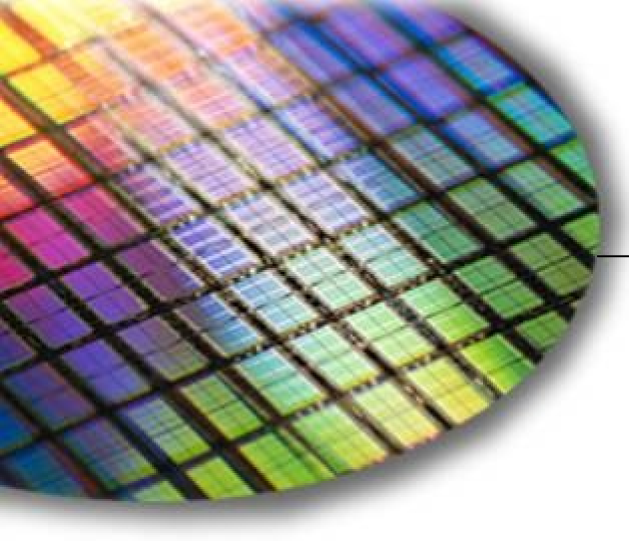


The World Leader in High Performance Signal Processing Solutions

Lockbox™ Secure Technology on Blackfin Processors

BLACKfin™

R.SALVETAT

ANALOG DEVICES

Agenda

- General security overview
- Lockbox overview:
  - Digital Signature (ECDSA) in Blackfin
  - Lockbox programming interface
  - Modifying VisualDSP++ linker files (LDF) for Lockbox
  - Developing practical Lockbox applications using overlays.
- Hands-on security example
- Q & A

- **Course prerequisites:**
  - View BOLD Lockbox Module on web (http://my.analog.com/onlinetraining/Static/BOLDList.html#ADEV021)
  - Basic understanding of security concepts and overlays is recommended but not required

Lockbox Secure Technology Benefits

Confidentiality

Cryptographic encryption/decryption supports situations that require the ability to prevent unauthorized users from seeing and using designated files and streams.

Lockbox's secure processing environment (Secure Mode) and secure memory support confidentiality.

Integrity

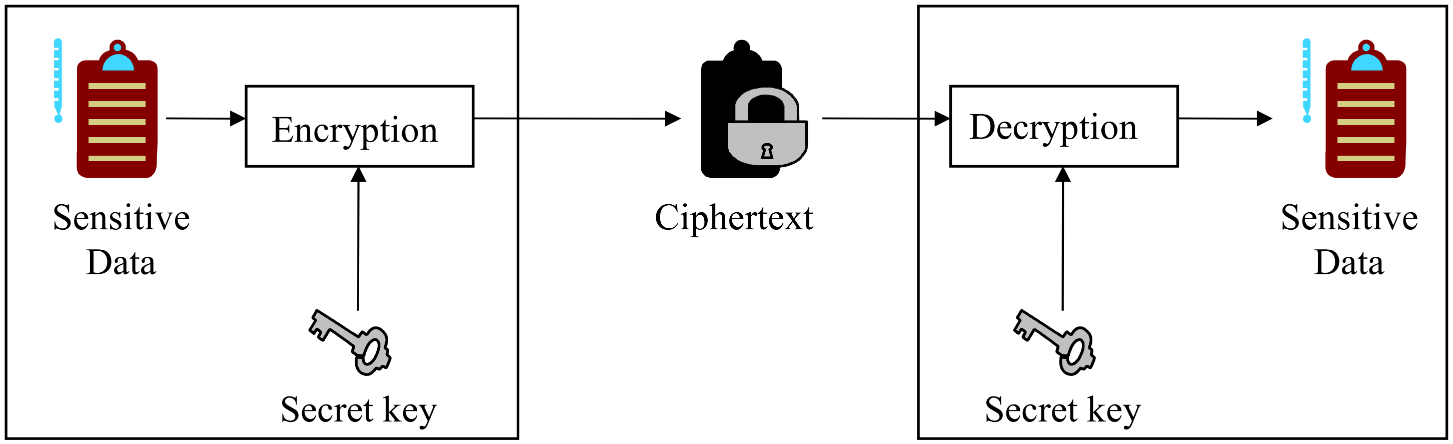
Developers can use a digital signature authentication process to ensure that the message or the content of the storage media has not been altered in any way. Integrity can be verified using Lockbox's authentication

of digital signatures.

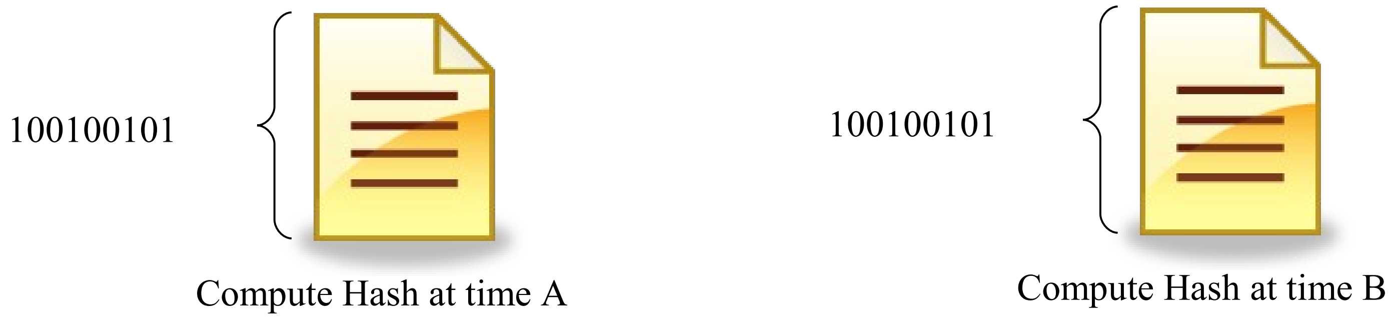
Authenticity/Origin verification

Lockbox secure technology allows for verification of a code image against its embedded digital signature, and provides for a process to identify entities and data origins.

Confidentiality – Symmetric Ciphers (AES, DES)



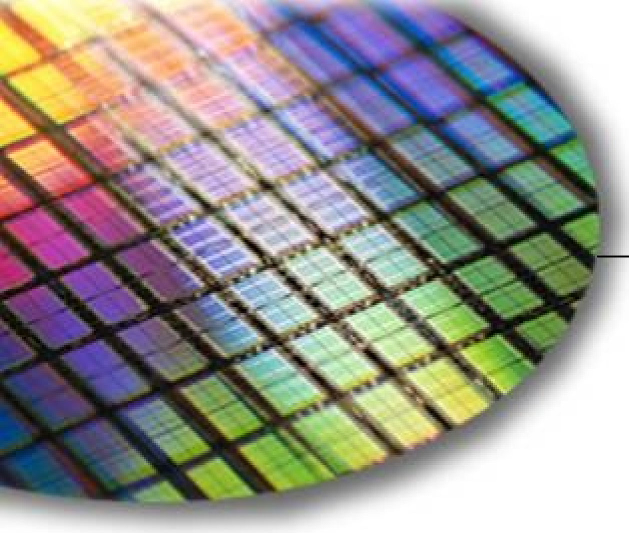
Integrity – Hash (SHA-1, SHA-256, SHA-512)



Authenticity – Digital Signatures (RSA, ECDSA)

Sign a document using a private key that is unique to you (your hand writing)

Your signature is verified using a public key that is known to anyone (your reference signature)



The World Leader in High Performance Signal Processing Solutions

Lockbox Overview

BLACKfin™

ANALOG DEVICES

Lockbox Introduction

- Lockbox enables security by providing a privileged (secure) mode of operation in which only trusted code is allowed to execute.
- Lockbox performs digital signature authentication
  - For an application to establish trust and to reach the privileged mode of operation, it has to undergo a digital signature authentication as specified in ANSI X9.62 (ECDSA).
  - The digital signature authentication process is carried out by firmware stored in on-chip ROM.
- Lockbox DOES NOT perform encryption or decryption
  - The secure processing environment supports execution of sensitive code and can safeguard the execution of an encryption/decryption algorithm as well as the decrypted 'message' when stored in protected on-chip memory.
- Security features are completely optional

Security Feature Highlights

- OTP memory for storage of customer programmable cipher keys, unique chip ID or a customer ID
- OTP write protection to protect programmed OTP memory locations from future tampering
- Private/Secret OTP memory region accessible only in Secure Mode
  - Store private key(s) for decryption of data or other validation
- Protection of L1 and on-chip L2 regions of memory with access controlled when in Secure Mode.
  - These memory areas are configurable in size and can protect sensitive data from any DMA access.
- A secure mode of operation to perform sensitive decryption or execution of code
- "Secured switches" to disable all avenues of attack in support of a secured environment
  - Disable DMA access to L1 and on-chip L2 memory
  - Disable ADI JTAG emulation from ICE port
  - Divert hardware reset to NMI
- A suitable mode (including on chip ROM) to perform code authentication

Blackfin enhancements for security

List of new hardware features

- **One Time Programmable memory (64Kx1 bits)**
  - **Public OTP memory (4 KBytes)**
    - Used to keep a trusted Public Key for proper authentication
  - **Private (secret) OTP memory (4 KBytes)**
    - Used to keep secrets only accessible in Secure Mode (example: secret keys for a cipher)
  - **Unique Chip ID (stored in Public OTP memory)**
    - Can be used to **prevent cloning of products** (bind software -in a flash memory- to a single Processor)
- **Secure ROM**
    - Used to store the authentication software (Secure Entry Service Routine, crypto)
- **Secure State Machine**
  - **Open Mode (Unsecured)**
    - Default power up mode of the Processor
  - **Secure Entry Mode**
    - Ensures integrity of authentication process
  - **Secure Mode**
    - Secure environment to execute sensitive code and protect data in on-chip memory
- **Hardware Monitor**
  - **Firmware execution is monitored for unexpected branches**
- **System switches (SECURE_SYSSWT)**
  - Controls secure environment and prevents attacks using JTAG, reset pin or DMA memory accesses

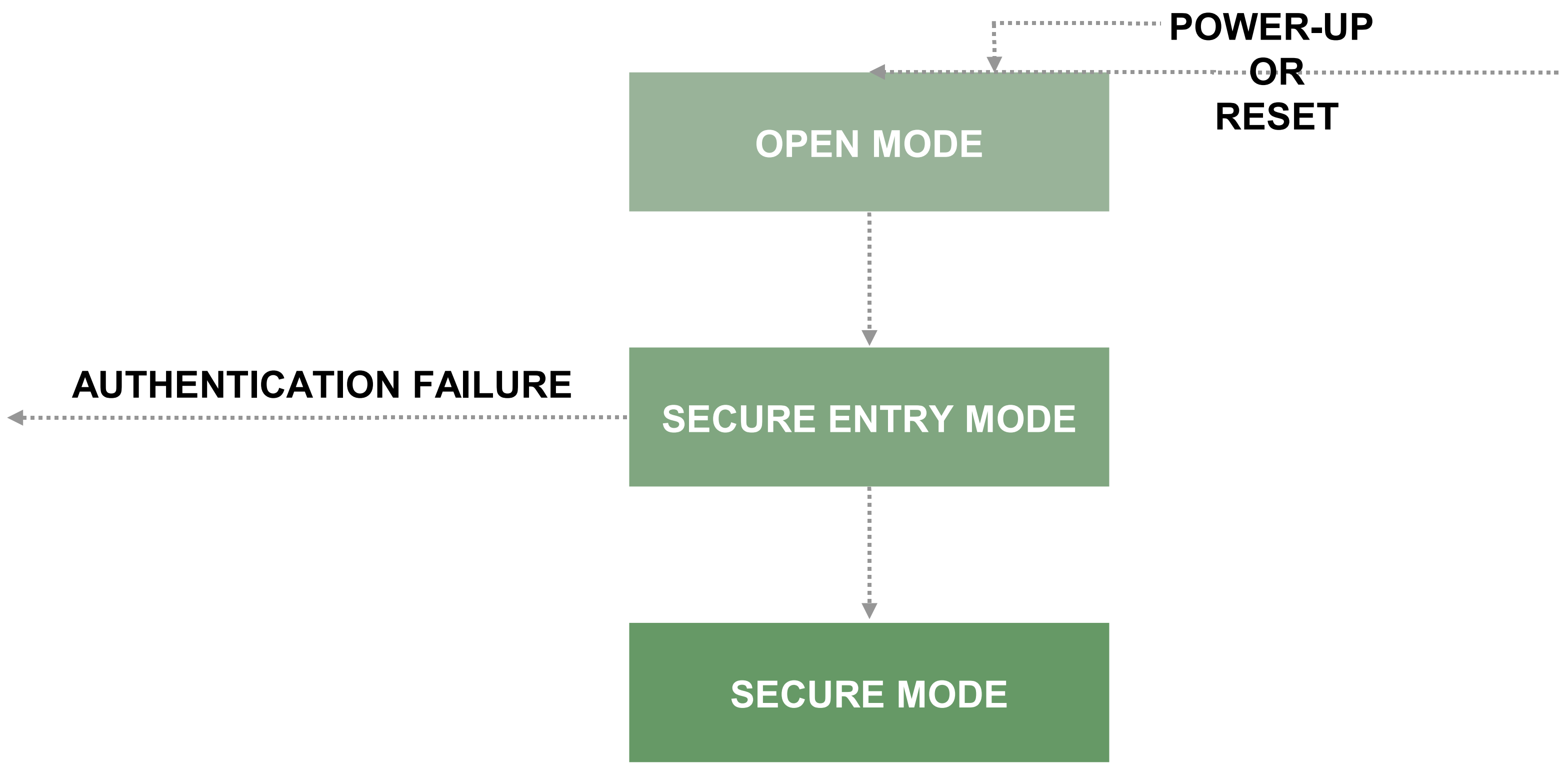
Secure State Machine

Secure State Machine Modes of operation

- **Open Mode (Unsecured)**
  - Default mode of processor upon power up/reset/boot
  - All secured system switches are deactivated.
  - OTP memory secrets are protected from access.
  - The chip is open, all features are available with no restrictions.
- **Secure Entry Mode (Authentication)**
  - Firmware is executing out of internal memory to authenticate a loaded code image
  - All secured system switches are activated.
- **Secure Mode**
  - Once authentication process results in success, device is in Secure Mode
  - Mode of operation to perform sensitive decryption or execution of code
  - OTP memory secrets are accessible.
  - Secured system switches are accessible to user (authenticated) code

ANALOG DEVICES

Secure State Machine



Using Security Features

- **To make use of Blackfin's Secure Lockbox Technology, it is necessary to do the following:**
  1. Generate a key pair
  2. Program the public key in Blackfin's OTP memory (Personalization)
  3. Sign the application that will run in secure mode using the private key
  4. Request Authentication

Generate a Key Pair

- An ECC key pair consists of a Public key and a Private key
  - The Public Key is stored on the Blackfin itself and is used for verifying (authenticating) digital signatures on the Blackfin
  - The Private Key is retained by the developer and kept confidential and is used for digitally signing messages off chip.
- Developers are responsible for key management, i.e., generating keys and programming the public key onto the processor as well as maintaining the confidentiality of their private key.

Public Key Programming (Personalization)

- **Public ECC key must be programmed into specified area within public OTP memory in order to perform Authentication and transition Secure State Machine through state flow.**
  - OTP pages 0x10, 0x11 and 0x12 hold the customer public key
  - Developers are responsible for key management, i.e., generating keys and programming the public key onto the processor as well as maintaining the confidentiality of their private key.
- **OTP can be programmed via JTAG or via loading and executing code on the Blackfin.**
- **Support for OTP programming by Device Programmer vendors such as BP Microsystems and DataIO is currently under development**

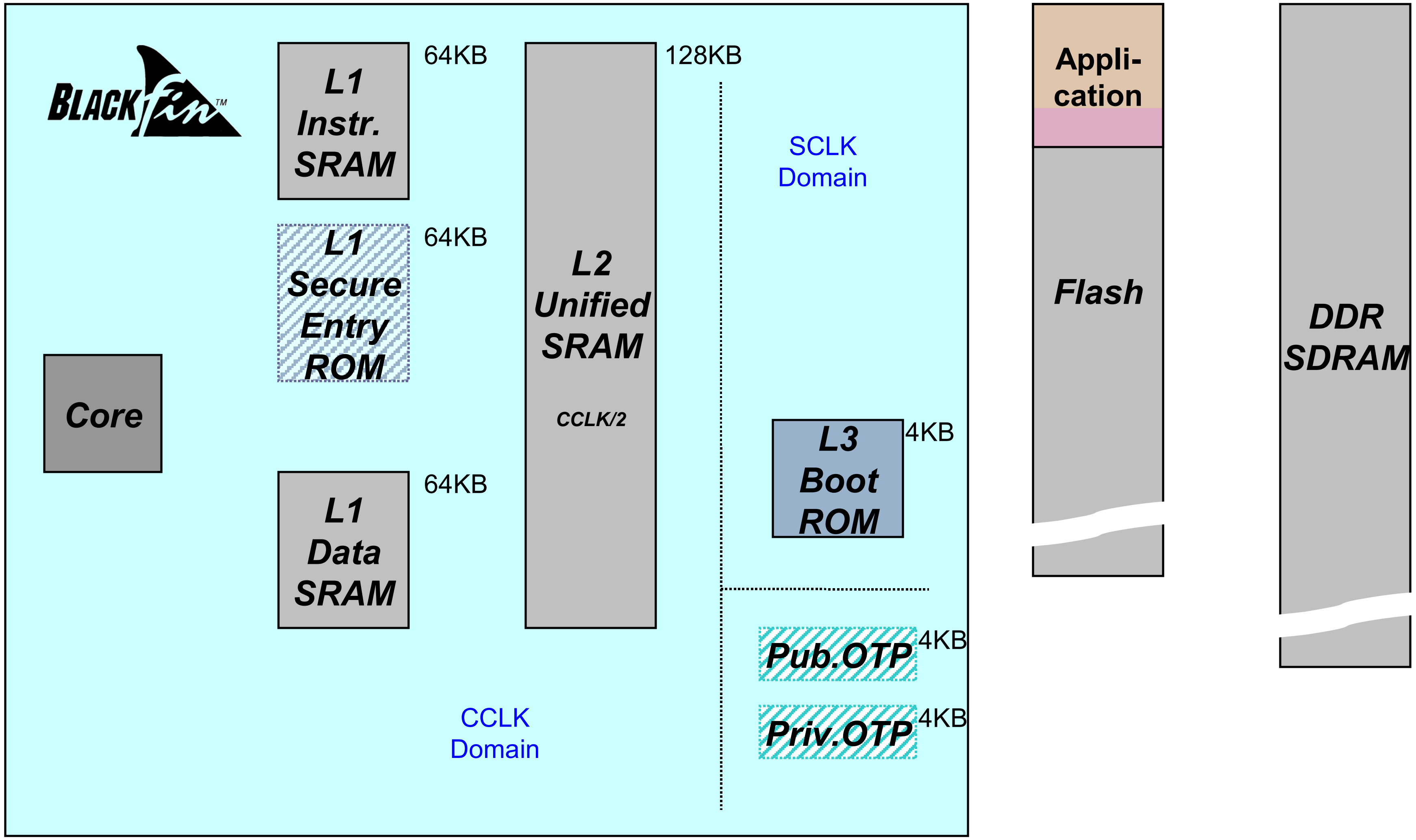
EZ-Kit Security Examples

- A utility is distributed with the VisualDSP++ 5.0 EZ-Kit example code to sign the application that will run in secure mode with publicly disclosed private key.
- WARNING: Please be aware that the digital signature utility provided is for demonstration purposes only. It is not sufficiently secure for practical purposes. Please do not use the provided digital signature utility to generate digital signatures for application in your product.
- BF54x and BF52x EZ-Kits will ship from ADI with a public key pre-programmed in OTP and a published private key for use with example code and to facilitate support of customer development

Initiating Authentication

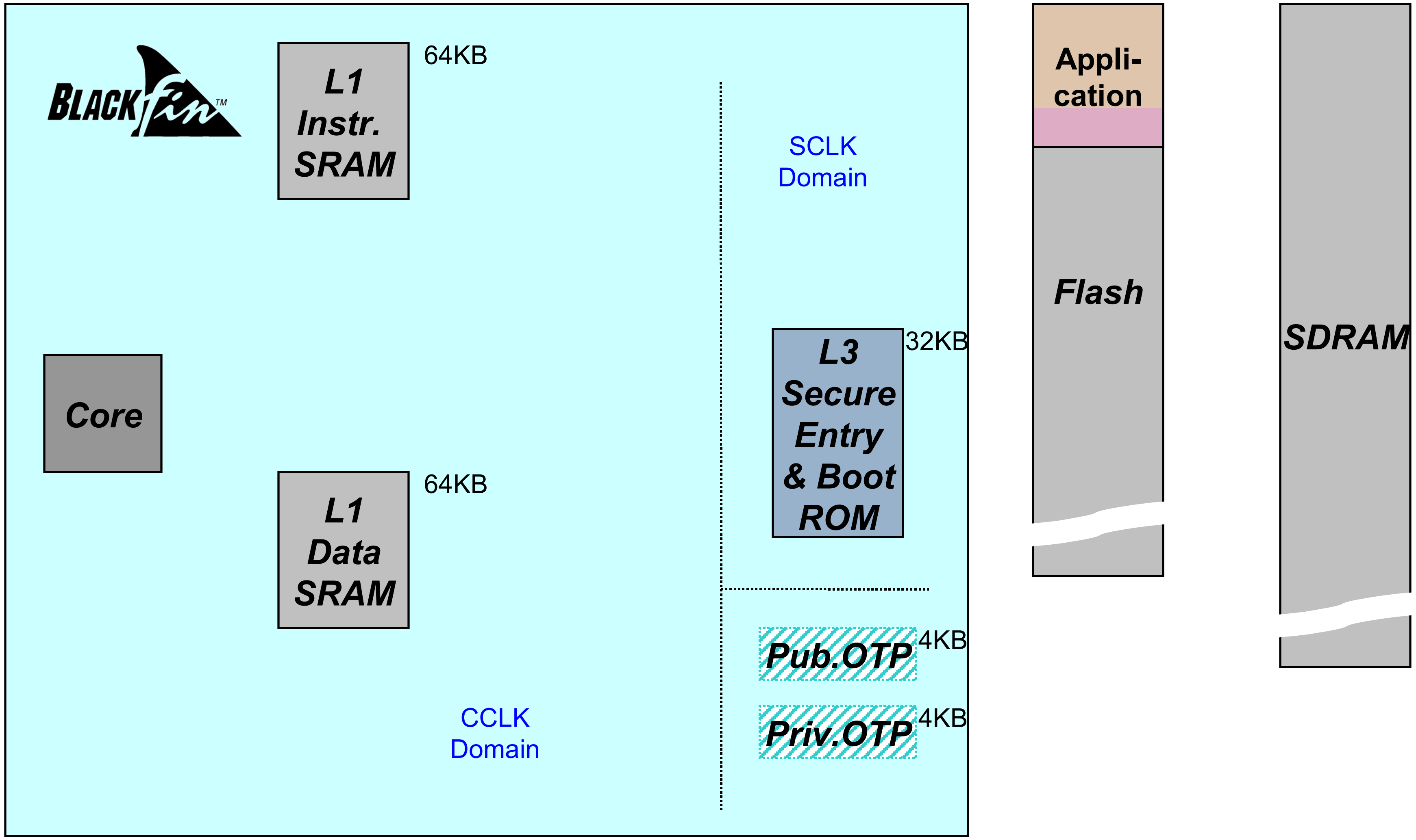
- Two conditions must be met to initiate Authentication and start the Secure Entry Service Routine (SESR) (i.e. authentication firmware),
  - First, the beginning address of the authentication firmware must be loaded into the NMI (EVT2) location of the event vector table.
  - Second, the NMI interrupt must be triggered. For sequential operation, “raise 2;” is executed.
- Since “raise 2;” is only allowed in supervisor mode, the steps to start authentication could be concealed in a system call. The system call can also set up the environment necessary to start the authentication. Specifically, it would set up the arguments and save or move the contents in L1 or L2 memory.
- Since the authentication firmware assumes a specific memory configuration before it begins, the system call also has the responsibility to temporarily move any data out of L1 or L2 regions used by the authentication firmware.

ADSP-BF54x



BLACKfin™
Core
L1 Instr. SRAM
64KB
L1 Secure Entry ROM
64KB
L1 Data SRAM
64KB
L2 Unified SRAM
CCLK/2
128KB
SCLK Domain
L3 Boot ROM
4KB
Pub.OTP
4KB
Priv.OTP
4KB
CCLK Domain
Appli-cation
Flash
DDR SDRAM

ADSP-BF52x



BLACKfin™
L1 Instr. SRAM
64KB
SCLK Domain
Appli-cation
Flash
SDRAM
Core
L3 Secure Entry & Boot ROM
32KB
L1 Data SRAM
64KB
Pub.OTP
4KB
Priv.OTP
4KB
CCLK Domain

Lockbox programming interface

- SESR API
  - Argument Structure
  - Message/Signature Structure
  - Error Return Codes
- Memory Configuration

SESR Argument Structure

The following security firmware arguments are the arguments accepted by the secure entry service routine:

/* SESR argument structure. Expected to reside at 0xFF900018*/

```
typedef struct SESR_args {
  unsigned short  usFlags;           /* security firmware flags */
  unsigned short  usIRQMask;         /* interrupt mask */
  unsigned long   ulMessageSize;     /* message length in bytes */
  unsigned long   ulSFEntryPoint;    /* entry point of secure function */
  unsigned long   ulMessagePtr;      /* pointer to the buffer containing
                                        the digital signature and message  */
  unsigned long   ulReserved1;       /* reserved  */
  unsigned long   ulReserved2;       /* reserved  */
} tSESR_args;
```

Flags Argument (Detail)

The usFlags argument is made up of the following bitfields:
/* SESR flags argument bitfields*/

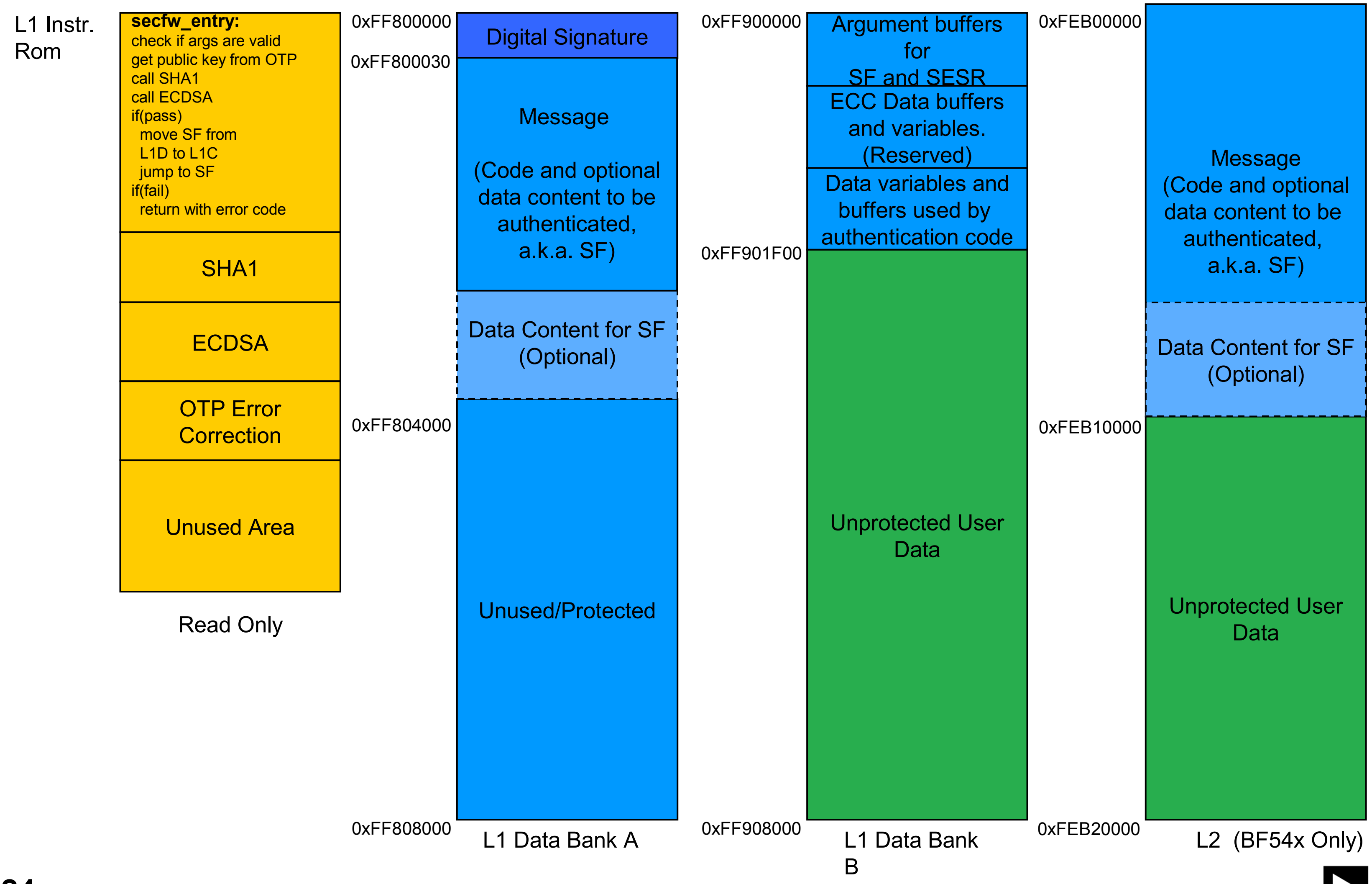
```
#define SESR_FLAGS_STAY_AT_NMI                         0x0000
#define SESR_FLAGS_DROP_BELOW_NMI                      0x0001
#define SESR_FLAGS_NO_SF_DMA                           0x0000
#define SESR_FLAGS_DMA_SF_TO_RUN_DEST                  0x0002
#define SESR_FLAGS_USE_ADI_PUB_KEY                     0x0000
#define SESR_FLAGS_USE_CUST_PUB_KEY                    0x0100
```

Message/Signature Structure

```
The structure of the message:               |-------------------------------|
                                            | Code to receive control from |
                                            | the security firmware (SESR) |
                                            |-------------------------------|
                                            | Code or Data                  |
                                            .                               .
                                            .                               .
                                            .                               .
                                            |-------------------------------|
```

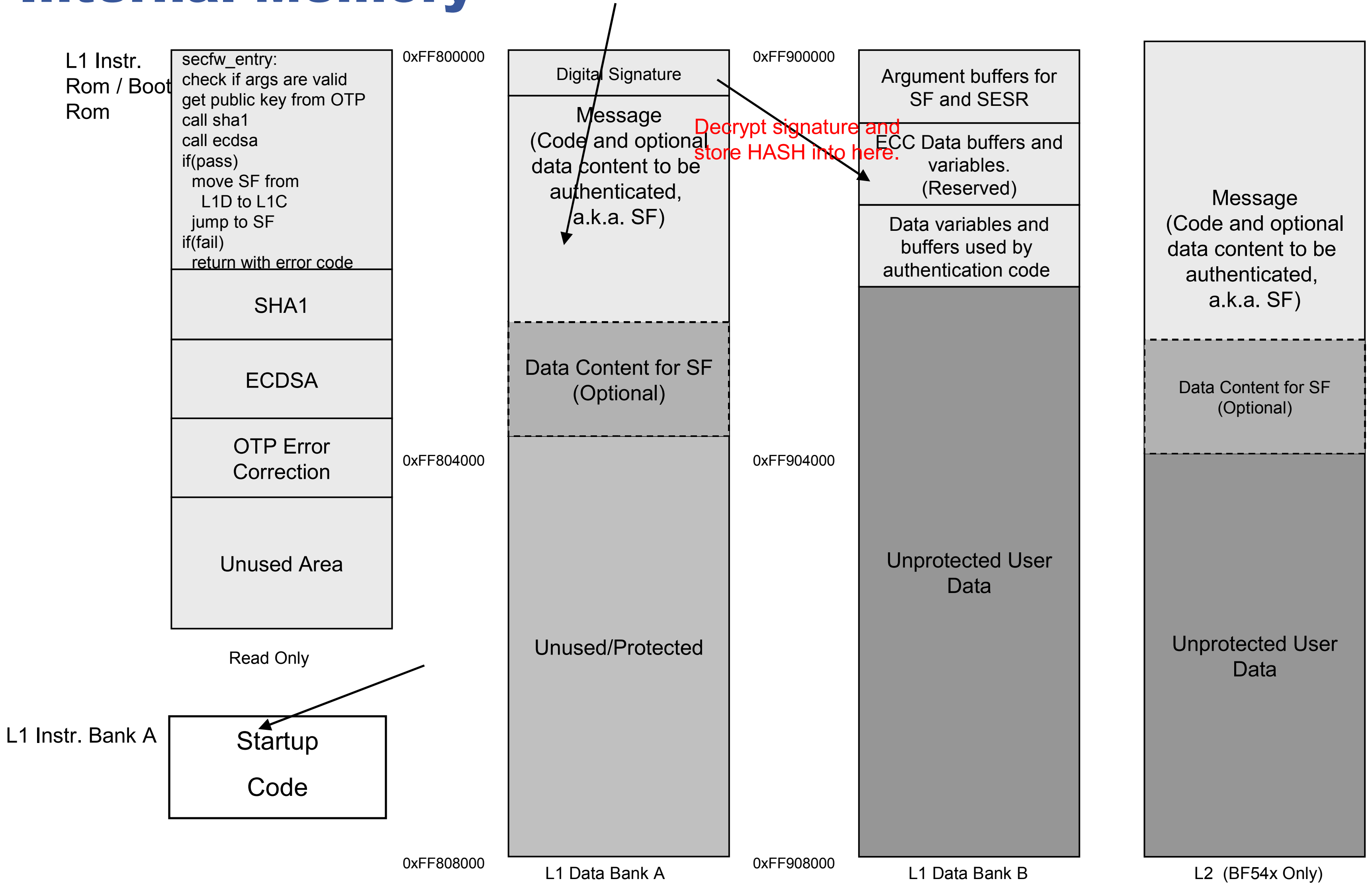
ANALOG DEVICES

Memory Configuration for Authentication



Internal Memory

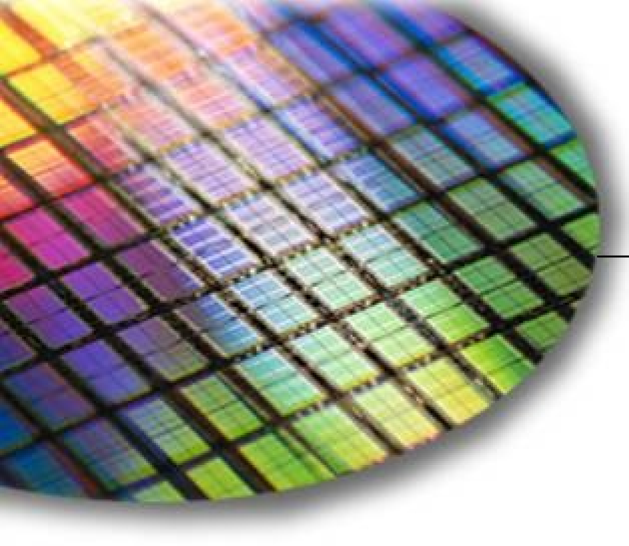
Developer's decryption algorithm will go here



Memory Configuration for Authentication

- The message can either be placed within the protected area of L1A or L2
  - If the message (i.e. code) is put into L1A for authentication, it *must* be DMA'd to either L1 code space or L2, where it can execute. If the message is placed into L2 for authentication, it can remain in L2 and be executed directly from L2
- The digital signature is a pair of 163 bit integers. Each integer is padded to the nearest 32-bit word (resulting in 192 bits). Thus, the total size of the digital signature is 384 bits.
- If the message is placed in L1A data memory, it must immediately follow the digital signature.
- Alternatively, the message can be placed anywhere in the secured region of L2 memory.
- When the Secure State Machine enters into Secure Entry Mode (authentication), certain portions of memory are protected from DMA accesses.
  - These include 32KB of L1A and 8KB of L1B data memory
  - L1 instruction memory (32KB)
  - Half of on-chip L2 memory (64KB on processors with L2).
  - This means that the message/code that needs to be authenticated is 32KB less 48 bytes for the digital signature, if placed in L1A data memory and 64KB less 48 bytes if placed in L2 memory.
- The arguments for both the SESR and the SF are stored in L1B data memory beginning at top (0xff900000).
- 8KB of L1B data memory is reserved for the firmware for scratch working space
  - All memory above 0xff901f00 in L1B is reserved for authentication.

The World Leader in High Performance Signal Processing Solutions

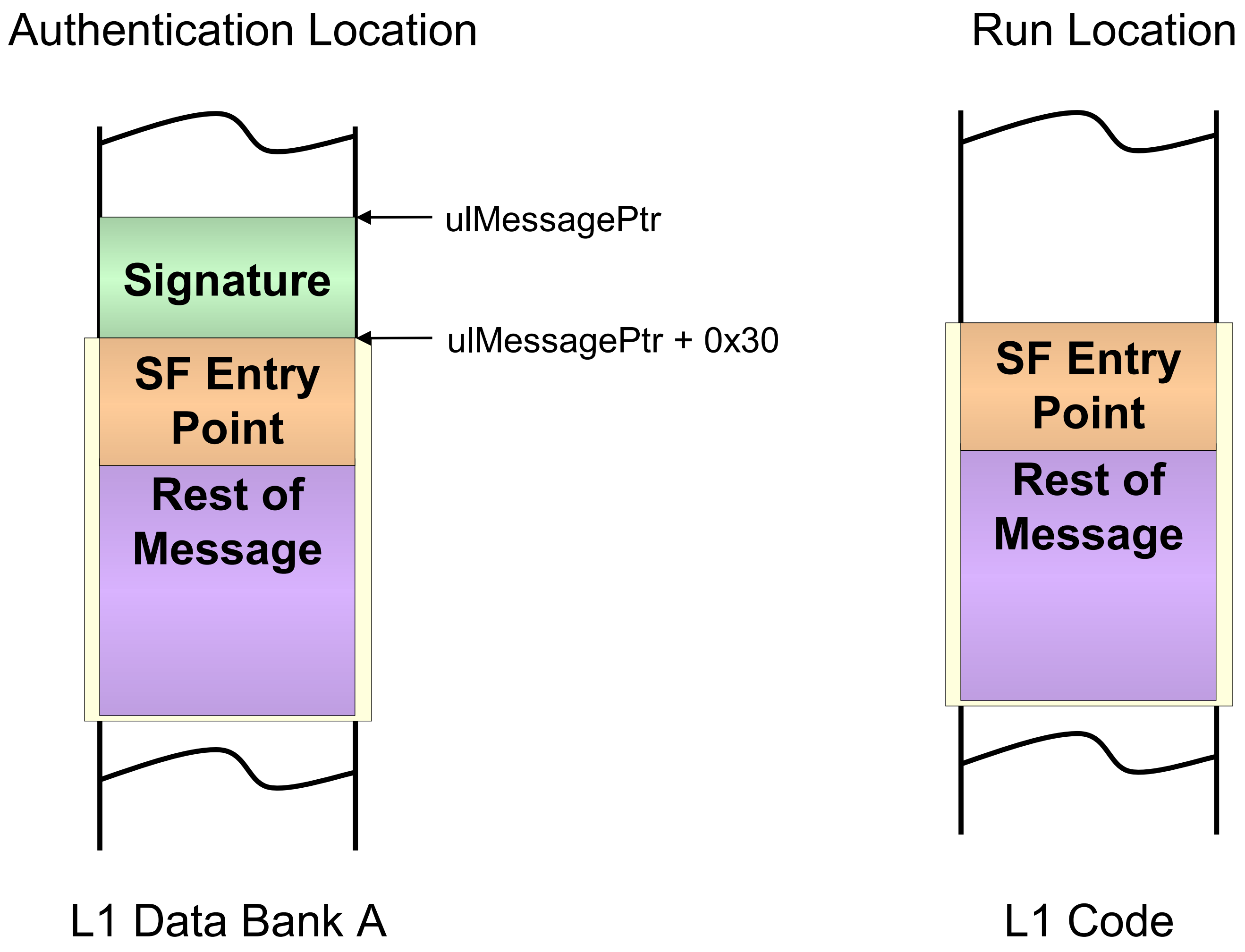


Lockbox Programming

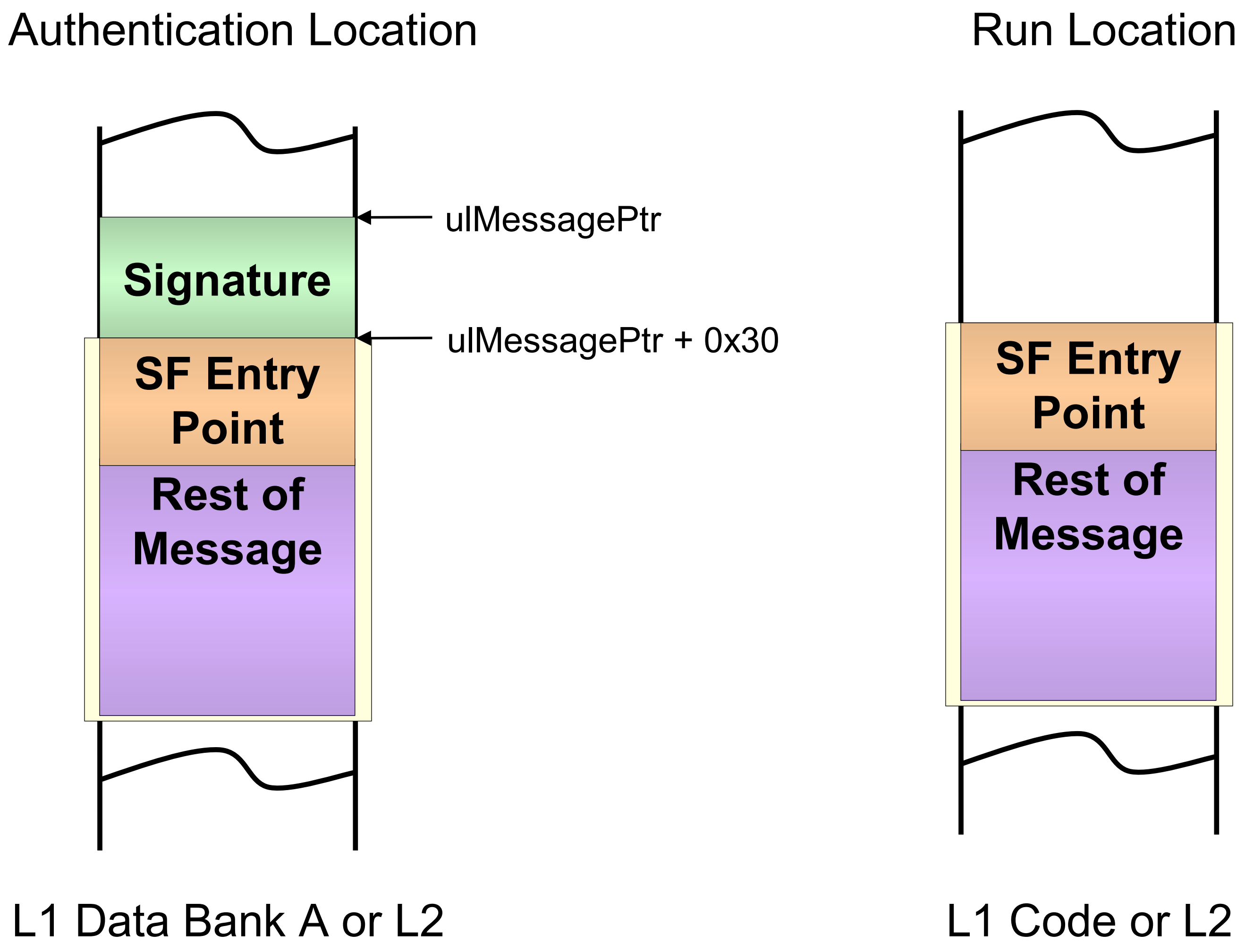
BLACKfin™

ANALOG DEVICES

Message Placement for BF52x Processors



Message Placement for BF54x Processors



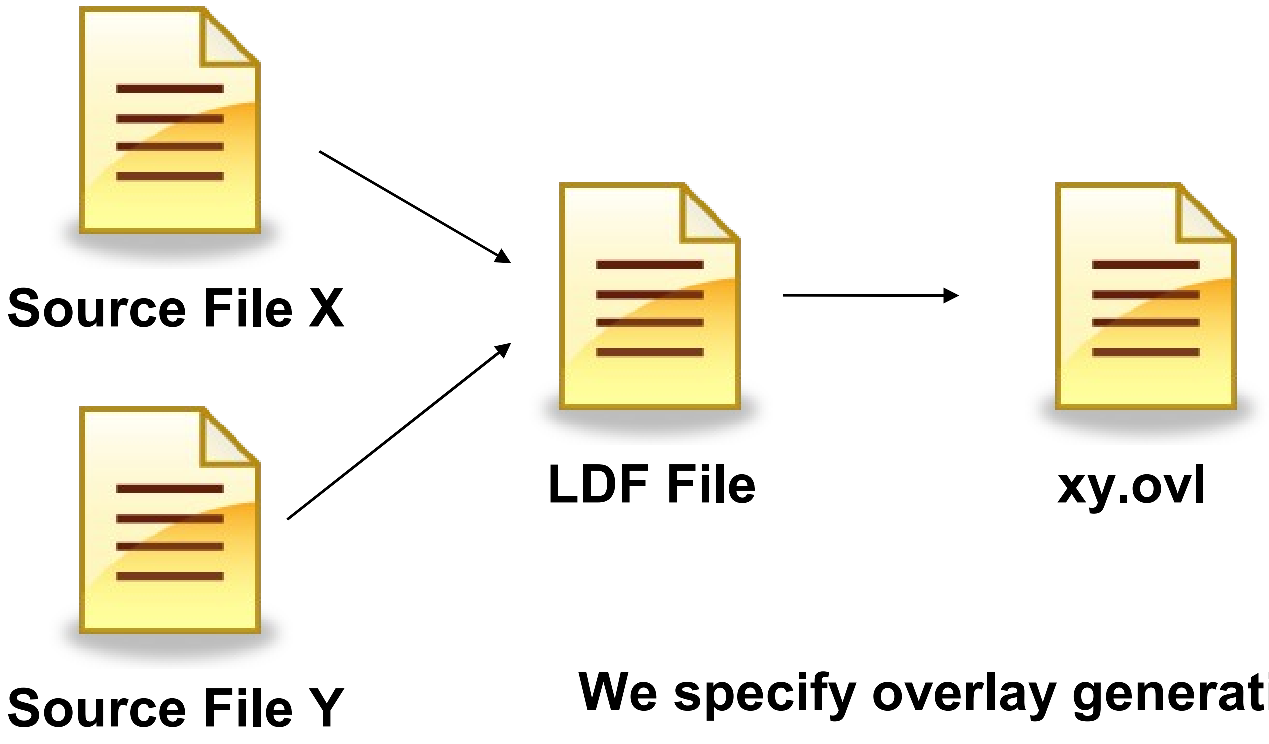
Message Placement – Deep Thoughts

How do I deal with the message?

- During authentication, the message will be in on-chip data memory
- In secure mode, the message will be in on-chip code memory (and possibly data memory as well)
- The message may not be a critical function that you want to reside in internal memory in the first place

The answer is overlays

Life Story of an Overlay – 1



We specify overlay generation in the LDF:

• Take section xx from source file x

•Take section yy from source file y

•Call the output xy.ovl

Message Placement – Source File

```
section("overlay_live_1") void secure_function(void)
{
   /* Enable JTAG */
   *pSECURE_SYSSWT = ( *pSECURE_SYSSWT & ENABLE_JTAG );
   ssync();

   log_authentication_results();

   led_blink();

   return;
}
```

Message Placement – LDF File

```
#define MEM_MESSAGE_OVL_RUN MEM_L1_CODE
#define MEM_MESSAGE_OVL_LIVE MEM_L1_DATA_A


message_overlays {

  ALIGN(4)

  OVERLAY_INPUT {
    INPUT_SECTION_ALIGN(4)
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT( $COMMAND_LINE_OUTPUT_DIRECTORY\secure_function.ovl )
    INPUT_SECTIONS( "secure_function.doj"(overlay_live_1) )
  } > MEM_MESSAGE_OVL_LIVE

} > MEM_MESSAGE_OVL_RUN
```

Digital Signature Placement

- In the source file:

```
section("L1_data_a_ds") u32
 digital_signature_in_L1_data_a[DS_LEN_WORDS];
```

- In the LDF file:

```
digital_signature
{
    FORCE_CONTIGUITY
    INPUT_SECTION_ALIGN(4)
    INPUT_SECTIONS( $OBJECTS(L1_data_a_ds) )
} > MEM_L1_DATA_A
```

Security Firmware Arguments & Scratch Buffer Requirements

- The security firmware expects its arguments to reside towards the beginning of L1 data bank B. (0xFF900018)
- It also expects to have free access to a scratch buffer of size 0x1F00 immediately following the argument buffers

```
L1 data bank B          FF900000    - - - -     |------------------------------|
                                                |                              |
                        FF900018    - - - -     |------------------------------|
                                                | Security Firmware Arguments  |
                                                |         (24 bytes)           |
                        FF900030    - - - -     |------------------------------|
                                                |  Scratch Buffer for Use by   |
                                                |    the Security Firmware     |
                                                |        (7936 bytes)          |
                        FF901F30    - - - -     |------------------------------|
```

ANALOG DEVICES

Security Firmware Arguments Placement

- In the source file:

```
tSESR_args security_firmware_args_in_L1_data_b;
```

- In the LDF file:

```
RESOLVE(_security_firmware_args_in_L1_data_b, 0xFF900018)
```

Dealing with the Scratch Buffer Requirement – Overlay Style

- In the source file:

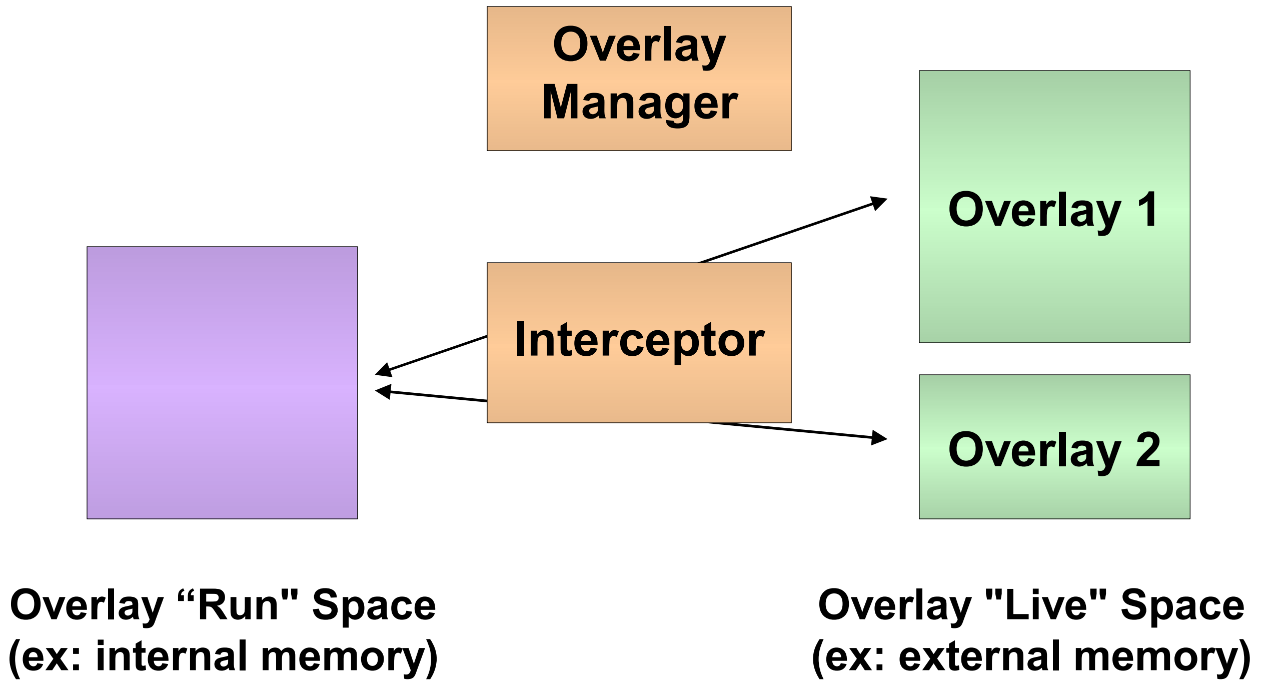
```
section("overlay_live_3") u8
  security_firmware_scratch_in_L1_data_b[SCRATCH_BUFFER_SIZE - 4];
```

- In the LDF file:

```
scratch_overlays {
  ALIGN(4)

  OVERLAY_INPUT {
    INPUT_SECTION_ALIGN(4)
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT($COMMAND_LINE_OUTPUT_DIRECTORY\secure_scratch.ovl )
    INPUT_SECTIONS( "secure_function.doj"(overlay_live_3) )
  } > MEM_SCRATCH_OVL_LIVE

} > MEM_SCRATCH_OVL_RUN
```

Life Story of an Overlay – 2

- Linker adds a unique 4Byte overlay ID to the beginning of each overlay
- Linker replaces all references to functions in an overlay with a custom code (interceptor)



Interceptor (PLIT) – 1

```
PLIT {
  [--SP] = RETS;
  [--SP] = (R7:0,P5:0);
  R0.L = lo(PLIT_SYMBOL_OVERLAYID); // pass overlay ID to overlay manager
  R0.H = hi(PLIT_SYMBOL_OVERLAYID);
  sp += -12;
  CALL (_OverlayManager);
  sp += 12;

  R0.L = lo(PLIT_SYMBOL_OVERLAYID); // are we dealing w/a secure overlay?
  R0.H = hi(PLIT_SYMBOL_OVERLAYID);
  R1 = 2;
  cc = R0 < R1;
  if !cc jump non_secure_overlay_PLIT_SYMBOL_OVERLAYID;

secure_overlay_PLIT_SYMBOL_OVERLAYID:
  …
non_secure_overlay_PLIT_SYMBOL_OVERLAYID:
  …
}
```

Interceptor (PLIT) – 2

```
secure_overlay_PLIT_SYMBOL_OVERLAYID:
  // Load the firmware address into the NMI handler:
  P0.H = (0xFFE02008 >> 16);
  P0.L = (0xFFE02008 & 0xFFFF);
  P1.H = (0xEF001000 >> 16);
  P1.L = (0xEF001000 & 0xFFFF);
  [P0] = P1;
  ssync;

  // restore state
  (R7:0,P5:0) = [SP++];
  RETS = [SP++];

  // Invoke SESR
  raise 2;

  // The stack is preserved accross SESR
  SP += -4;
  P0 = [SP++]; // re-read RETS
  jump (P0);
```

Interceptor (PLIT) – 3

```
non_secure_overlay_PLIT_SYMBOL_OVERLAYID:

  // restore state
  (R7:0,P5:0) = [SP++];
  RETS = [SP++];

  P0.L = lo(PLIT_SYMBOL_ADDRESS);
  P0.H = hi(PLIT_SYMBOL_ADDRESS);

  // "call" to resolved symbol
  JUMP (P0);
```

Overlay Manager – 1

```
/* Element type for a data overlay linked list */
typedef struct data_overlay_pointer_struct {
  s32                                      dataOvlIndex[DATA_OVL_INDEX_SIZE];
   struct data_overlay_pointer_struct*    next;
} t_data_ovl_ptr_struct;



/* overlay table layout */
typedef struct overlay_struct {
  u32 liveAddress;                           /* RAM-based live addresses */
  u32 liveSize;                              /* live size */
  u32 runAddress;                            /* run address */
  u32 runSize;                               /* run size */
  u32 flags;                                 /* flag to indicate whether or not we
                                                              are processing a
   secure overlay */
  t_data_ovl_ptr_struct * dataOverlays;     /* pointer to a linked list of
                                                related data overlays */
} t_overlay_struct;
```

Overlay Manager – 2

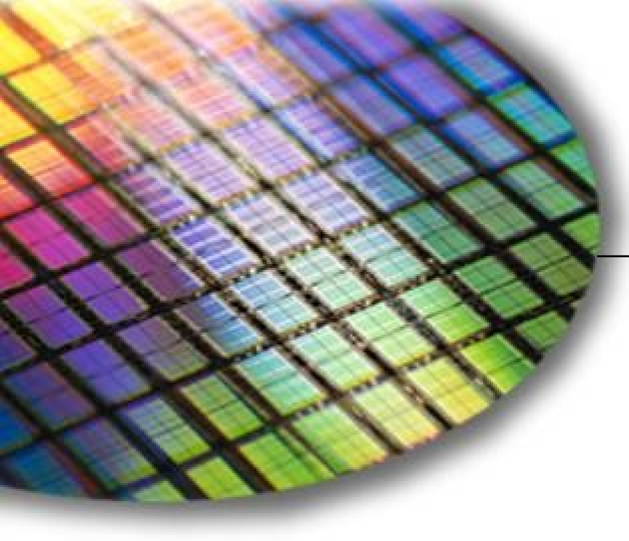
```
void OverlayManager(s32 requestID)
{
  requestIndex = requestID - 1;  // zero-based index

  if ( DATA_OVERLAY != overlay_table[requestIndex].flags ) {
    /* Bring in any associated data overlays */
  }

  /* For secure overlays, the overlay manager doesn't DMA the code
   into
     run space because the security firmware does it. */
  if ( SECURE_OVERLAY == overlay_table[requestIndex].flags )
    return;

  /* Check is overlay is already loaded */
  if (residentID == requestID)
    return;

  /* DMA in the overlay code */
  …
}
```

Overlay Manager – 3

- If the overlay is a secure overlay, the overlay manager does not need to perform the DMA because the firmware code will move the message from its authentication location to its run location.
- If the overlay is a secure overlay, the SESR (firmware) will require some scratch space. One way to implement this is to associate a data overlay with the secure overlay. Every time the overlay manager encounters a secure overlay, it checks to see if any data in the scratch space needs to be saved before it loads in the secure overlay.

Signing a Message

```
cmd /C echo Delete previous state
cmd /C del Debug\secure_function.bin 1>nul 2>&1
cmd /C del Debug\secure_function_signature.txt 1>nul 2>&1
cmd /C echo Extract the overlay section
cmd /C $(VDSP)\elfpatch.exe -get _ov_message_overlays_1 -o Debug\secure_function.bin
   Debug\secure_function.ovl(overlay1.elf)
cmd /C echo sign the contents of the overlay section
cmd /C cd ..\Bin &&
       ecsign.exe ..\ADSP-BF527_EZ-KIT_Lite\elliptic_curve_parameters_527.ecs
                  ..\ADSP-BF527_EZ-KIT_Lite\private_key_527.ecs
                  ..\ADSP-BF527_EZ-KIT_Lite\random_number_527.ecs
                  ..\ADSP-BF527_EZ-KIT_Lite\Debug\secure_function.bin &&
       cd ..\ADSP-BF527_EZ-Kit_Lite
cmd /C echo Place the digital signature in the digital_signature section
cmd /C $(VDSP)\elfpatch.exe -replace digital_signature -bits Debug\secure_function.bf
   -text Debug\lockbox_527.dxe
```

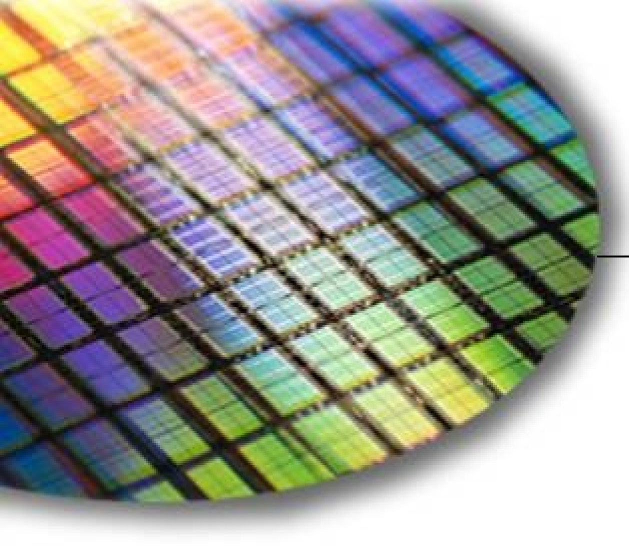


The World Leader in High Performance Signal Processing Solutions

Instructor led demo

BLACKfin™

ANALOG DEVICES



The World Leader in High Performance Signal Processing Solutions

Resources & References

BLACKfin™

ANALOG DEVICES

Resources & References

- http://www.rsasecurity.com/rsalabs/
- http://www.certicom.com/index.php
- http://csrc.nist.gov/
- http://www.cryptnet.net/fdp/crypto/crypto-dict.html
- http://www.keylength.com/index.php
- http://www.schneier.com/index.html
- http://www.kilopass.com/e15/

- Textbook references:
  - Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, by Bruce Schneier, Wiley; 2 edition (October 19, 1995), ISBN: 0471128457
  - Security Engineering: A Guide to Building Dependable Distributed Systems, by Ross Anderson, Wiley (January 22, 2001), ISBN: 0471389226
  - Handbook of Applied Cryptography, by A. Menezes, P. van Oorschot, and S. Vanstone, CRC Press, 1996. For further information, see www.cacr.math.uwaterloo.ca/hac

Cryptography Made Easy

- **An Illustrated Guide to Cryptographic Hashes**
  http://www.unixwiz.net/techtips/iguide-crypto-hashes.html
- ***Digital Signature GuidelinesTutorial***
  http://www.abanet.org/scitech/ec/isc/dsg-tutorial.html
- **What is a Digital Signature?**
  http://www.youdzone.com/signature.html
- **Cryptography Dictionary**
  http://www.cryptnet.net/fdp/crypto/crypto-dict.html
- **Wikipedia, the free encyclopedia**
  http://www.wikipedia.org/

Glossary

- Asymmetric algorithm - A cryptographic algorithm which uses two different keys for encryption and decryption.
- Authentication - Verifying a code image against its embedded digital signature. Process for identifying either entities or data origins
- Authentication Control Code - Firmware code stored in ROM to control Secure State Machine and hardware modes during the process of Authentication
- Chip ID - Unique identification number per chip (stored in public OTP memory)
- Ciphertext - Encrypted message
- Cleartext - Unencrypted message (synonomous with ”plaintext”)
- Confidentiality - Cryptographic means to ensure privacy or secrecy of information from unauthorized parties (so that only after authorized access, data can be read). Typically, confidentiality is ensured using data encryption via symmetric algorithms.
- Digest - Secure digital fingerprint, created by a one-way hashing function
- Digital Certificate - A piece of information digitally signed by a trusted third party, or certificate authority (CA), that establishes a user's credentials and identity. Typically consists of customer's public key and customer ID signed by a certification authority
- Digital Signature - A digitally signed hash result of the message. Any digest encrypted with a customer's private key
- Elliptic Curve Cryptography [ECC] - A class of cryptosystems that are based on the difficulty of finding points on an elliptic curve over a field with special properties. Most often, the strength of ECC is provided by the discrete logarithm problem; however, the factoring problem can also be used.
- Integrity - Cryptographic means to ensure that the message or the content of the storage media has not been altered in any way. Integrity is verified using authentication.
- Key management - Refers to the distribution, authentication, and handling of keys
- Non-repudiation - Preventing an entity from denying previous commitments or actions

ANALOG DEVICES

Glossary

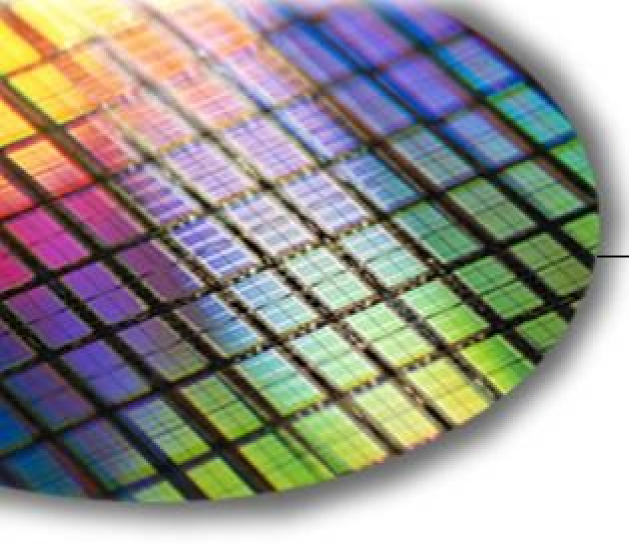
- **Open Mode** - Default operating mode of the processor in which nothing is restricted except for access to Private OTP memory.
- **OTP** - One Time Programmable memory. Customer-programmable via code executing on processor
- **Plaintext** - Unencrypted message
- **Private Key** - Private (secret) part of asymmetric key used to create digital signatures
- **Private OTP** - Customer-programmable OTP memory area for private (secret) key and sensitive information storage. Accessible *only* in Secure Mode.
- **Public Key** - Public part of asymmetric key used to verify digital signatures
- **Public OTP** - Customer-programmable OTP memory area for public key and information storage. Accessible in all operating Modes including Open Mode, Secure Entry Mode and Secure Mode.
- **Secret Key** - Any symmetric secret key (e.g., download key, key encryption key (KEK))
- **Secure Entry Mode** - Secure operating mode in which firmware controls the authentication process
- **Secure Mode** - Secure operating mode allowing execution of authenticated code, decryption of sensitive information, authenticated code access chip secrets in private OTP memory area, etc.
- **Secure RAM** - Configurable part of internal system RAM accessible only in Secure Mode
- **Security Framework** - Application code executed in RAM to pass parameters to Authentication Control Code and invoke an Authentication Request
- **Symmetric Key algorithm** - A cryptographic algorithm in which only one key is used to both encrypt and decrypt data.

References:
Cryptography Dictionary (http://www.cryptnet.net/fdp/crypto/crypto-dict.html)
Wikipedia (http://www.wikipedia.org/)

Acronyms

- AES - Advanced Encryption Standard specified in FIPS 197.
- ANSI - American National Standards Institute
- CA - Certification Authority
- DSA - Digital Signature Algorithm specified in FIPS 186-2.
- ECDSA - Elliptic Curve Digital Signature Algorithm
- FIPS - Federal Information Processing Standard.
- HMAC - Keyed-Hash Message Authentication Code spec’d in FIPS 198.
- IV - Initialization Vector.
- MAC - Message Authentication Code
- NIST - National Institute of Standards and Technology
- PKI - Public Key Infrastructure
- PRNG - Pseudorandom Number Generator
- RNG - Random Number Generator
- TDES - Triple Data Encryption Standard; Triple DES

The World Leader in High Performance Signal Processing Solutions



Q & A