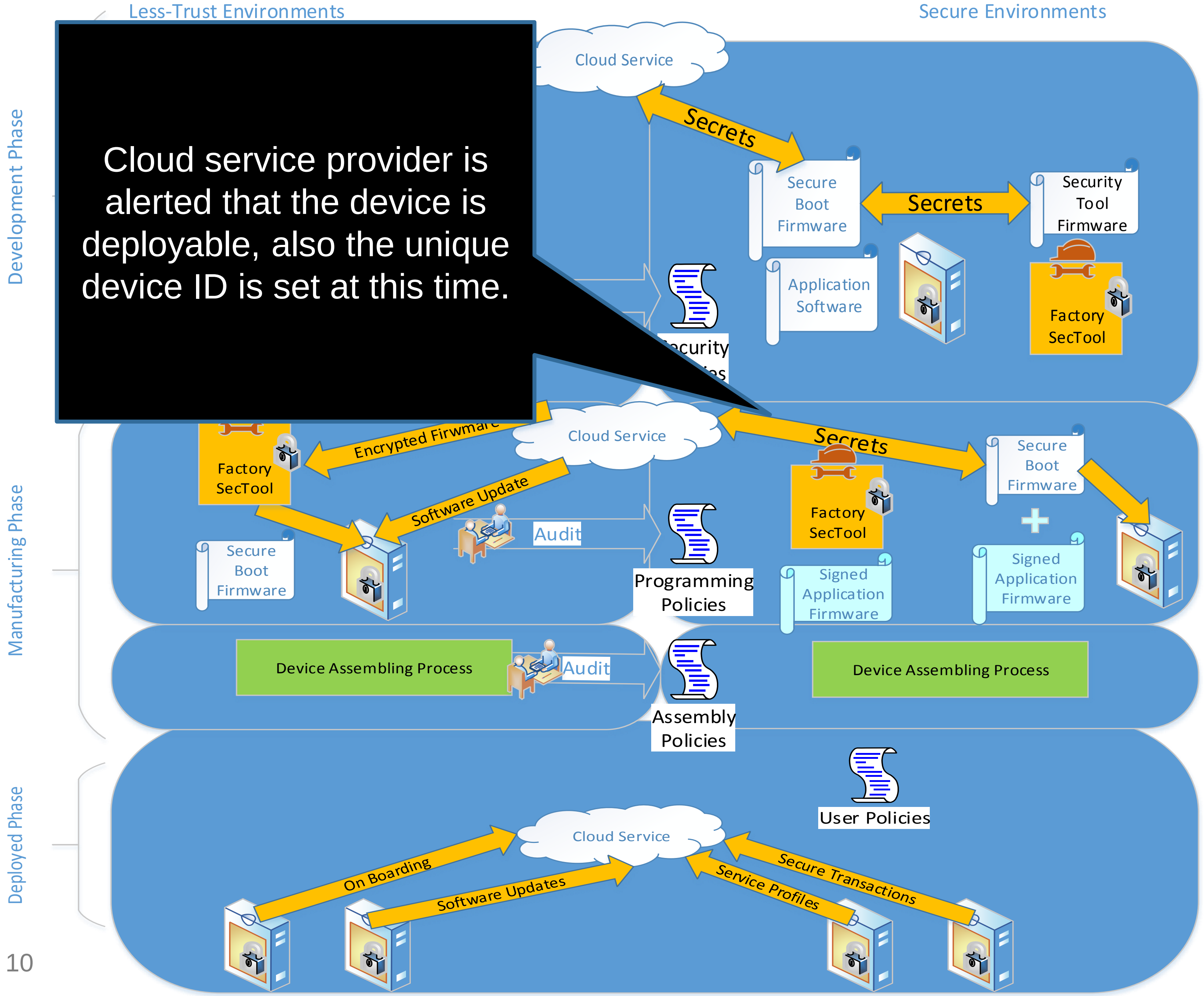


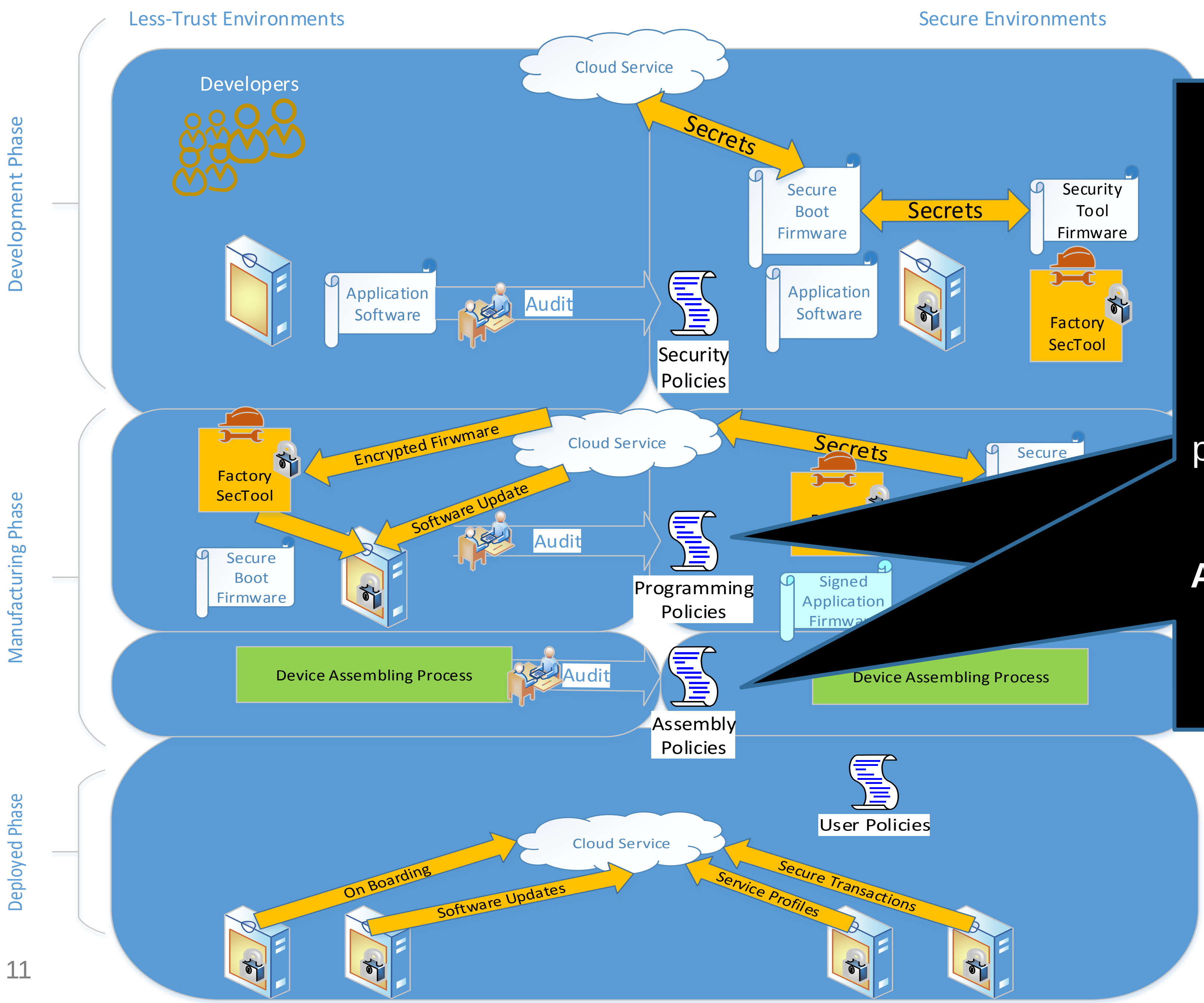
For the case of a controlled manufacturing site, then the factory tool is used to sign application software.

Chip security mechanisms are used to protect the secure boot firmware. EXAMPLE: Kinetis flash block protection, Flash Access control, Chip security.



If an untrusted Manufacturing site is used, then the factory tool must be deployed there. The factory tool can interface to the Cloud service provider securely to get the secure boot Firmware. The secure boot Firmware must be securely placed on to the end device, then the device can accept signed application code.

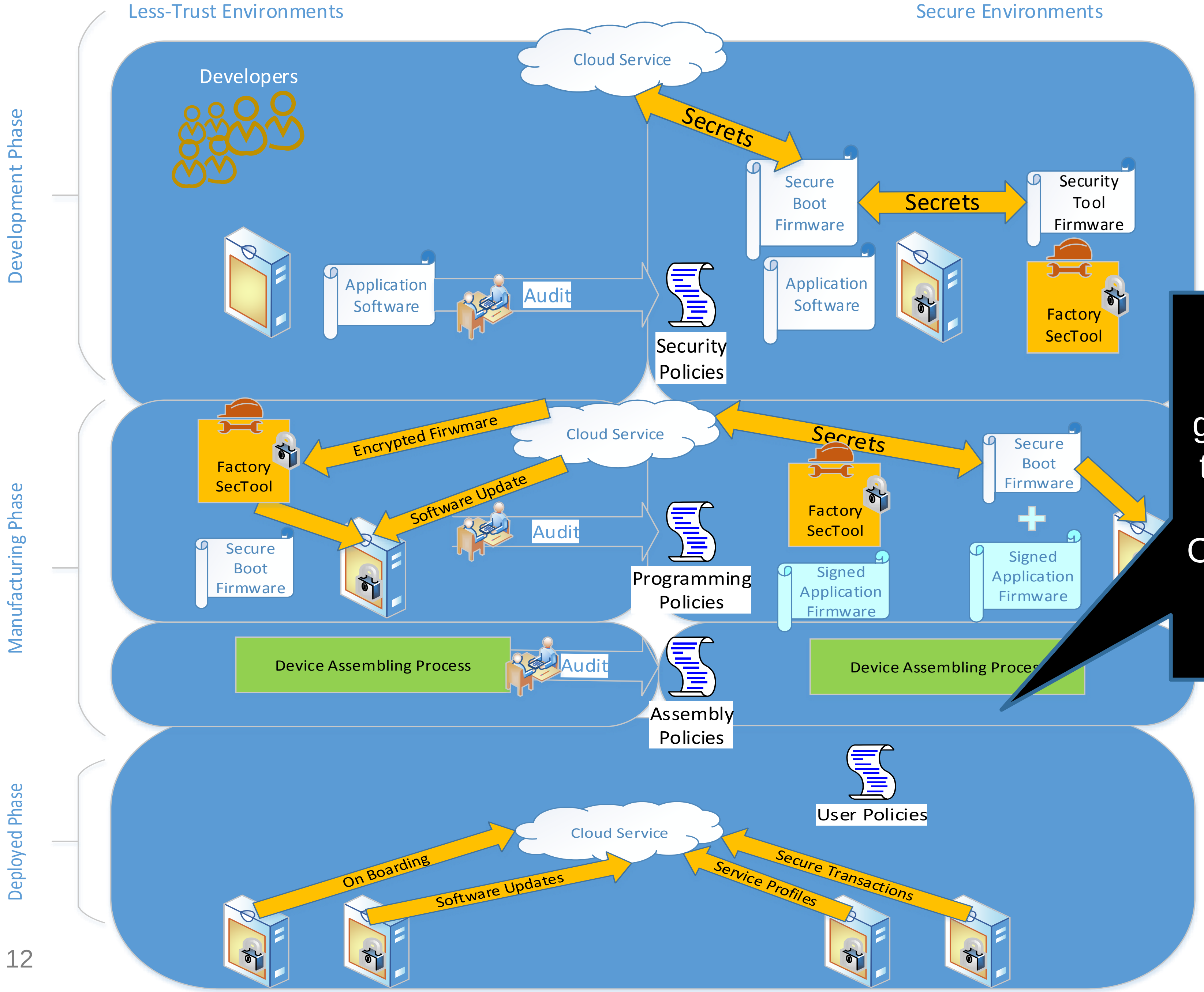

Less-Trust Environments
Secure Environments
Development Phase
Manufacturing Phase
Deployed Phase
Cloud service provider is alerted that the device is deployable, also the unique device ID is set at this time.
Cloud Service
Secrets
Secure Boot Firmware
Secrets
Security Tool Firmware
Application Software
Factory SecTool
Encrypted Firwmare
Cloud Service
Secrets
Secure Boot Firmware
Factory SecTool
Software Update
Audit
Secure Boot Firmware
Programming Policies
Factory SecTool
Signed Application Firmware
Signed Application Firmware
Device Assembling Process
Audit
Assembly Policies
Device Assembling Process
User Policies
Cloud Service
On Boarding
Software Updates
Secure Transactions
Service Profiles



For either a secure manufacturing site or a less trust environment:

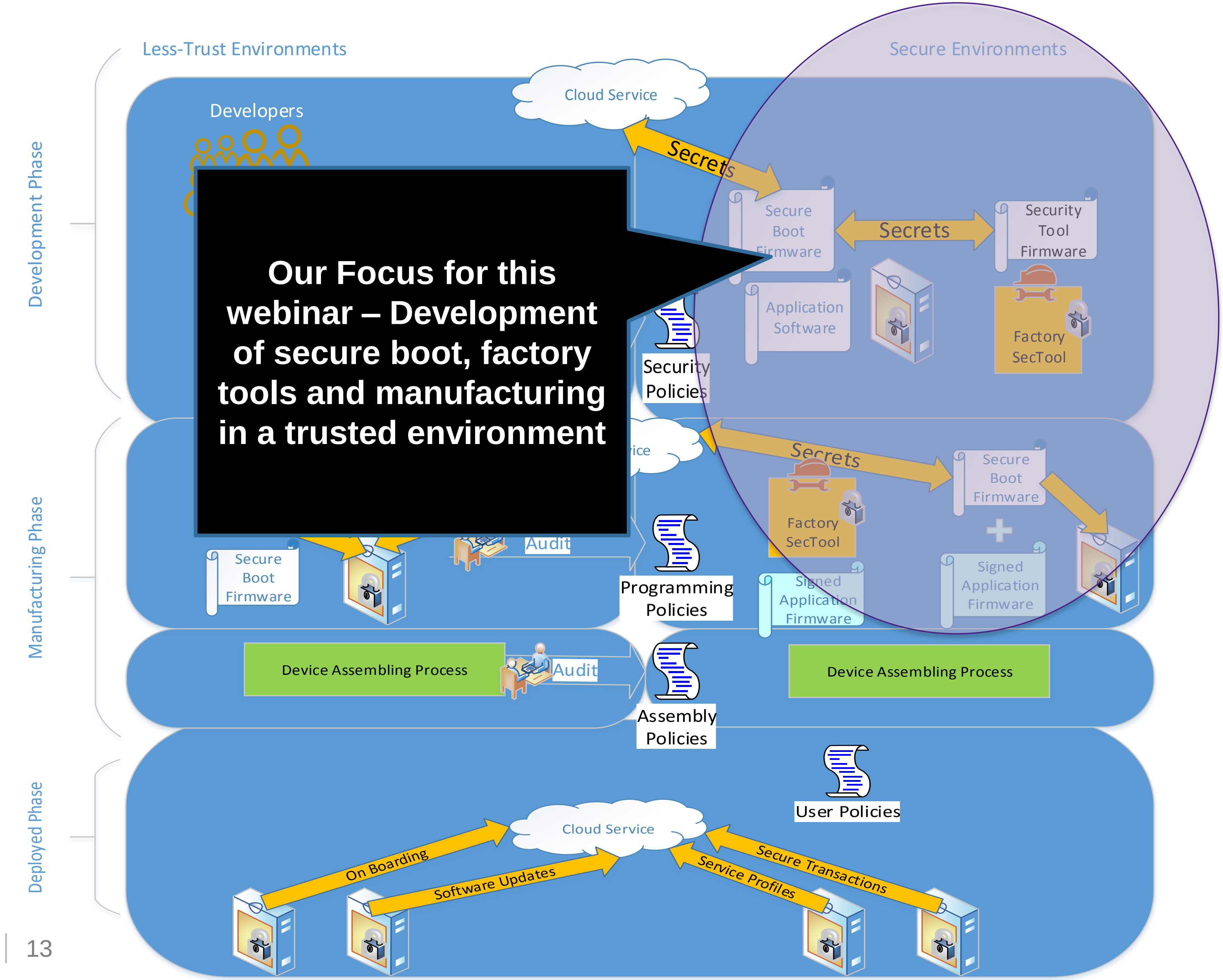
Programming policies ensure that the proper steps are taken and controls are in place to protect the programming of the end device.

Assembly Policies ensure that only approved components are used



User policies provide guidelines for the end user to maintain the security of the device. EXAMPLE: Check for pin pad overlays, or skimmers.

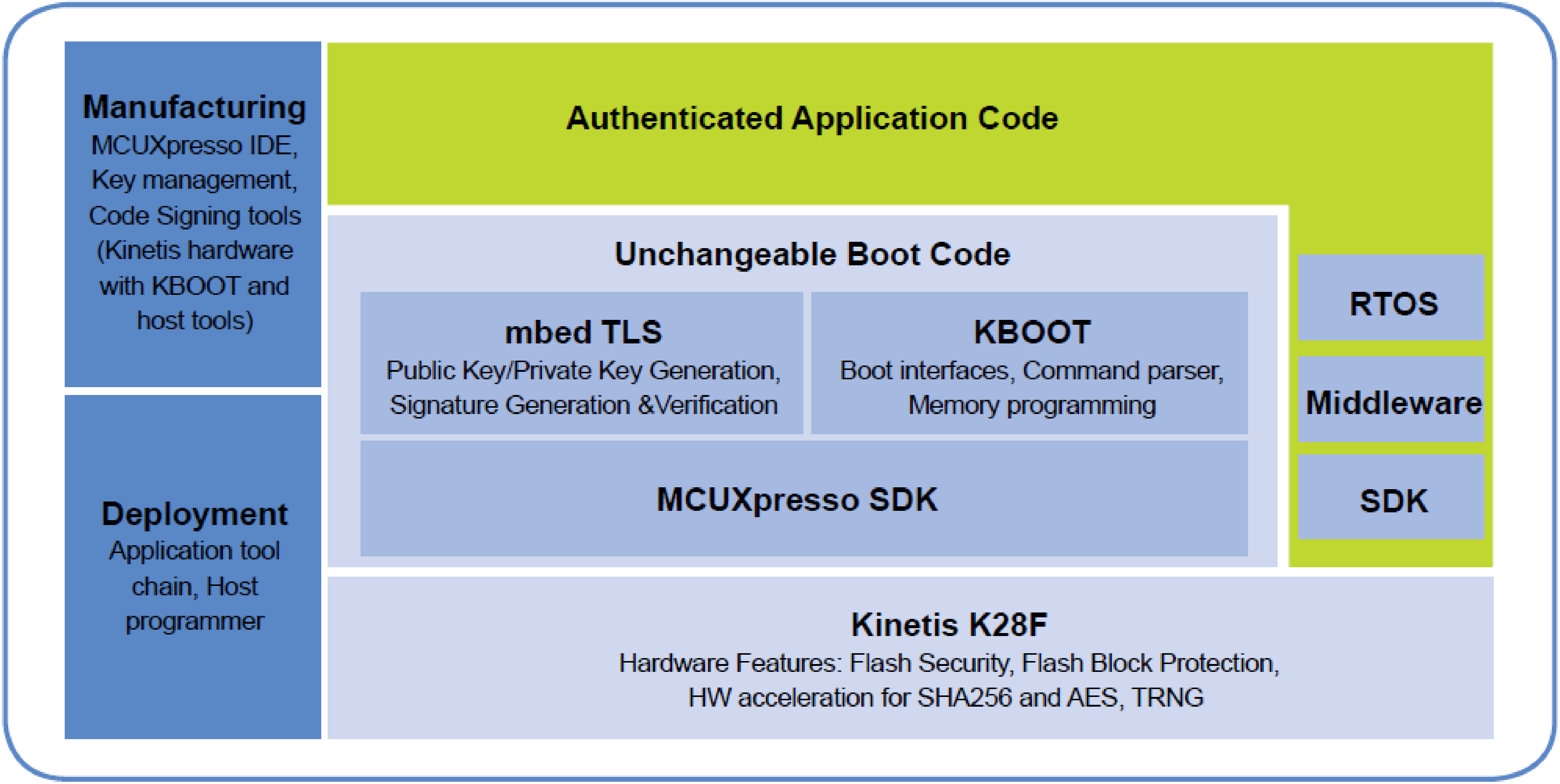
NXP | ARM


Less-Trust Environments
Secure Environments
Development Phase
Manufacturing Phase
Deployed Phase
Cloud Service
Developers
Secrets
Secure Boot Firmware
Secrets
Security Tool Firmware
Application Software
Factory SecTool
Security Policies
Our Focus for this webinar – Development of secure boot, factory tools and manufacturing in a trusted environment
Secrets
Secure Boot Firmware
Factory SecTool
Signed Application Firmware
Signed Application Firmware
Audit
Secure Boot Firmware
Programming Policies
Device Assembling Process
Audit
Assembly Policies
Device Assembling Process
User Policies
Cloud Service
On Boarding
Software Updates
Service Profiles
Secure Transactions

2

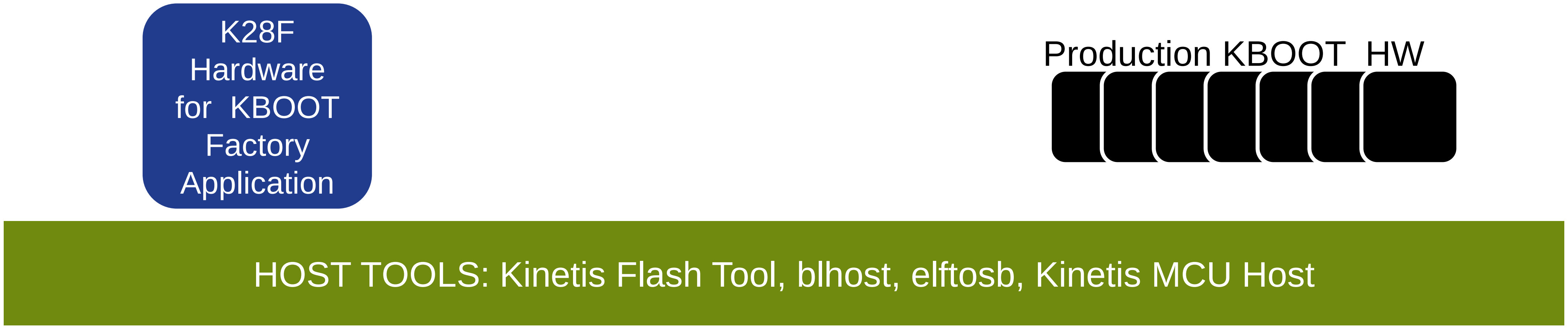
Secure Boot Architecture

System Architecture for Secure Boot

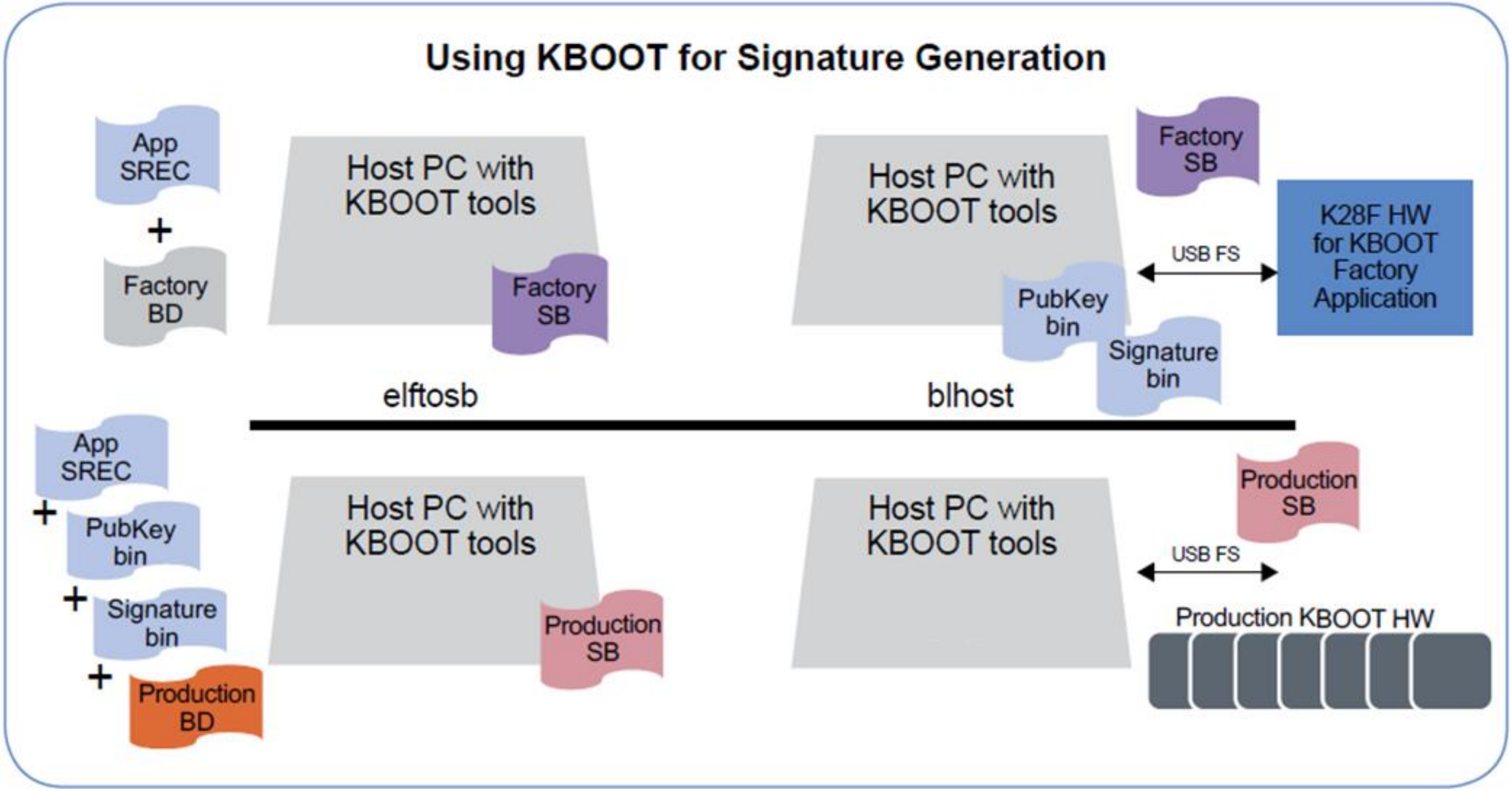


Using KBOOT for Secure Boot Functions

- Factory KBOOT application
  - This bootloader application is for use in a secure manufacturing environment. The main security functions in addition to bootloader functions are to generate a PUB/PRIV key pair and to generate the signature for application code using the **private key**.
- Production KBOOT application
  - This bootloader application is for use in a deployed device. The main security functions in addition to bootloader functions are to check the signature of application code using the **public key**, and only allow execution of the application code if the signature is authentic.



Using KBOOT Tools in Manufacturing Phase

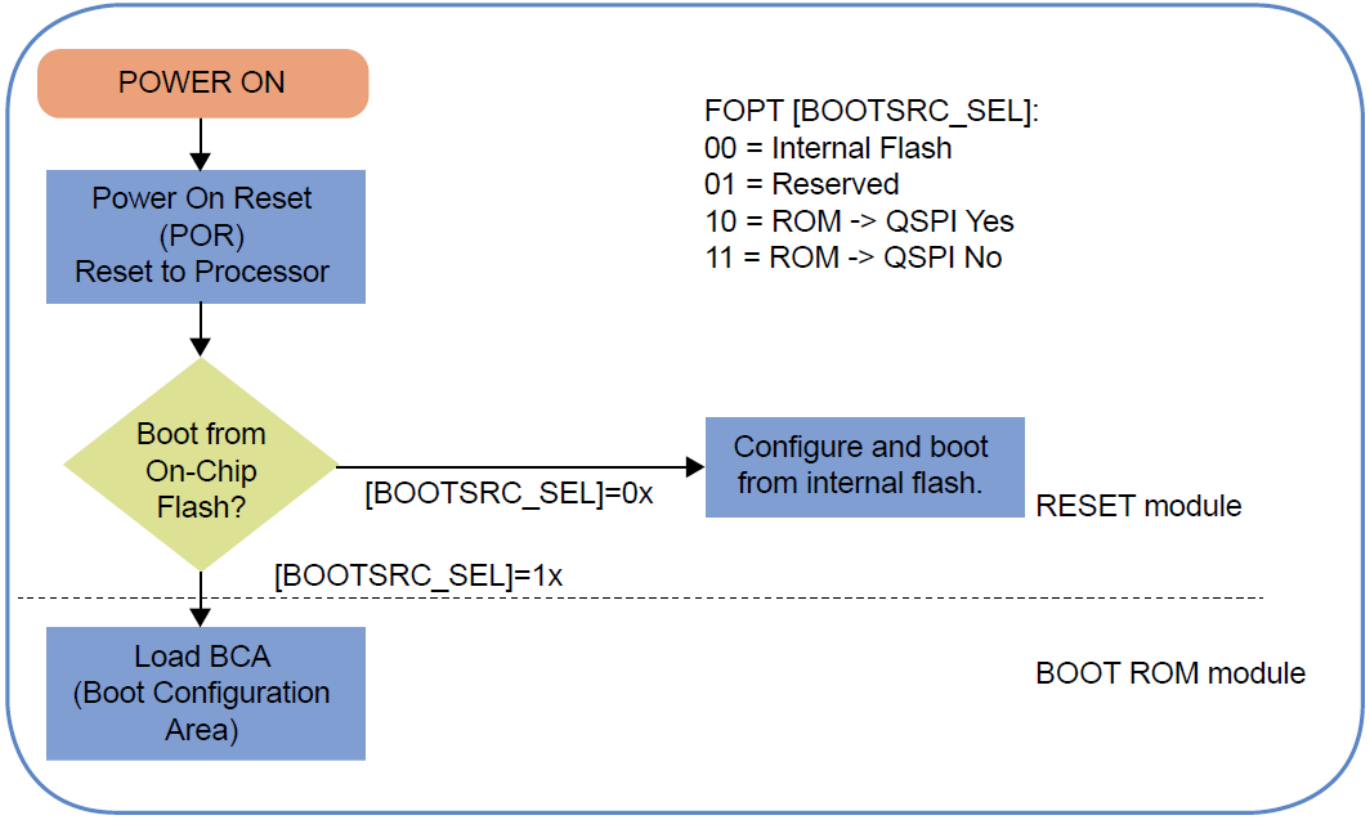


2.1

Kinetis K28F: How to Configure Hardware

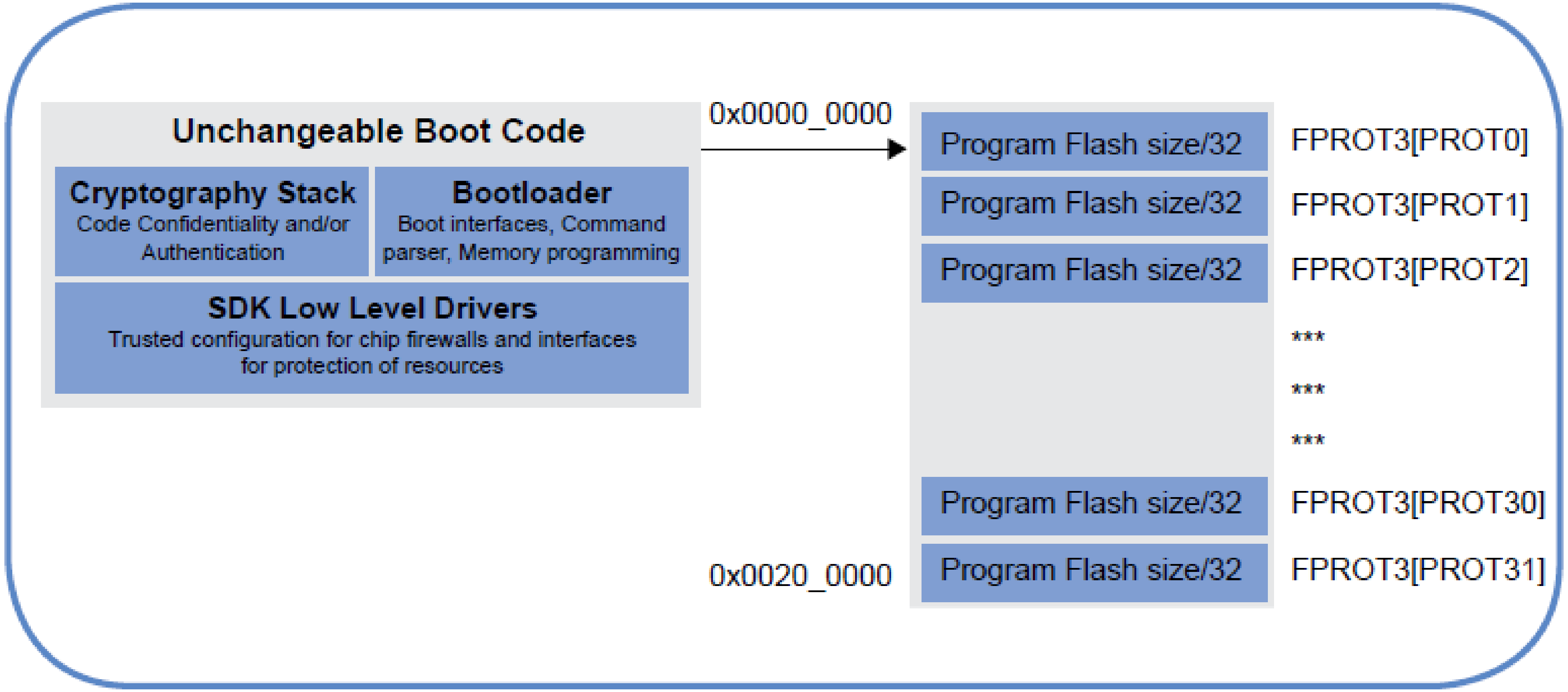
Take Control of Boot Flow

- Non-volatile control register bits [BOOTSRC_SEL]
- K28F reference manual section *7.3.4 Boot Sequence* .
- Once configured this way, the RESET module state machine of the K28_150MHz device will ensure that internal flash will be fetched and the secure boot code will always run.



Flash Block Protection

- As detailed in section 33.3.3.6 of the K28_150MHz reference manual, *“The FPROT registers define which program flash regions are protected from program and erase operations. Protected flash regions cannot have their content changed; that is, these regions cannot be programmed and cannot be erased…”*



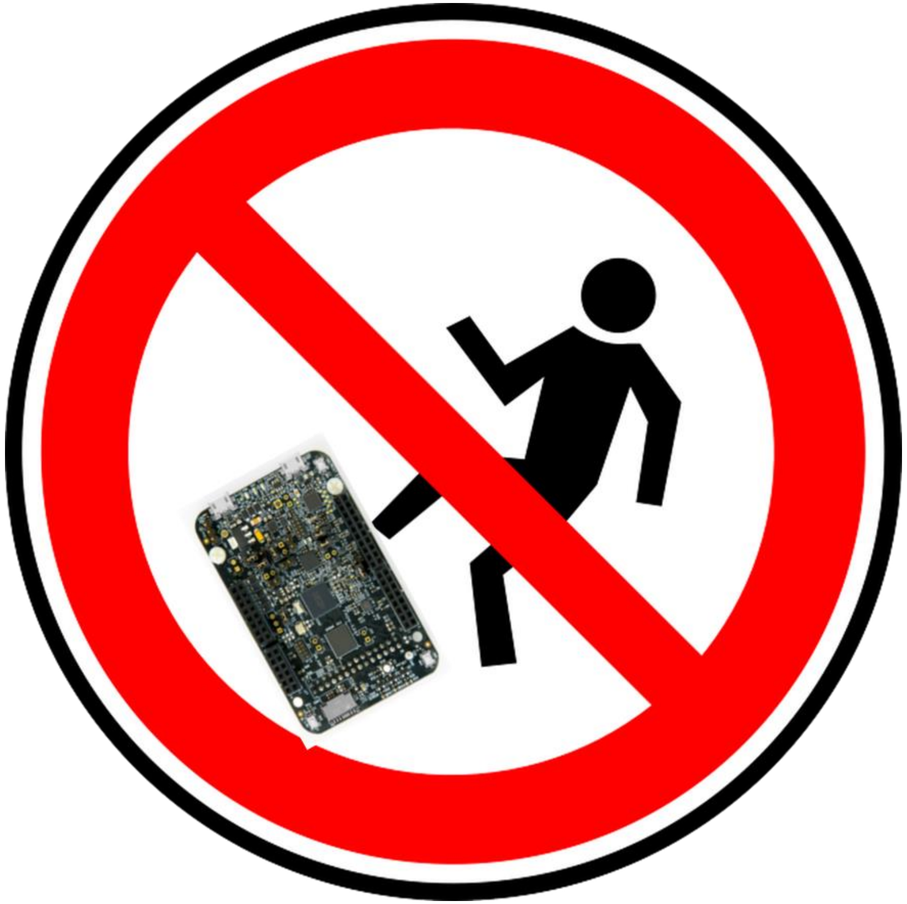
Flash Configuration Field

- The control registers for controlling boot flow, setting flash block protect and chip security settings are all part of a block of non-volatile registers as detailed in section 33.3.1 *Flash Configuration Field Configuration*

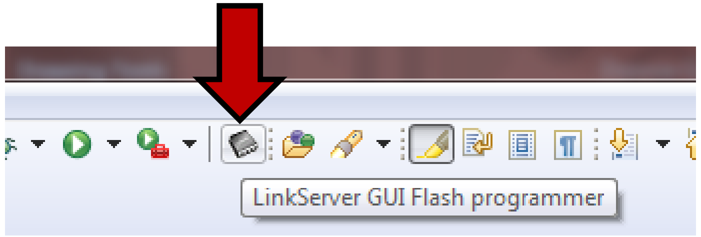
Flash Configuration Field Offset Address	Size (Bytes)	Field Description
0x0_0400 - 0x0_0407	8	Backdoor comparison key.
0x0_0408 - 0x0_040B	4	Program flash protection bytes. Refer to the description of the Program Flash Protection Registers (FPROT0-3).
0x0_040F	1	Reserved
0x0_040E	1	Reserved
0x0_040D	1	Flash nonvolatile option byte. Refer to the description of the Flash Option Register (FOPT).
0x0_040C	1	Flash nonvolatile option byte. Refer to the description of the Flash Security Register (FSEC).

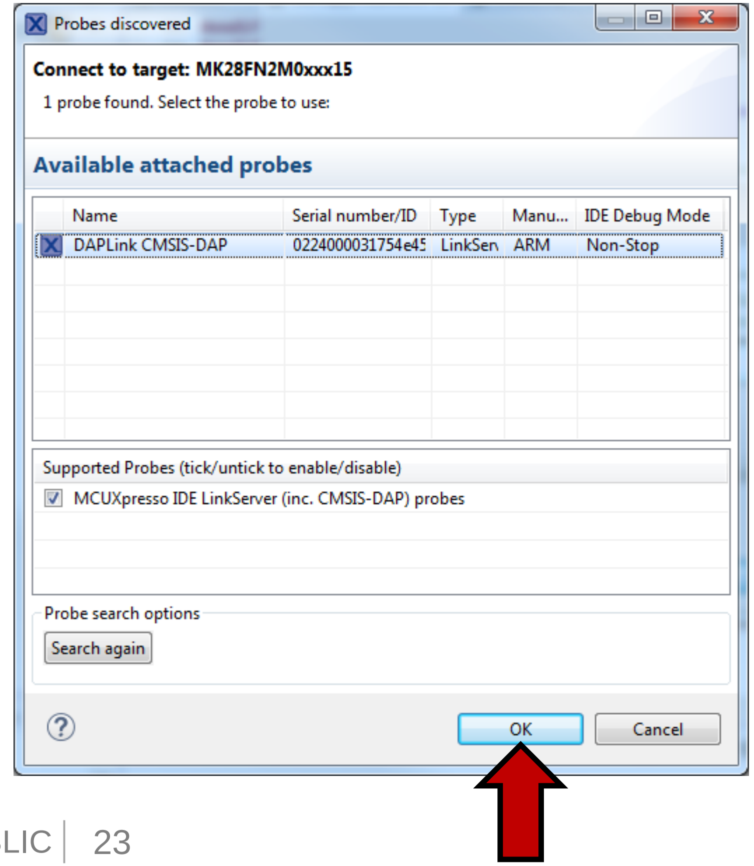
Warning: Use Caution

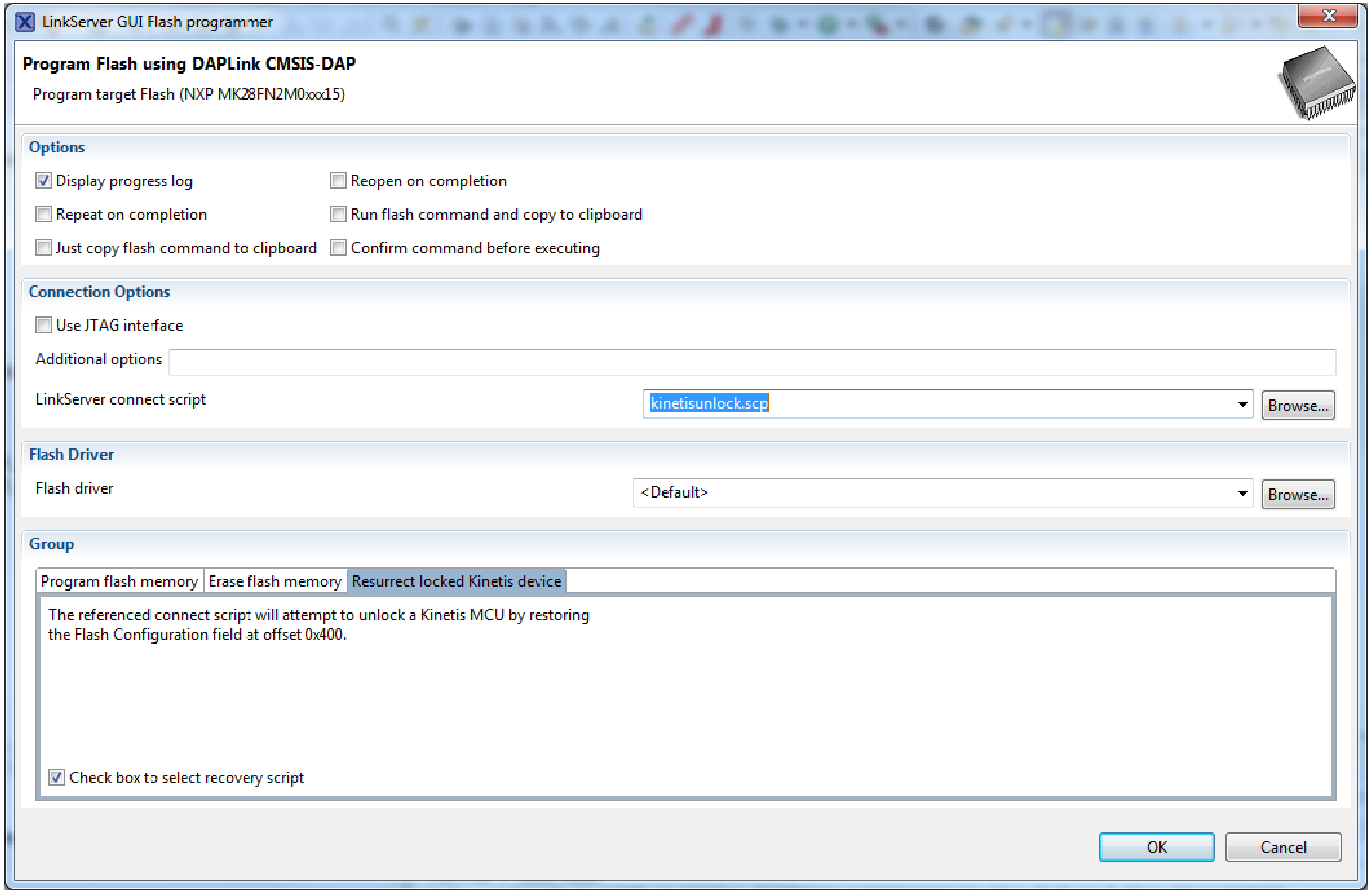
- Extreme care must be taken when using these fields because the chip can be locked out in flash programming if the program image does not have these fields setup correctly.



Recovery for Security Locked Devices in MCUXpresso IDE

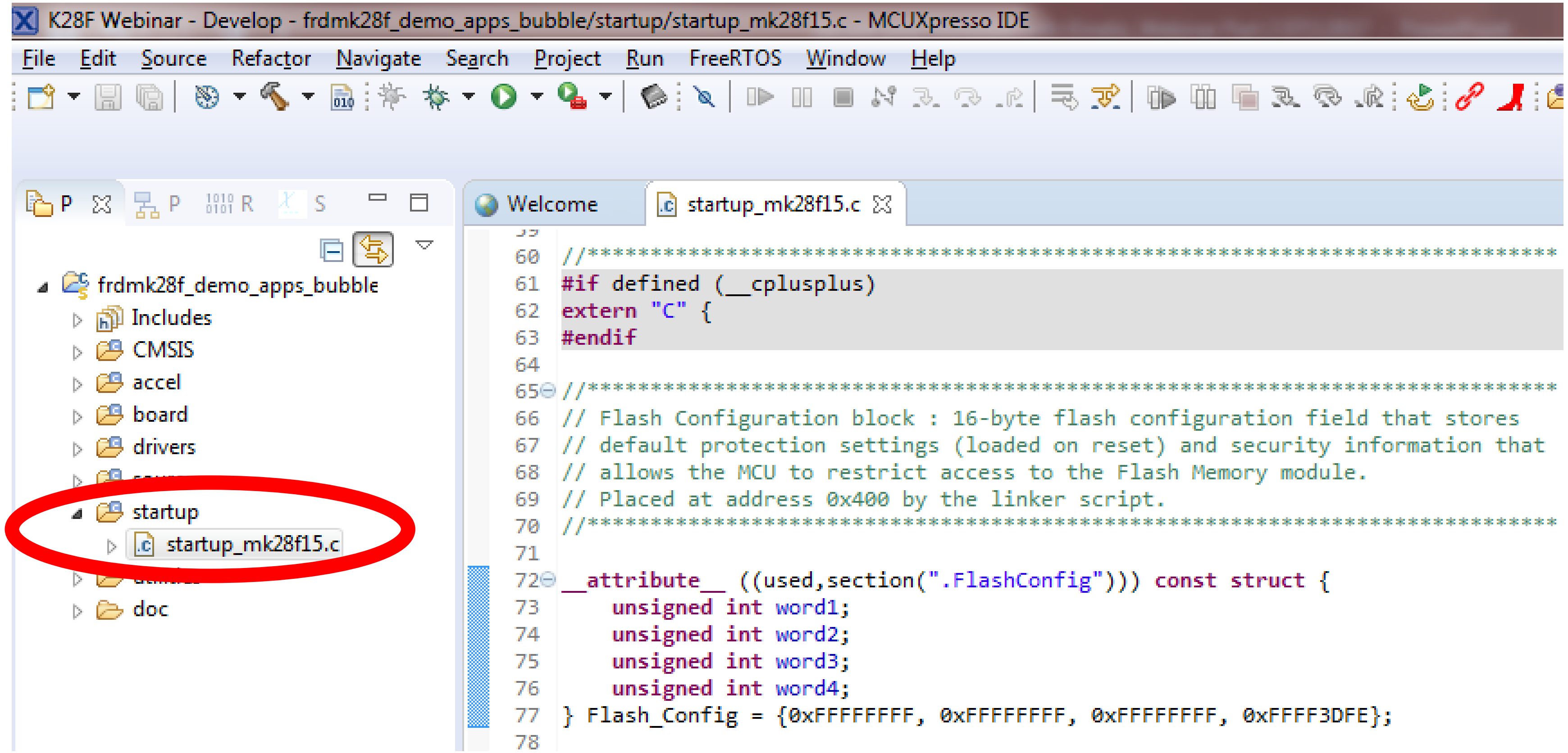






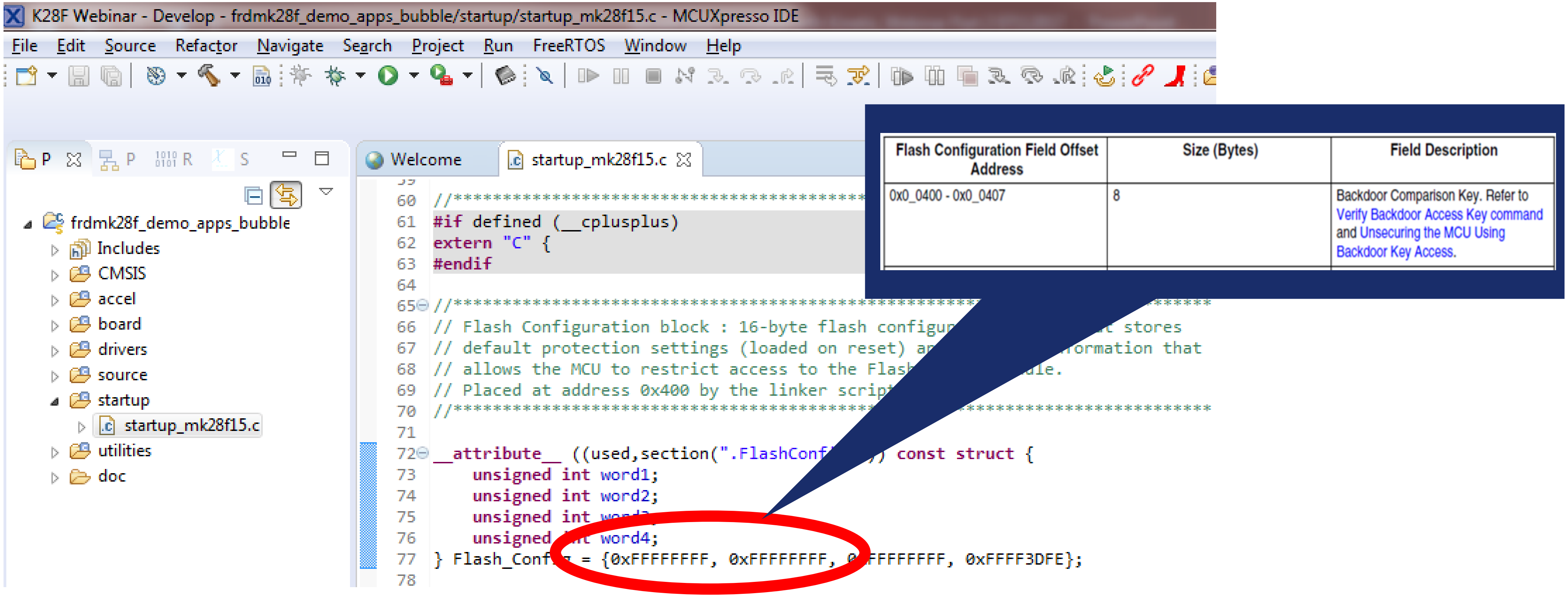
MCUXpresso and Setting Flash Configuration Field

- The Flash Configuration Field is handled by the Managed Linker Script mechanisms of MCUXpresso IDE



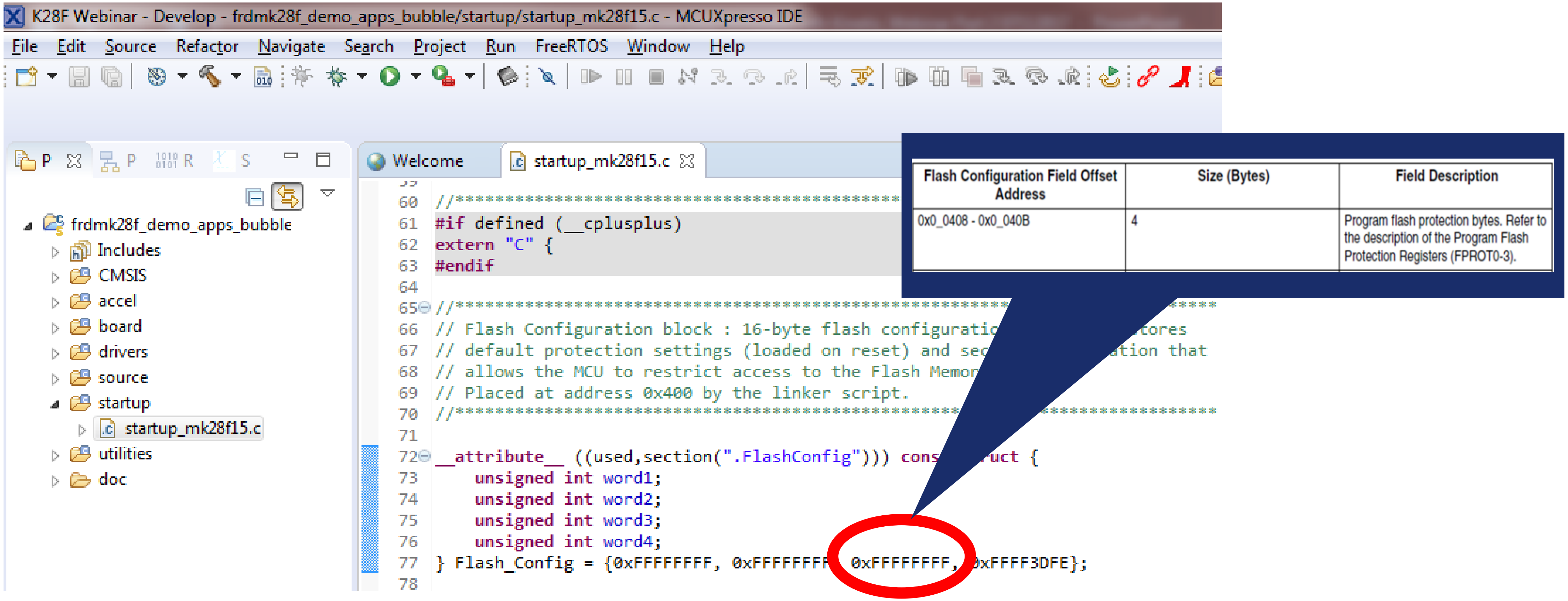
MCUXpresso and Setting Flash Configuration Field

- The Flash Configuration Field is handled by the Managed Linker Script mechanisms of MCUXpresso IDE



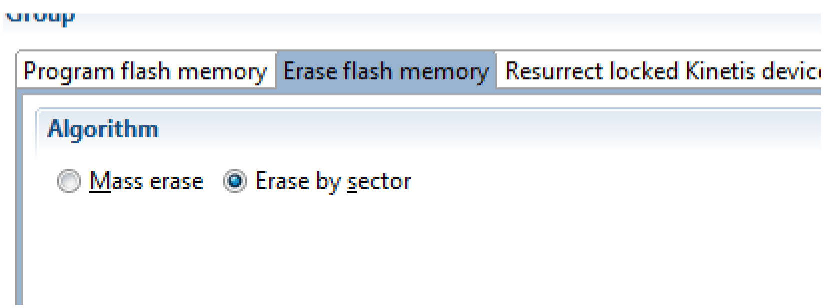
MCUXpresso and Setting Flash Configuration Field

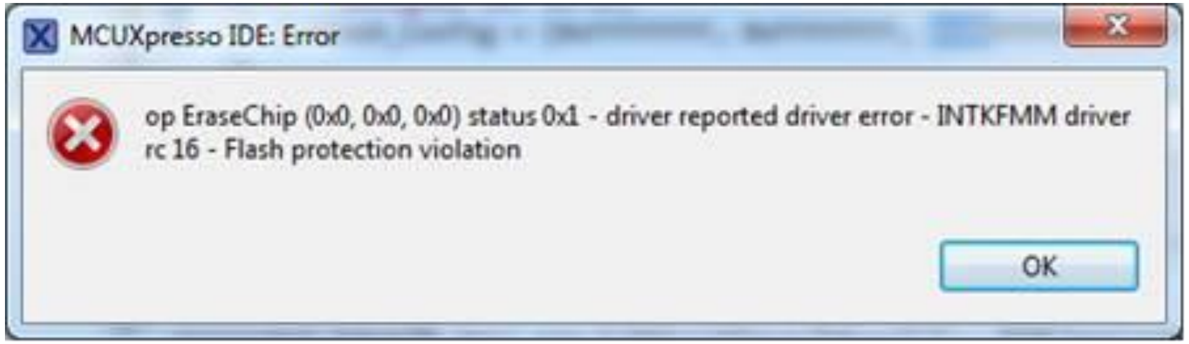
- The Flash Configuration Field is handled by the Managed Linker Script mechanisms of MCUXpresso IDE



MCUXpresso and Setting Flash Configuration Field

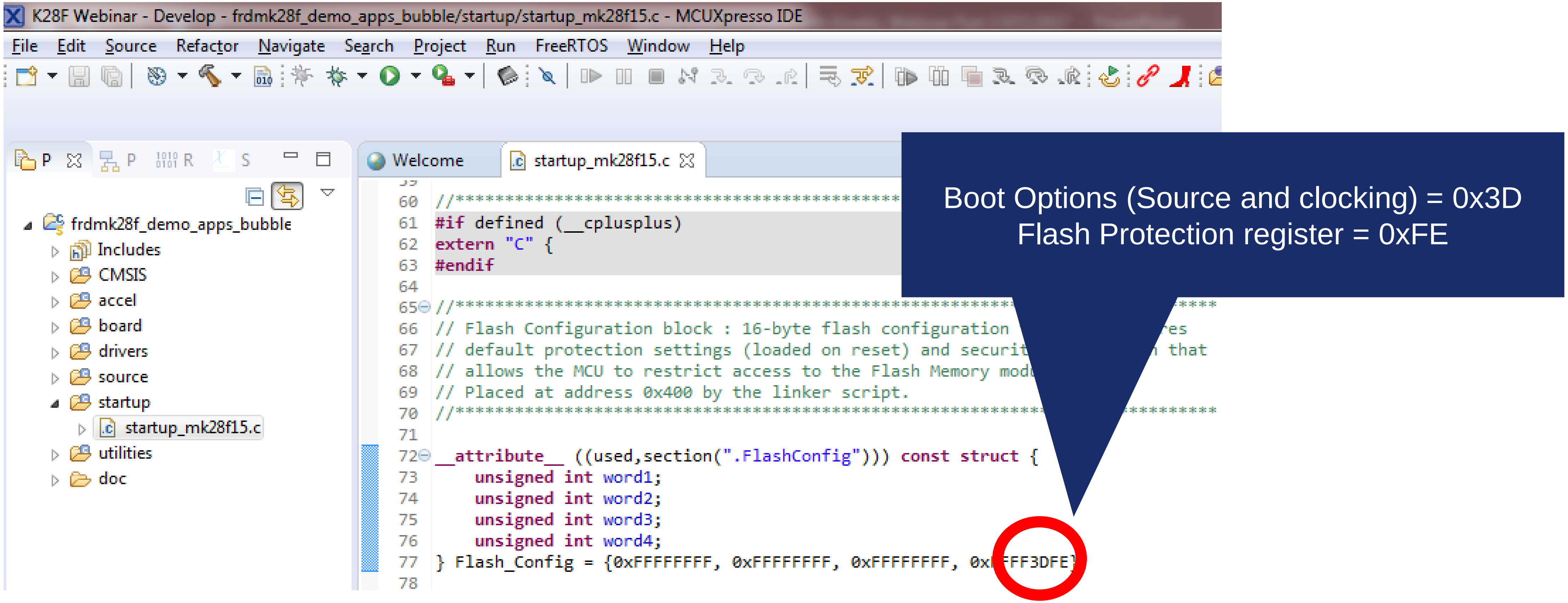
```
__attribute__ ((used,section(".FlashConfig"))) const struct {
    unsigned int word1;
    unsigned int word2;
    unsigned int word3;
    unsigned int word4;
} Flash_Config = {0xFFFFFFFF, 0xFFFFFFFF, 0x00FFFFFF, 0xFFFF3DFE};
```





MCUXpresso and Setting Flash Configuration Field

- The Flash Configuration Field is handled by the Managed Linker Script mechanisms of MCUXpresso IDE



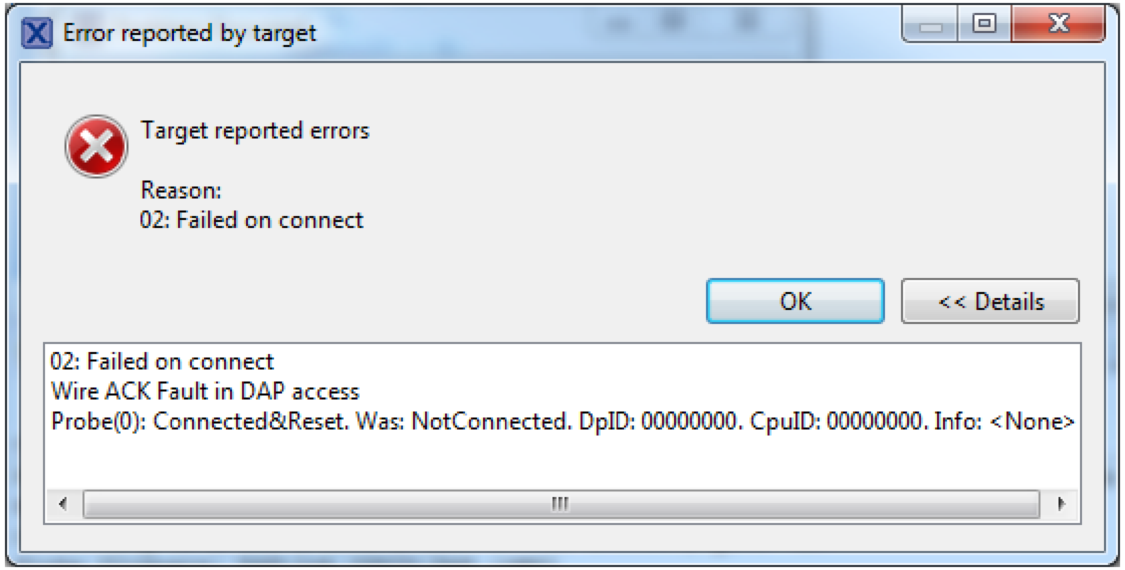
MCUXpresso and Setting Flash Configuration Field

- The Flash Configuration Field is handled by the Managed Linker Script mechanisms of MCUXpresso IDE

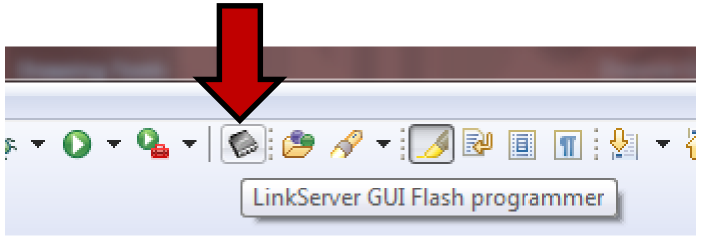
Address: 4002_0000h base + 2h offset = 4002_0002h

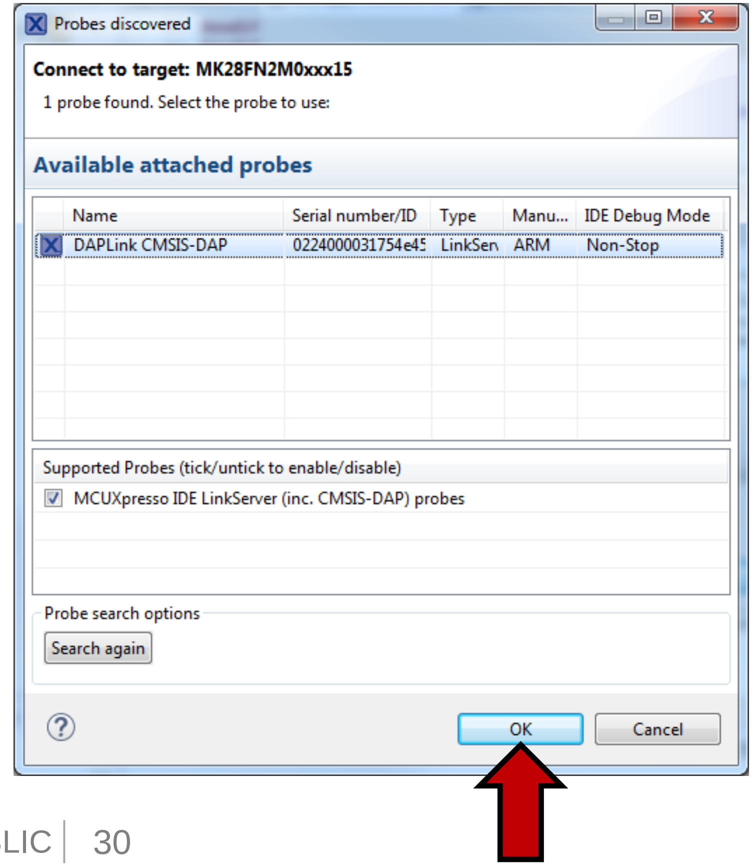
Bit	7	6	5	4	3	2	1	0
Read	KEYEN		MEEN		FSLACC		SEC	
Write								
Reset	x*	x*	x*	x*	x*	x*	x*	x*

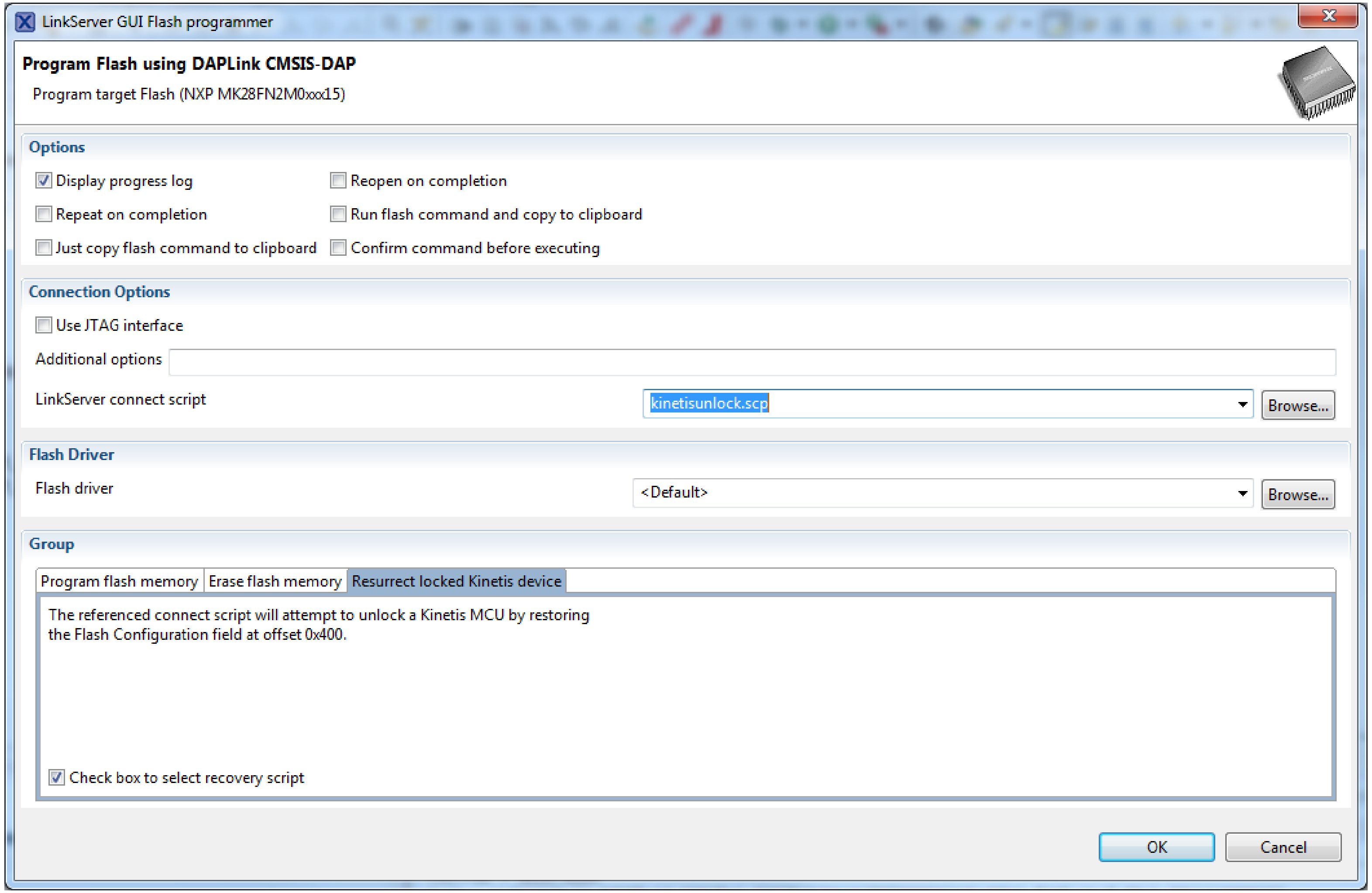
* Notes:
- x = Undefined at reset.



Recovery for Security Locked Devices







2.2

mbed TLS

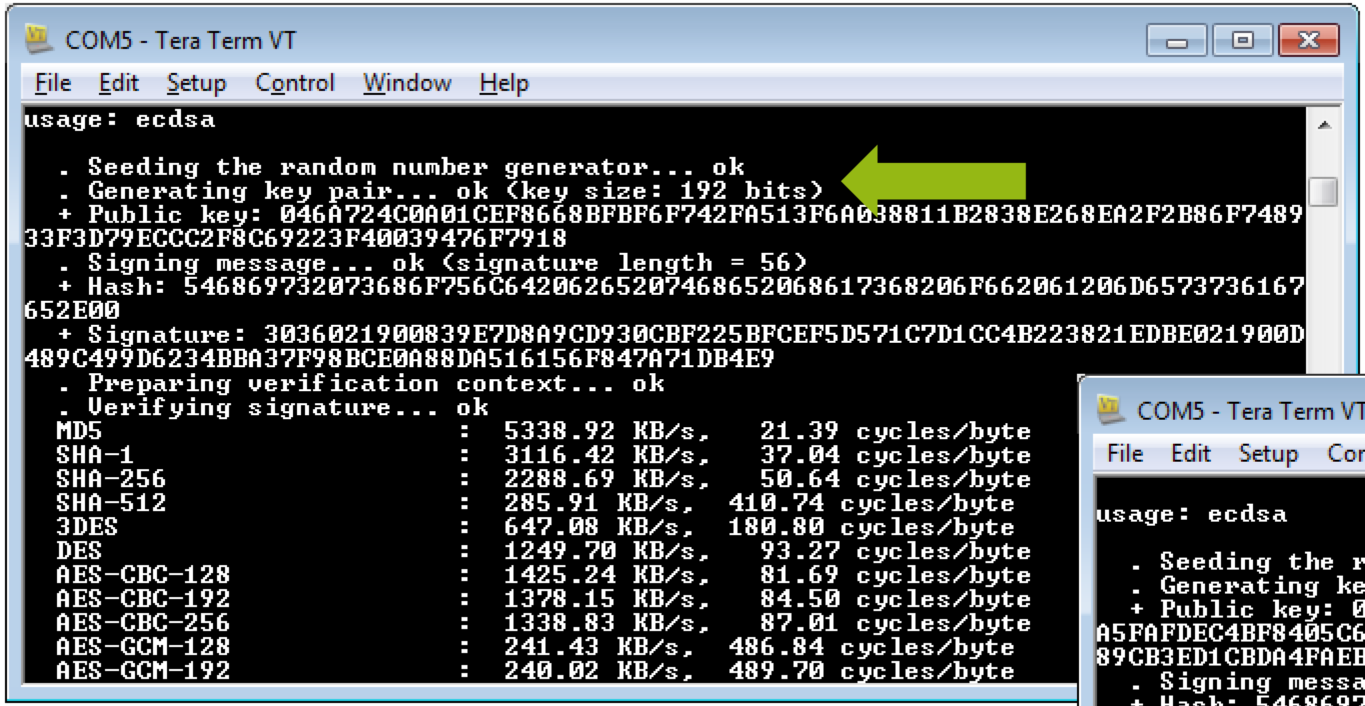
ARM mbed TLS Files and Relevant APIs

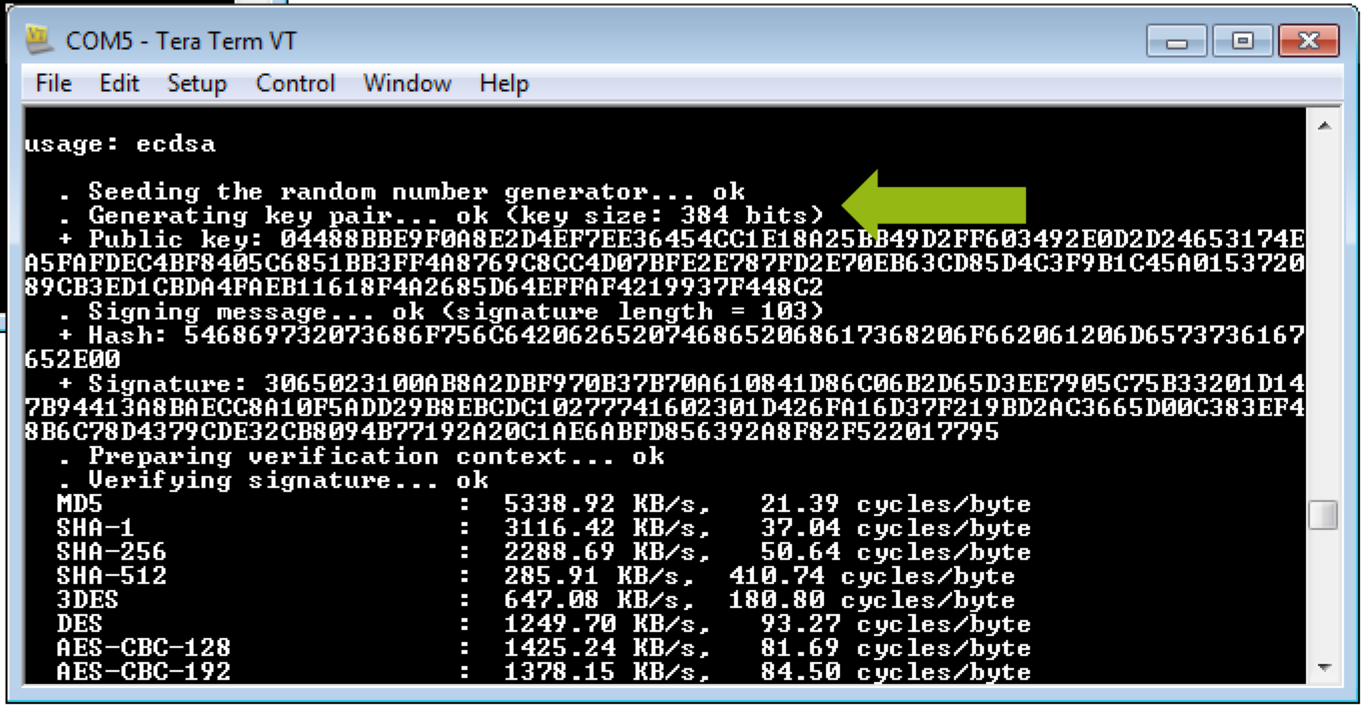
- Ecdsa example program path
  - SDK_2.2_FRDM-K28F\middleware\mbedtls_2.3.0\programs\pkey\ecdsa.c

```c
int main( int argc, char *argv[] )
{
    int ret;
    mbedtls_ecdsa_context ctx_sign, ctx_verify;
    mbedtls_entropy_context entropy;
    mbedtls_ctr_drbg_context ctr_drbg;
    unsigned char hash[] = "This should be the hash of a message.";
    unsigned char sig[512];
    size_t sig_len;
    const char *pers = "ecdsa";
    ((void) argv);

    mbedtls_ecdsa_init( &ctx_sign );
    mbedtls_ecdsa_init( &ctx_verify );
    mbedtls_ctr_drbg_init( &ctr_drbg );

    memset(sig, 0, sizeof( sig ) );
    ret = 1;

    if( argc != 1 )
    {
        mbedtls_printf( "usage: ecdsa\n" );

#if defined(_WIN32)
        mbedtls_printf( "\n" );
#endif

        goto exit;
    }

    /*
     * Generate a key pair for signing
     */
    mbedtls_printf( "\n  . Seeding the random number generator..." );
    fflush( stdout );

    mbedtls_entropy_init( &entropy );
    if( ( ret = mbedtls_ctr_drbg_seed( &ctr_drbg, mbedtls_entropy_func, &entropy,
                               (const unsigned char *) pers,
                               strlen( pers ) ) ) != 0 )
    {
        mbedtls_printf( " failed\n  ! mbedtls_ctr_drbg_seed returned %d\n", ret );
        goto exit;
    }
```

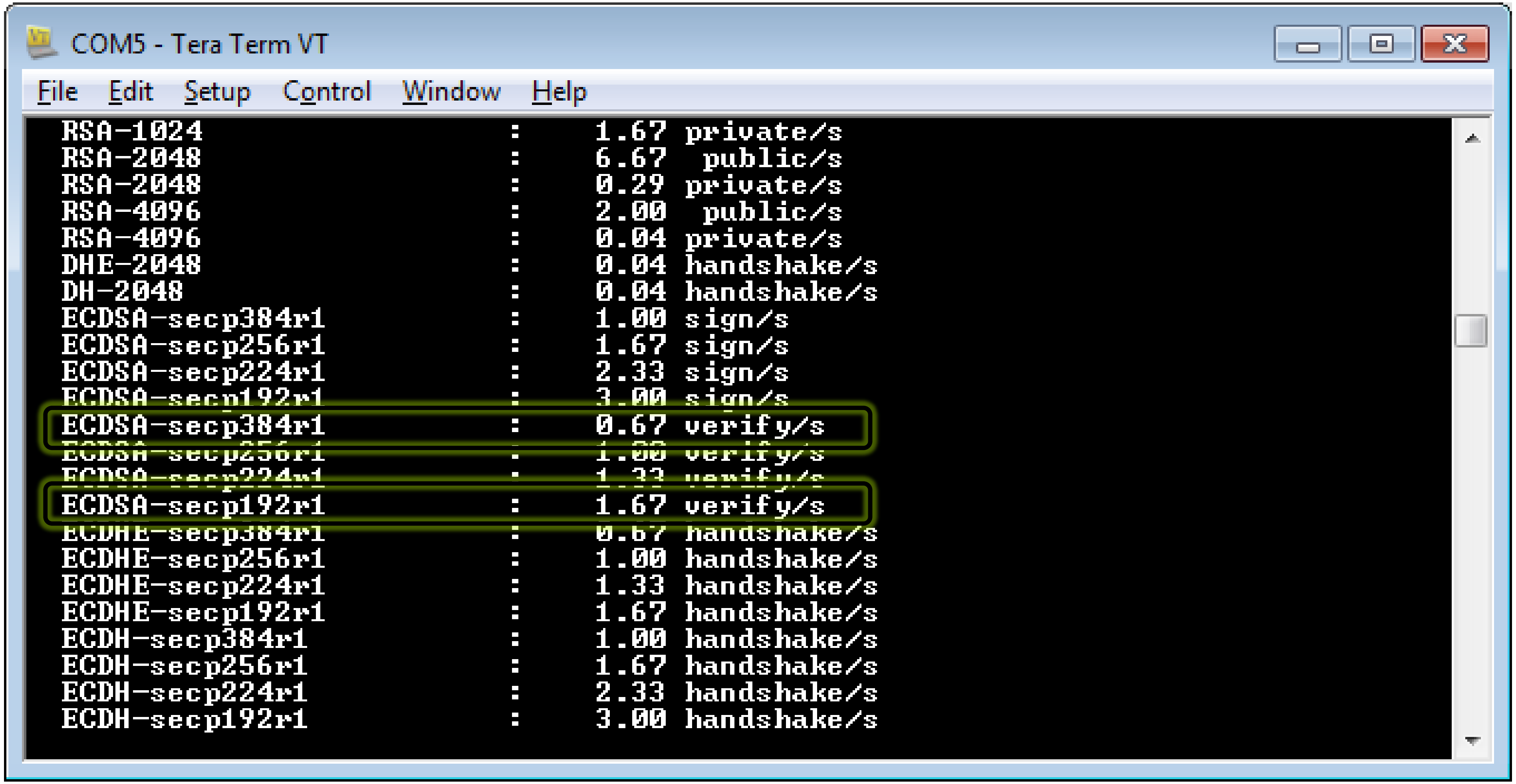
```c
     * Sign some message hash
     */
    mbedtls_printf( "  . Signing message..." );
    fflush( stdout );

    if( ( ret = mbedtls_ecdsa_write_signature( &ctx_sign, MBEDTLS_MD_SHA256,
                                       hash, sizeof( hash ),
                                       sig, &sig_len,
                                       mbedtls_ctr_drbg_random, &ctr_drbg ) ) != 0 )
    {
        mbedtls_printf( " failed\n  ! mbedtls_ecdsa_genkey returned %d\n", ret );
        goto exit;
    }
    mbedtls_printf( " ok (signature length = %u)\n", (unsigned int) sig_len );

    dump_buf( "  + Hash: ", hash, sizeof hash );
    dump_buf( "  + Signature: ", sig, sig_len );

    /*
     * Transfer public information to verifying context
     *
     * We could use the same context for verification and signatures, but we
     * chose to use a new one in order to make it clear that the verifying
     * context only needs the public key (Q), and not the private key (d).
     */
    mbedtls_printf( "  . Preparing verification context..." );
    fflush( stdout );

    if( ( ret = mbedtls_ecp_group_copy( &ctx_verify.grp, &ctx_sign.grp ) ) != 0 )
    {
        mbedtls_printf( " failed\n  ! mbedtls_ecp_group_copy returned %d\n", ret );
        goto exit;
    }

    if( ( ret = mbedtls_ecp_copy( &ctx_verify.Q, &ctx_sign.Q ) ) != 0 )
    {
        mbedtls_printf( " failed\n  ! mbedtls_ecp_copy returned %d\n", ret );
        goto exit;
    }

    ret = 0;

    /*
     * Verify signature
     */
    mbedtls_printf( " ok\n  . Verifying signature..." );
    fflush( stdout );
```

mbed TLS ecdsa.c Example





Kinetis K28F mbed TLS ecdsa Benchmark



ARM mbed TLS Files and Relevant APIs

- Factory Application vs Production secure boot loader

Key Generation/Signature

mbedtls_ctr_drbg_init
mbedtls_entropy_init
mbedtls_ctr_drbg_seed
mbedtls_ecdsa_genkey

mbedtls_ecdsa_write_signature

Hash of Firmware

mbedtls_sha256

Signature Verification

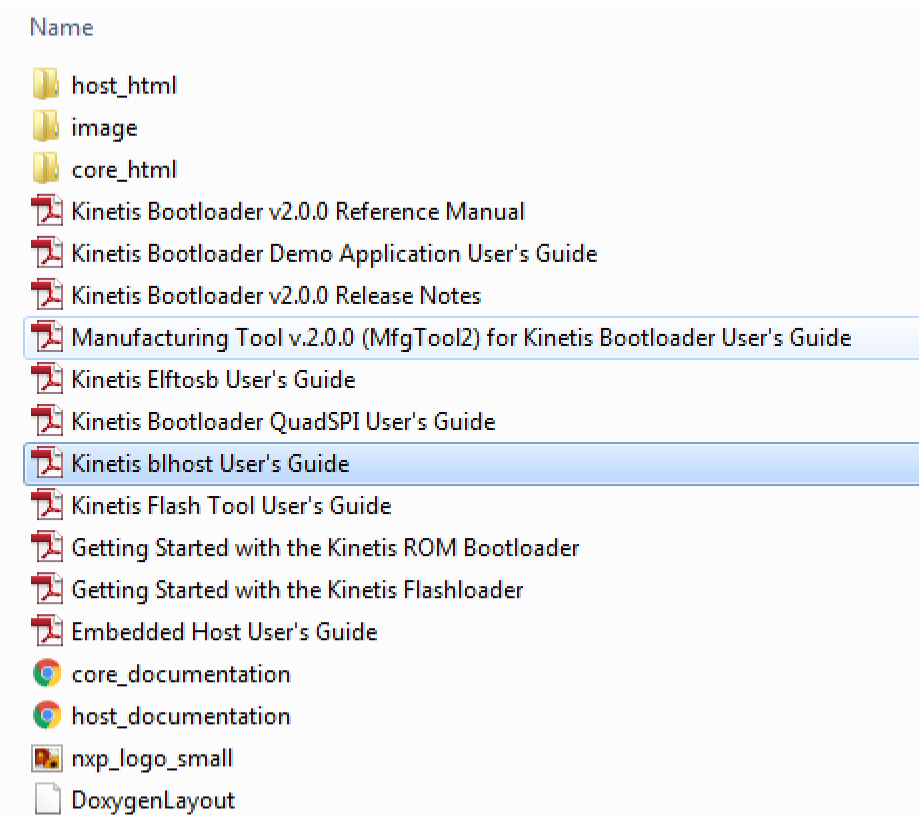
mbedtls_ecp_group_copy
mbedtls_ecp_copy

mbedtls_ecdsa_read_signature

2.3

KBOOT Tools

KBOOT Tools: Documentation



- Kinetis KBOOT Documentation
  - Getting started documents
  - Includes applications users guides
  - Specific users guides for tools
    - Blhost users guide for interfacing to a Kinetis device running KBOOT
      - Blhost commands allow manufacturing sites to extract signature and public key information
    - ElftoSB users guide for generating secure binaries
      - ElftoSB is used to group binaries for building the production application

Blhost Tool: Documentation of Commands

- Blhost users guide Section 4.2

4.2.7 read-memory <addr> <byte_count> [<file>]

Example: -- `read-memory 0x3c0 32 myConfigData.dat`

Used to export pubkey.bin and signature.bin to be used in production application

4.2.12 call <address> <arg>

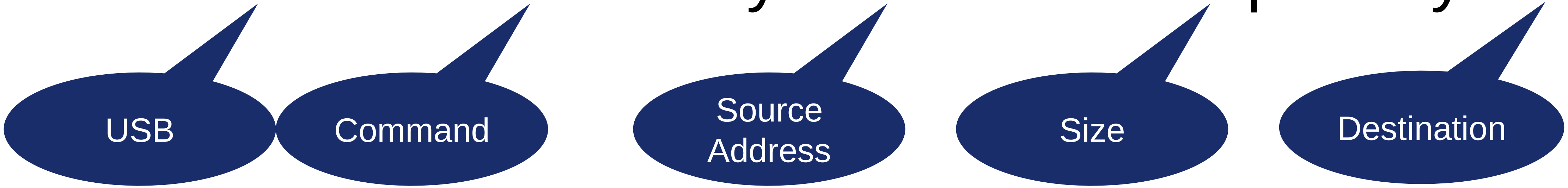
Example: -- `call 0x6000 0x21`

4.2.10 receive-sb-file <file>

Example: -- receive-sb-file mySecureImage.sb

Blhost Tool: Commands Exporting Binaries

- blhost –u -- read-memory 0x2000040 24 pubkey.bin




```
C:\Users\r1aald\Documents\Work\Training Materials\Webinar\NXP_Kinetis_Bootloader
_2_0_0\NXP_Kinetis_Bootloader_2_0_0\bin\Tools\blhost\win>blhost -u -- read-memor
y 0x20000040 24 pubkey.bin
Inject command 'read-memory'
Successful response to command 'read-memory'
(1/1)100% Completed!
Successful generic response to command 'read-memory'
Response status = 0 (0x0) Success.
Response word 1 = 24 (0x18)
Read 24 of 24 bytes.
```

Elftosb Tool Documentation of BD file

3.1.1.3 Sources

The sources block is where the input files are listed and assigned the identifiers with which they are referenced throughout the rest of the command file. Each statement in the sources block consists of an assignment operator (the "=" character) with the source name identifier on the left hand side, and the source's path value on the right hand side. Individual source definitions are terminated with a semicolon.

The syntax for the source value depends on the type of source definition. The two types are explicit paths and externally provided paths. Sources with explicit paths simply list the path to the file as a quoted string literal.

The external sources use an integer expression to select one of the positional parameters from the command line. This type of source allows the user to easily vary the input file by changing the command line arguments.

The sorce definition grammar follows this form:

```
source_def ::= IDENT '=' source_value ( '(' source_attr_list? ')' )?
  ;
source_value ::= STRING_LITERAL
  | 'extern' '(' int_const_expr ')'
  ;
source_attr_list
  ::= source_attr ( ',' source_attr )*
  ;
source_attr ::= IDENT '=' const_expr
  ;
```

There source definition can optionally have a list of source attributes contained in parentheses at the end of the definition. These attributes are the same as options in an options block but only a few options apply to sources. See Table 2 for the complete list of options.

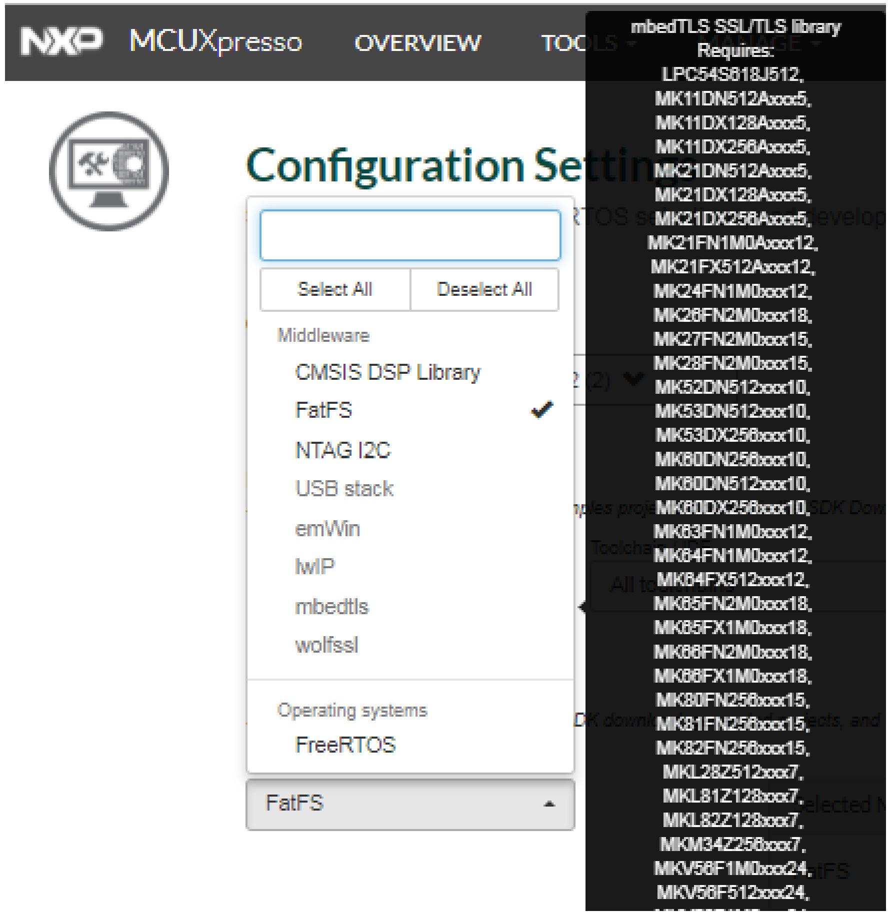
```
# The sources block assigns file names to identifiers
sources {

   # SREC File path
    mySrecFile = "IoT_App_code.srec";
   # pubkey file path
    pubKeyBlock = "pubkey.bin";
    # signature
    signatureBlock = "sign.bin";
   }
```

3

Portability

Applying This Solution to Other Platforms

- Kinetis K28F is highly capable processor with large memory footprint, but it may not fit for your every IoT edge node application
  - Size constraints
  - Performance/power limitations
  - Not the right I/O voltage or peripherals
  - Boot time



- Migrating within the Kinetis MCU portfolio
  - mbed TLS support allows portability

Secure Card Reader Solution

SLN-POS-RDR: Point of Sale (POS) Reader Solution

OVERVIEW | GETTING STARTED | DOCUMENTATION | SOFTWARE & TOOLS | TRAINING & SUPPORT

Jump To

- Overview & Features
- Kit Contains
- Supported Devices
- Target Applications

Overview

The SLN-POS-RDR Point of Sale (POS) Reader Solution enables you to quickly add a PCI®- and EMVCo®-compliant PIN entry device (PED), NFC reader, chip card reader and magnetic stripe reader (MSR) to any design to enable credit card payment. Many companies are creating products today that would benefit from adding payment capabilities to the design. However, getting the necessary PCI and EMVCo certifications are a significant engineering and development barrier. This solution is pre-certified for EMVCo and PCI PTS standards to give companies confidence that they will have a high likelihood of passing certification the first time without the added

Features

- Chip-and-PIN keypad based on Cirque® SecureSense™ technology
- EMVCo Level 1 CT/CL stacks by NXP®
- EMVCo Level 2 CT/CL stacks by Cardtek
- EMVCo and PCI4.x Certification
  - EMVCo Pre-certification on Level 1 CT/CL by FIME
  - PCI 4.1 Pre-certification on the K81 performed by Infogard
  - PCI 4.1 PIN Entry Device (PED) Certification by Infogard
- Kinetis® K81 Secure MCU

4

ARM® TrustZone for ARMv8-M

Objective: Security for All Embedded Applications

Root-of-trust applications - IoT

Trusted software

Trusted hardware

Crypto | Secure system | Secure storage | TRNG*

IP Protection

Valuable firmware

Trusted drivers

Trusted hardware

Sandboxing

Certified OS / functionality

Trusted drivers

Trusted hardware

Untrusted | Trusted

ARM TRUSTZONE

System security

Industry standard

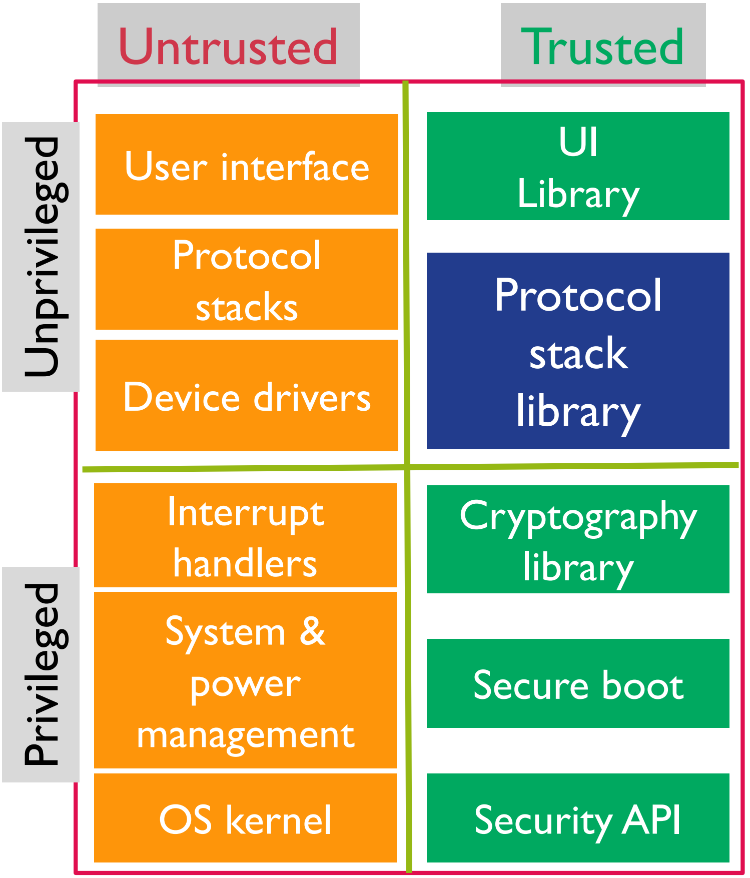
Developer friendly

Ecosystem friendly

Embedded friendly

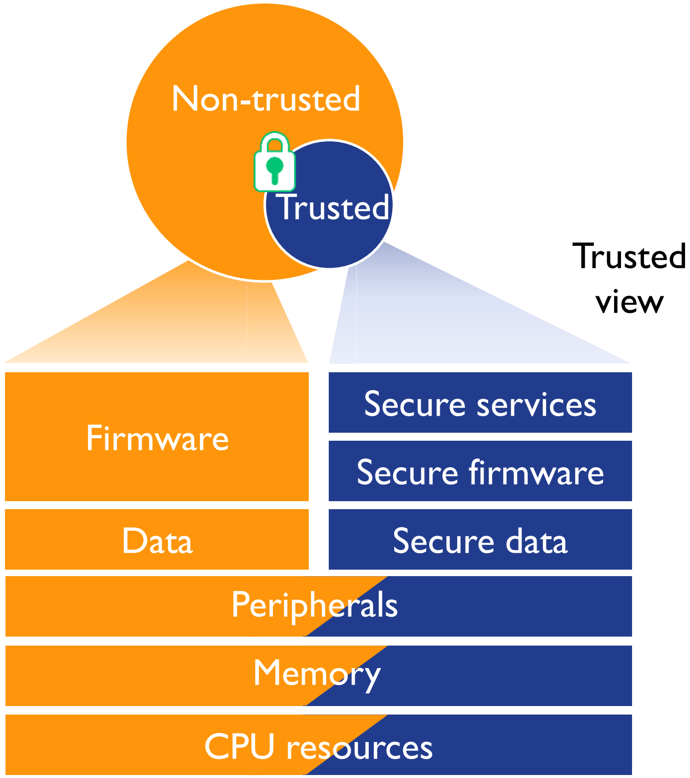
*True random number generator

Future Software Architecture



Software from MCU system developers

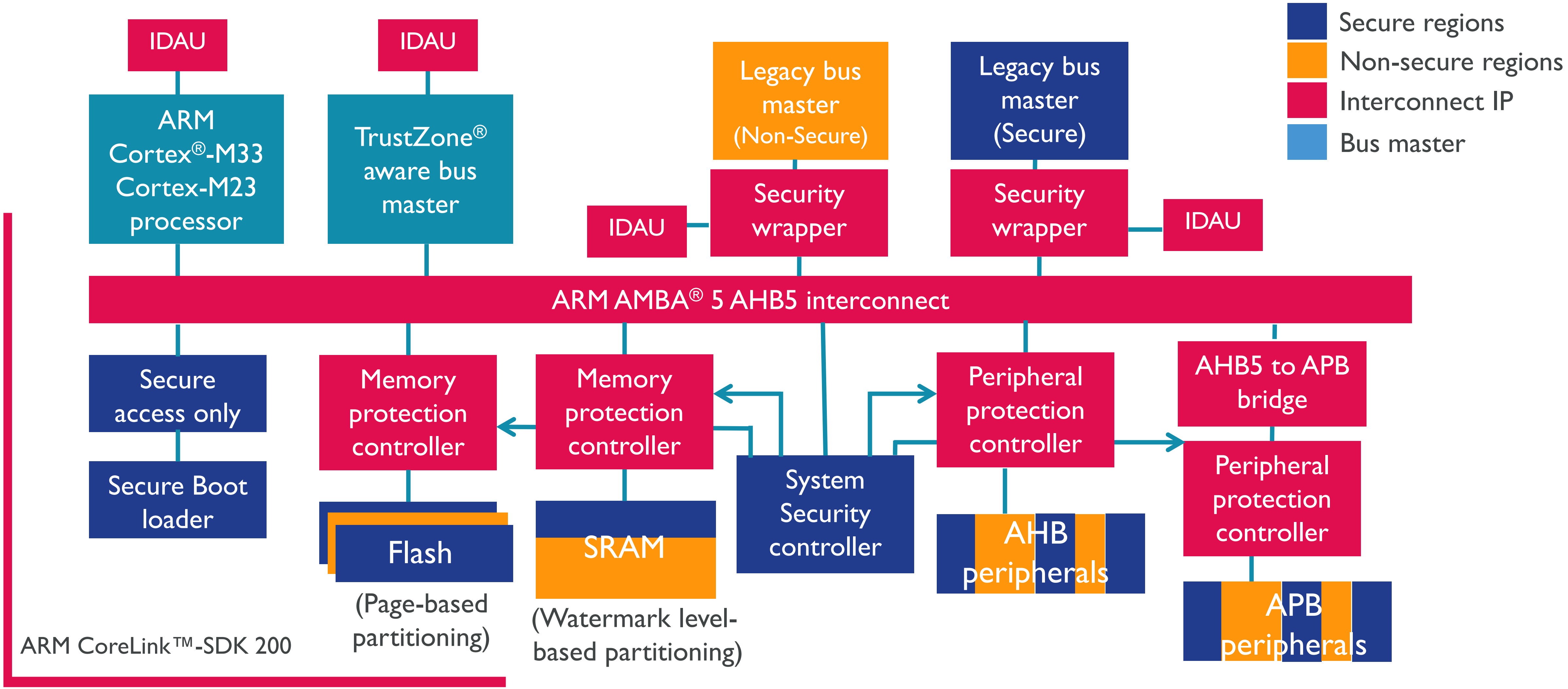
Built-in firmware



Two worlds - one CPU
Real-time transition*

*≤2 cycles

Future Device Architecture



Conclusion

In today's connected world, the protection of firmware is an essential component to delivering solutions that safeguard device manufacturers and their customers. Essential to sustaining end-to-end security is a secure and trusted boot, which can be achieved with the right MCU hardware capabilities and ARM mbed TLS. NXP's microcontrollers contain the hardware features and software enablement that can be integrated to strengthen end device security and protect value. As the drive towards lower power and higher performance efficiency for IoT edge nodes continues, future capabilities in embedded controllers and ARM processors will provide the basis for future security solutions for the IoT.

http://www.nxp.com/video/how-to-protect-your-firmware-against-malicious-attacks-using-the-latest-kinetis-development-board:SECURE-YOUR-FIRMWARE-WITH-KINETIS

Resources

- http://www.nxp.com/video/how-to-protect-your-firmware-against-malicious-attacks-using-the-latest-kinetis-development-board:SECURE-YOUR-FIRMWARE-WITH-KINETIS
- http://www.nxp.com/products/reference-designs/kinetis-bootloader:KBOOT?&tid=vanKBOOT
- https://community.arm.com/processors/trustzone-for-armv8-m/
- https://developer.arm.com/products/processors/cortex-m/cortex-m23
- https://developer.arm.com/products/processors/cortex-m/cortex-m33

White paper

Prevent edge node attacks by securing your firmware

Configuring Kinetis® MCU capabilities with ARM® mbed™ TLS for a secure boot

***Donnie Garcia**, Solutions Architect for Secure Transactions, NXP*

***Diya Soubra**, Senior Product Marketing Manager, ARM*