CANcryptFD NXP LPC54618
Software Manual

for software of 6-AUG-2018

A manual from

Jointly published by

Embedded Systems Academy, Inc.
1250 Oakmead Parkway, Suite 210
Sunnyvale, CA 94085, USA

Embedded Systems Academy GmbH
Bahnhofstraße 17
30890 Barsinghausen, Germany

Limitation of Liability

Neither Embedded Systems Academy (ESA) nor its authorized dealer(s) shall be liable for any defect, indirect, incidental, special, or consequential damages, whether in an action in contract or tort (including negligence and strict liability), such as, but not limited to, loss of anticipated profits or benefits resulting from the use of the information or software provided in this book or any breach of any warranty, even if ESA or its authorized dealer(s) has been advised of the possibilities of such damages.

The information presented in this book is believed to be accurate. Responsibility for errors, omission of information, or consequences resulting from the use of this information cannot be assumed by ESA. ESA retains all rights to make changes to this book or software associated with it at any time without notice.

1 Contents

2 Introduction

2.1 About this manual

This manual focus is on the CANcryptFD software implementation as provided by Embedded Systems Academy for the NXP LPC54618 microcontroller.

It does not contain detail documentation of the CANcrypt protocol and mechanism. For more information on these, see the book “Implementing scalable CAN Security with CANcrypt” ISBN 978-0-9987454-0-4 or ISBN 978-0-9987454-1-1.

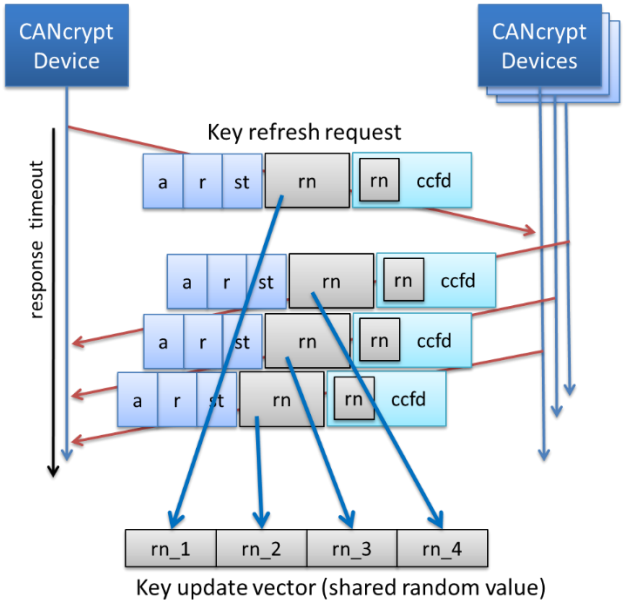
2.2 CANcrypt(FD) features provided

CANcryptFD primary functionality is the CANcrypt grouping mode with the secure heartbeat and dynamic key update mechanism for a maximum of up to 16 participants. More than 16 devices can use CANcryptFD, however, only up to 16 devices can actively participate in the secure heartbeat and key update.

All devices can participate in secure messaging, if they follow (read-only) the secure heartbeat and key updates.

2.3 Powering up

On power up, the devices actively participating in the dynamic key generation process start a secure grouping cycle, as in the classical CANcrypt grouping. All nodes that wish to participate in secure communications must monitor the key generation processes to maintain a local copy of the dynamic key. This cycle is illustrated in the figure below. All participating devices exchange random values which are used as an initialization vector for generating the next key from a previous or known key.



GENERATING AN INITIALIZATION VECTOR FOR KEY GENERATION

Once the dynamic key is generated, all devices (also those not actively participating in the key generation) may start using it by transmitting secure CAN FD messages.

2.4 Initial key generation

CANcryptFD uses a new method for the initial grouping and key generation. The initial grouping cycle for key generation is based on a default key. Then a total of 3 secure heartbeat cycles are executed, each re-generating the dynamic key. After the three key re-generation cycles, the current shared dynamic key is saved as a permanent initial key to the grouped devices.

2.5 Project file structure

The software modules are split into the following directories:

Cc_CANCrypt

This directory contains all source files implementing the core functionality of CANcryptFD. Include them all into your project.

We recommend to not make changes to any of these if you want to be able to easily install future security updates.

Cc_user

This directory contains the user customizable files. Here you can modify and adopt some of the key security functions to your special security requirements.

source

This directory contains the CANcryptFD demo implementation. Where keys are hard coded, simple patterns have been used to easily recognize keys when debugging.

For real implementations do not use default keys and do not use any recognizable patterns. Preferably keys are generated based on true random numbers.

3 Definitions and structures

3.1 Common CANcryptFD parameters

In this section we describe the parameters required to maintain CANcryptFD.

3.1.1 Device numbering and addressing

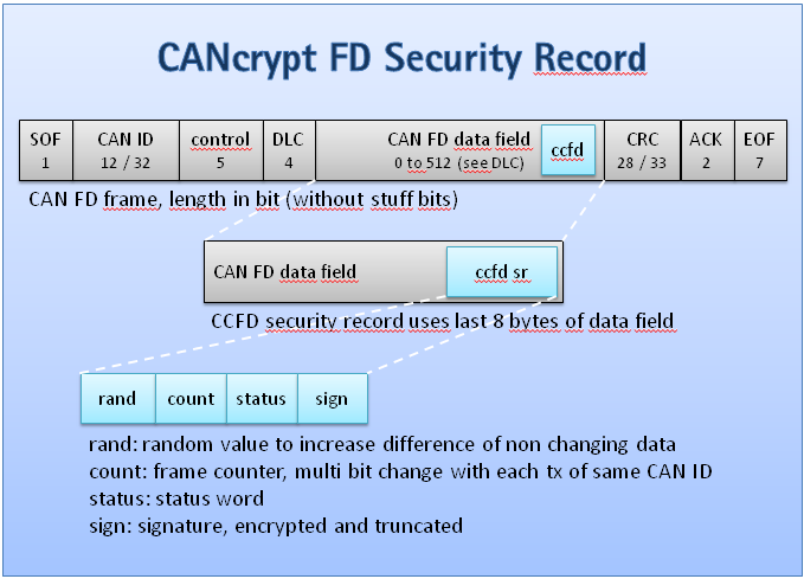
Address, Cc_DEVICE_ID

In all CANcryptFD request or command messages, an address value from 1-127 is used to the target a specific CANcryptFD device. A value of zero broadcasts to all devices (for example, used by the identify request).

To simplify code optimizations, the addresses should be assigned incrementally starting with 1. In the CANcryptFD implementation, a parameter can be set to the "highest address used". If this is set to a value below 14, CANcryptFD devices using an address higher than that value must not be used.

3.1.2 The security record and the digital signature

Each secure CAN FD message has a security record embedded at the end of the data field. The figure below shows the CAN FD frame with the location of the security record.

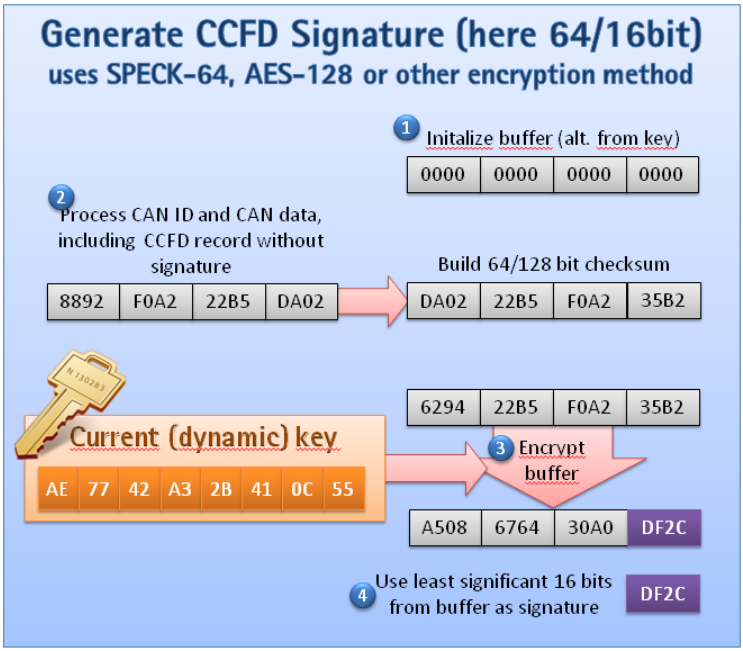


THE CANCRYPT FD SECURITY RECORD

The default security record consists of four 16-bit values:

- Random data: increases entropy and decreases predictability of content
- Frame counter: to guard against re-play attacks, increases on transmit of frame with same CAN ID, increment value is such, that four or more bits change occur per count
- Status word: padding info, current key identification
- Signature: derived from a 64 or 128-bit checksum, encrypted, truncated to 16 bits

As popular secure hash digests like SHA-256 are quite big, CANcrypt FD uses an encrypted checksum, truncated as the digital signature to authenticate the message. The figure below illustrates how it is generated (here, 16-bit signature and 64-bit key).



GENERATING THE DIGITAL SIGNATURE

First, a buffer the size of the key is initialized. Instead of an all-zero initialization, it can also be based on another shared secret. For example, if the shared secret key is larger than required (e.g. 128-bit key, but using 64-bit encryption method), then the key could be split, one half used as main key, the other half as checksum initializer.

Second step, a checksum is build. The number of bits the checksum uses is equal to the buffer width (64 or 128). It covers the CAN ID, the data field before the security record and the security record without the signature.

Third step, the buffer gets encrypted using the current dynamic key.

Forth step, the checksum is truncated and the least-significant 16 bits of the buffer are used as the signature.

3.1.3 The Keys

CANcryptFD supports a number of permanent keys. This allows having multiple keys per device, such as a manufacturer key for bootloader access, a system key (created upon first startup of a CAN system), or further application-specific keys or session-limited keys. For any key stored in non-volatile memory, the size is in the range 128 –1024 bits.

The main keys used are the dynamic key and the permanent key. The permanent key is the non-volatile stored key used for the initialization of the current secure communication. The dynamic key is initialized from that permanent key (a direct copy or generated using a common mixup function) and continuously modified either based on the random bit-select cycles or via the bit-update request.

The last session key can store the dynamic key over a power cycle. If there is a proper shut down procedure before power down, the dynamic key can be saved as the last session key. On the next power up, the key is reloaded to the dynamic key, drastically shortening the initialization phase.

To globally identify the keys, CANcryptFD uses 8-bit Key ID and Key length parameters. These values are used as described below.

Key ID

The Key ID is divided into a 3-bit major value and a 5-bit minor value.

The major value specifies one of eight key types and directly implements a key hierarchy. Higher values have a higher authority. The key erase command can be used only on keys that have the same or lower major value as the key currently in use.

The minor value plus specifies 32-bit segments within the key.

The key length value determines, if a key is used by itself without modifications or gets combined (mixed up) with the local serial number.

The values are mapped to UNSIGNED8 values. The major part uses the three most significant bits, and the minor part uses the five least significant bits.

Default use	Memory	Key ID major	Key ID minor	Length (bit)
Reserved		7		
Manufacturer key	NVOL	6	0–31	128–1024
System Integration key	NVOL	5	0–31	128–1024
Owner key	NVOL	4	0–31	128–1024
User key	NVOL	3	0–31	128–1024
Last group session key	NVOL	2	0–15	128–512
Dynamic pair session key	RAM	1	0–15	128–512
Dynamic group session key	RAM	0	0–15	128–512

THE KEY HIERARCHY

Key length

The Key Length is of type UNSIGNED8. To support a wide variety of key lengths with 8-bit encoding, the highest bit determines if the size is specified in bits or in other units as shown in the table below (Key Length Values Supported by CAN-cryptFD).

Value	Interpretation
00h	Reserved
01h–20h	Key length in bits, 1–32
21h–7Fh	Reserved
80h	Single bit of dynamic key
81h–A0h	Key length in multiples of 32 bits, 1–32 (32–1024 bits)
A1h–C0h	As above, but key is combined with serial number
C1h–FFh	Custom, manufacturer specific sizes

KEY LENGTH VALUES SUPPORTED BY CANCRYPTFD

3.1.4 Status

This section describes the status information that must be provided by all participating CANcryptFD communication partners.

Status

The CANcryptFD status byte provides the following information and is the same for both the CANcryptFD devices:

- Bits 0–1: Pairing status, unused
- Bits 2–3: Grouping status
  0: not grouped
  1: grouping in progress
  2: grouped, secure heartbeat enabled
  3: grouping error
- Bits 4–5: Result of last command or request
  0: unknown
  1: success
  2: ignored
  3: failure
- Bit 6: Reserved
- Bit 7: Key generation in progress
  When set, this device is participating in key generation

3.1.5 Controls

This section describes the control commands and requests available to the CANcryptFD devices.

Request and commands

The 4-bit request value is used in most CANcryptFD protocols.

Message	Type	Consumer Address	Request
Abort	event, response	1–15	0
Acknowledge	response	1–15	1
Alert	event	0	2
Identify	event	0	3

Pairing/Grouping	request, response	1–15	4
Unpairing/Ungroup	request, response	1–15	5
Secure heartbeat	Event	0	9
Save last session key	event	0	15

REQUESTS USED BY CANCRYPTFD DEVICES

3.1.6 Methods

CANcryptFD supports a variety of algorithms and features. The parameters selecting these are listed below. For more details about the specific algorithms used, see chapter 6, CANcryptFD customizable functions.

Method

The 4-bit method parameter selects the base algorithm used to generate the random bit and specifies a security method.

- Bits 0–1: Security functionality
  0: Basic security
  1: Regular security
  2: Advanced Security
  3: Custom security

The security settings influence the bit-generation cycle, authentication, and encryption.

Authentication:
The signature used for messages is 16 bits. The signature is generated by the combination of a checksum that is encrypted and truncated, in advanced mode by AES-128.

Encryption:
The encryption is based on a mixup of the current dynamic key or AES-128 in advanced mode.

3.1.7 Functionality

Individual CANcryptFD functionality may be enabled or disabled.

Functionality

If a corresponding bit is set, the functionality is enabled

- Bit 0: authentication used
- Bit 1: encryption used
- Bits 2–3: reserved

3.1.8 Timings

CANcryptFD uses various timings and timeouts. To minimize the number of definitions, specific values are defined as a group.

Timeout

The 4-bit timeout value defines the timing and timeout options CANcryptFD uses:

- Bits 0–1: timing used
  0: fast
  1: medium
  2: slow
  3: custom timing
- Bits 2–3: reserved

Values 0–2 activate the defined timings in the table below, Timeouts Used by CANcryptFD). Value 3 selects custom, manufacturer-specific timings.

CANcyrpt message timeout:
If a CANcryptFD message contains a request, requiring a response, then the transmitter uses this timeout to wait for a response from the device addressed. If no response is received within this time, the transmitter internally marks the addressed device as not present.

Secure message timeout:
Every secure message combination using a preamble and one or multiple following data messages have to transmit the messages back to back on the network. On the receiving side the data message is only considered to be received in time, if the time since reception of the preamble does not exceed this timeout.

Timeouts	Fast	Medium	Slow
CANcryptFD message timeout (request to response)	100 ms	200 ms	400 ms
Secure message timeout (preamble to message)	25 ms	50 ms	100 ms
Secure heartbeat event time	250 ms	500 ms	1000 ms

(slowest repetition)			
Secure heartbeat event timeout	500 ms	1 s	2 s
Secure heartbeat inhibit time (fastest repetition)	50 ms	100 ms	250 ms
Secure heartbeat cycle timeout	75 ms	150 ms	333 ms
Bit select cycle time for random delay method	25 ms	50 ms	100 ms
Bit select cycle time for direct response method with no delay	10 ms	25 ms	50 ms
Bit select cycle random delay window	0–16 ms	0–32 ms	0–64 ms

DEFAULT TIMEOUTS USED BY CANCRYPTFD

Secure heartbeat event time and timeout:
The longest possible duration between two secure heartbeat cycles is defined by the event time. A device is considered unsecure or missing if the time since the last secure heartbeat transmission exceeds the timeout.

Secure heartbeat inhibit time and cycle timeout:
The shortest possible duration between two secure heartbeat cycles is defined by the inhibit time. All devices may start a new secure heartbeat cycle at any time, as long as they ensure that the inhibit time is met. If a secure heartbeat cycle started, than all active devices must join the cycle with their own secure heartbeat within the cycle timeout. A device not participating in time is considered unsecure or missing.

Bit select cycle time and delay window:
The key- or bit-generation cycle time is a fixed value, the CANcryptFD system tries to determine one bit per cycle. If the method with delays is used (each participant transmits their claim message randomly within a time window), then the maximum value for this delay is defined.

3.1.9 CANcryptFD error counter

CAN uses transmit and receive error counters to determine the “health” of an individual CAN controller. When errors occur, the timers are incremented by a number greater than 1. However, the timers are also decremented when com-

munication works fine. As a result, occasional errors are ignored. But if the counters keep increasing and hit limits, the CAN controller goes “passive” or eventually “bus off,” which is a complete disconnection of the CAN controller from the network.

Event	Counter change
Successful reception of a secure message or secure heartbeat (no timeout, successful authentication)	If counter > 0, decrement
Secure Heartbeat failure (timeout or authentication failure)	Set counter to 128
Intruder alert (injected message with harmful data detected)	Set counter to 128
Intruder alert (injected message with harmless data detected)	Increment counter by 63
Secure message authentication failure	Increment counter by 63
Other error in secure message (preamble timeout, receive without preamble)	Increment counter by 31
Repetitive request from same device to re-initialize the dynamic key	Increment counter by 31
Any other alert event, CAN errors	Increment counter by 15

CANCRYPTFD ERROR COUNTER CHANGES

In the same manner, CANcryptFD uses an UNSIGNED8 error counter to determine the health of the CANcryptFD connection. The table below (CANcryptFD Error Counter Changes) shows which events influence the error counter. Once the error counter reaches 128 or higher, the CANcryptFD device unpairs, or disconnects itself, from secure communication and uses the unpair protocol/status to inform all other devices.

Once a device unpairs itself, it should not be allowed to immediately participate in a re-pairing process. A generous timeout should be required before a retry of the re-pairing starts. If the unpairing is a result of an attack, the intruder may try

brute-force methods to participate in pairing processes. To slow such attacks, every failed pairing attempt should cause a delay of increasing seconds.

Even if your application requires constant operation, keep in mind that once we reach the state of unpairing, we are either under attack or something went seriously wrong (device disconnected or powered down).

3.2 CANcryptFD secure message table

Datatype	Name	Use
UNSIGNED32	CAN ID Match	CAN ID of the secure message. Set bit 30 to indicate that a 29-bit CAN ID is used.
UNSIGNED32	CAN ID Mask	CAN ID mask value (id & mask == match)
UNSIGNED8	First encrypted byte	If using encryption, the first byte to which encryption is applied (starting at zero).
UNSIGNED8	Number of encrypted bytes	If using encryption, the number of encrypted bytes.
UNSIGNED4	Functionality	CANcrpytFD functionality used for this message.
UNSIGNED4	Method	CANcryptFD methods used for this message.
UNSIGNED4	Producer	CANcryptFD address of the device producing this message (1–14).
UNSIGNED4	Reserved	

ENTRY IN THE SECURE MESSAGE LIST TABLE

The last entry of the table is different, see below.

Datatype	Name	Use
UNSIGNED32	End of table	Set to FFFF FFFFh to mark the end of the table.
UNSIGNED16	Reserved	Set to FFFFh
UNSIGNED16	Checksum	Checksum covering all table entries without the End of Table entry.

LAST ENTRY IN THE SECURE MESSAGE LIST TABLE

Note that for optimization individual devices may only store those elements of the table that they require. If a message is not used by a local device, its details do not need to be known by the device.

Each element in the table is 8 bytes and provides details about a secure message handled by CANcryptFD. The last entry in the table must have a CAN ID of FFFF FFFFh to indicate the end of the table. The last record also uses a 16-bit checksum for the entire table.

If a high security level is desired, the configuration options for the secure message table should be limited. An intruder with access to this level (being able to edit the table) could reconfigure a device to listen for different messages other than those originally intended.

The checksum method used shall be the highest level method supported by a device. If a device supports only the regular security method, the checksum method of that level is used. The checksum initializer for this checksum shall be FFFFh.

For better optimization, each device uses two local tables, one for secure messages received by this device and one for secure messages transmitted.

3.2.1 Pairing and grouping implementation note

As the mechanisms used to produce and consume secure messages are the same for a paired and a grouped communication, the tables and other resources required may be shared for both paired and grouped communication.

However, when resources are shared, secure communication cannot be used at the same time by a device that is both grouped and paired. If the application re-

quires that secure communication is possible at the same time for both paired and grouped devices, then the keys and tables need to be duplicated, one set for the paired communication and one set for the grouped communication.

4 CANcryptFD customizable functions

CANcryptFD uses a value in the range of 0–3 to select one of four security function levels as shown in the table below (selecting CANcryptFD methods and algorithms).

Name	Value	Description
Basic	0	Minimal security level, requires minimal computational resources, usable on most microcontrollers. Cryptographic method used is the 64-bit Speck Cipher, optionally with limited rounds. Protects from accidental misuse and simple record and replay scenarios.
Regular	1	Default security level, adequate for all applications without specific security requirements, suited for 32-bit microcontrollers. Cryptographic method used is the 64-bit XTEA Cipher with full rounds.
Advanced	2	Highest security level. Uses a combination of XTEA-64 and AES-128.
Custom	3	Allows customization of all security relevant functions.

SELECTING CANCRYPTFD DEFAULT METHODS AND ALGORITHMS

All elementary CANcryptFD functions that actively influence the security level are located in the *CANcrypt_userfct.h* module. System developers can select either one of the default settings or use their own customized configuration. In this chapter we show the provided functions, for the default implementation of ESAcademy's commercial CANcryptFD release.

4.1 Collect random numbers

Both pairing and grouping functions collect random data from the participating devices to initialize the secure communication. For the initial key generation, CANcryptFD requires an array of random numbers that is as long as the current key length. This function takes the collected initial random numbers and expands them to fill the required array.

```
/*********************************************************************
BOOK:    Section 6.1 "Collect random numbers"
DOES:    This function expands an array with a limited number of
         random bytes to an array of random bytes with the length
         of the current dynamic key.
RETURNS: nothing
*********************************************************************/
void Ccuser_ExpandRandom(
  UNSIGNED32 pkey[Cc_KEY_LEN32],  // key input, length Cc_KEY_LEN32
  UNSIGNED32 pdest[Cc_KEY_LEN32], // destination: array with length
of dyn. key
  UNSIGNED32 psrc[12]             // array with zeros and random num-
bers (3*15)
)
```

4.2 Bit mixup

The bit mixup function is used in basic and regular modes to generate the initial dynamic key from the permanent key and to generate the dynamic one-time pad from the current dynamic key. This CANcryptFD implementation uses variations of the 32-bit and 64-bit Speck and XTEA Ciphesr. The number of rounds executed is configurable.

```
/*********************************************************************
Macros to rotate 32bit value right or left and a single mix up round
in add-rotate-xor (ARX) style as used by Speck cipher
*********************************************************************/
#define ROR32(x,r) ( (x >> (r & 0x1F)) | (x << (32 - (r & 0x1F))) )
#define ROL32(x,l) ( (x << (l & 0x1F)) | (x >> (32 - (l & 0x1F))) )
#define MIXROUND32(a,b,k)   (a=ROR32(a,8),a+=b,a^=k,b=ROL32(b,3),b^=a)
// extended CANcrypt version, create a disturbance in algorithm to
make it less predictable
// if you want to make use of such a feature, we recommend to use
your own, secret distortion
#define MIXROUND32x(a,b,k,j) (a=ROR32(a,9-
(j&3)),a+=b,a^=k,b=ROL32(b,1+(j&7)),b^=a)


/*********************************************************************
Rotation round of XTEA cipher
*********************************************************************/
#define MIXXTEA(d,s,k) ((((d << 4) ^ (d >> 5)) + d) ^ (s + k)
```

```
/*********************************************************************
BOOK:    Section 6.2 "Bit mixup"
DOES:    This function mixes the bits in a 64bit value by applying
         a Speck cipher. Used by key initialization functions and
         one-time pad generation.
MOTE:    Recommended number of rounds is 27,
         USING LESS ROUNDS DECREASES RELIABILITY
RETURNS: Value pmixed[] returns the mixed bits
*********************************************************************/
void Ccuser_Mix64_SPECK(
  UNSIGNED32 pkey[Cc_KEY_LEN32], // key input 64 or 128 or 256bit
  UNSIGNED32 pdat[2],   // data input of 64 bit
  UNSIGNED32 pmixed[2], // mixed bits output of 64 bit
  UNSIGNED8 rounds      // number of mixing rounds to execute
);


/*********************************************************************
BOOK:    Section 6.2 "Bit mixup"
DOES:    This function mixes the bits in a 64bit value by applying
         a XTEA cipher. Used by key initialization functions and
         one-time pad generation.
MOTE:    Recommended number of rounds is 64,
         USING LESS ROUNDS DECREASES RELIABILITY OF CIPHER
         Recommended key size 128bit,
         USING LESS ROUNDS DECREASES RELIABILITY OF CIPHER
RETURNS: Value pmixed[] returns the mixed bits
*********************************************************************/
void Ccuser_Mix64_XTEA(
  UNSIGNED32 pkey[Cc_KEY_LEN32], // key input 64 or 128 or 256bit
  UNSIGNED32 pdat[2],   // data input of 64 bit
  UNSIGNED32 pmixed[2], // mixed bits output of 64 bit
  UNSIGNED8 rounds      // number of mixing rounds to execute
);


/*********************************************************************
DOES:    This function mixes the bits in a 128bit value
RETURNS: Value pmixed[] returns the mixed bits
*********************************************************************/
void Ccuser_Mix128(
  UNSIGNED32 pkey[Cc_KEY_LEN32], // key input
  UNSIGNED32 pdat[4],   // data input of 128 bit
  UNSIGNED32 pmixed[4], // mixed bits output of 128 bit
  UNSIGNED8 rounds      // number of mixing rounds to execute
                        // ignored for AES
);
```

4.3 Generate dynamic key

When a CANcryptFD system powers up it does not yet have a shared dynamic key, only the stored permanent key. Initialization of the dynamic key depends on the connection method. In both available methods the selected permanent key is copied to the dynamic key and gets modified before its first use.

The same function is also used to update or re-generate the dynamic key in grouping mode and to generate a one-time pad from the dynamic key.

The main idea behind this is that a permanent key should never be used directly for any security functions, as that would provide an attack vector, therefore a modified copy is used.

```
/*********************************************************************
BOOK:    Section 6.3 "Generate keys"
DOES:    Takes input from 2 keys and 1 factor to create a new key.
         Used to create a dynamic key from a permanent key using
         random input and a serial number.
         Used to create a one-time pad from a permanent and
         dynamic key and a counter.
RETURNS: TRUE if key initialization completed,
         FALSE if not possible due to parameters
*********************************************************************/
UNSIGNED8 Ccuser_MakeKey(
  UNSIGNED32 pin1[Cc_KEY_LEN32],// input 1: pointer to primary key
  UNSIGNED32 pin2[Cc_KEY_LEN32],// input 2: pointer to 2nd inputy
  UNSIGNED32 factor,    // input 3: optional, set zero if not used
                        // used for serial number, counter
  UNSIGNED32 pout[Cc_KEY_LEN32] // output: the dynamic key or one
time pad
);
```

4.3.1 Grouping: Take random values from grouping message

The messages used to initialize the grouping mode each contain a 24-bit random value. These random values from all participating devices are used to initiate the dynamic key before its initial use.

4.3.2 Generate one-time pad

The one-time pad is re-generated before every use of secure messages. An individual message counter is part of the generation, ensuring that some value changes with every use.

4.4 Updating the dynamic shared key

One of the core features of CANcryptFD is that the dynamic key is continuously updated. The methods used differ between pairing and grouping.

4.4.1 Grouping: Key update based on secure heartbeat

In grouping mode, a secure heartbeat is produced.

Every secure heartbeat includes a counter and random, encrypted bytes. The secure heartbeat happens in cycles, and within each cycle, all active participants produce their heartbeats. All decrypted random values from all devices are used to update the dynamic key. This key update uses the *Ccuser_MakeKey()* function from section 4.3 to generate the dynamic key.

4.5 Secure Heartbeat and Messaging

All secure messages feature a CANcryptFD security record with a digital signature. The signature covers the entire message, including CAN ID and all data bytes.

4.5.1 Generate signature

```
/*****************************************************************
BOOK:    Section 6.5.1 "Generate signature value"
         CANCRYPT FD ADOPTED VERSION
         Modified for CANcrypt FD, signs entire CAN FD message
DOES:    Generates a signature for a CAN FD message
RETURNS: TRUE, when generation success
*****************************************************************/
UNSIGNED8 Ccuser_MakeSignature(
  CAN_MSG *pCAN,        // CAN FD message to generate signature for
  Cc_SECURITY_RECORD *pSec, // Security record, values pre-filled
  UNSIGNED32 pKey[Cc_KEY_LEN32], // key used for signature
  UNSIGNED32 pVec[Cc_KEY_LEN32]  // key used for chksum init
);
```

4.5.2 Verify signature

```
/*****************************************************************
BOOK:    Section 6.5.2 "Verify signature value"
         CANCRYPT FD ADOPTED VERSION
DOES:    Verifies a signature received in a CAN FD message
RETURNS: TRUE, if signature was verified
*****************************************************************/
UNSIGNED8 Ccuser_VerifySignature(
  CAN_MSG *pCAN,        // CAN FD message to generate signature for
  Cc_SECURITY_RECORD *pSec, // Security record, values pre-filled
  UNSIGNED32 pKey[Cc_KEY_LEN32], // key used for signature
  UNSIGNED32 pVec[Cc_KEY_LEN32]  // key used for chksum init
);
```

4.6 Encryption and decryption

Encryption algorithms are kept simple in CANcryptFD, the security effort is placed into the dynamic keys and one-time pads.

4.6.1 Secure message encryption

When it comes to secure messaging, CANcryptFD ensures that these always are made up of two CAN messages of eight bytes, providing a total data length of 128 bits. The first message is a preamble, the second the data message, with unused bytes filled with random bytes. This potentially allows 128-bit algorithms to be used if the entire message needs to be encrypted.

Per default, encryption is a single exclusive or with the current one-time pad. Only the bytes specified get encrypted.

If AES-128 is used, consider using AES-128 to generate the one-time pad, then data less than 128bit can be encrypted.

```
/*****************************************************************
BOOK:    Section 6.7.1 "Secure message encryption"
DOES:    Encrypts a data block in a secure message
NOTE:    This version NOT optimized for 32 bit architecture
RETURNS: TRUE if encryption completed,
         FALSE if not possible due to parameters
*****************************************************************/
UNSIGNED8 Ccuser_Encrypt(
  UNSIGNED32 ppad[Cc_KEY_LEN32], // pointer to current one-time pad
  UNSIGNED32 *pdat,     // pointer to the data to encrypt
  UNSIGNED16 first,     // first byte to encrypt
  UNSIGNED16 bytes      // number of bytes to encrypt
);
```

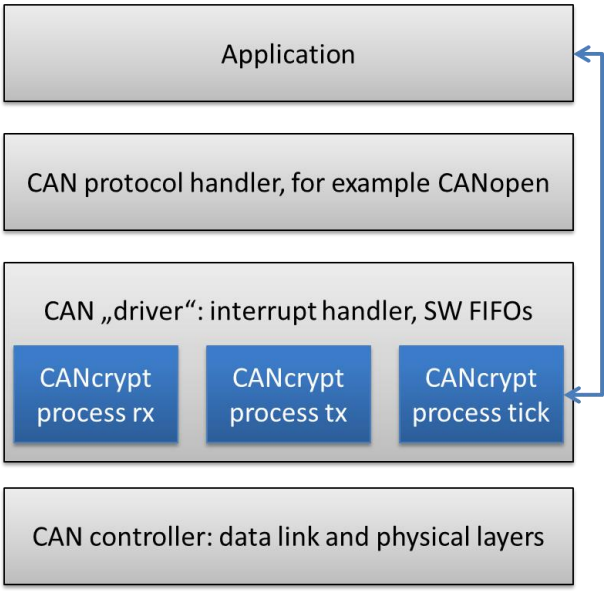
4.6.2 Secure message decryption

The decryption function uses the same parameters. If the encryption method is fully symmetric, then the encrypt function can also be used for decrypt.

```
/*****************************************************************
BOOK:    Section 6.7.2 "Secure message decryption"
NOTE:    Only used if cryptographic function is not symmetric and
         decryption requires a different function then encryption
DOES:    Decrypts a data block
RETURNS: TRUE if decryption completed,
         FALSE if not possible due to parameters
*****************************************************************/
UNSIGNED8 Ccuser_Decrypt(
  UNSIGNED32 ppad[Cc_KEY_LEN32], // pointer to current one-time pad
  UNSIGNED32 *pdat,     // pointer to the data to decrypt
  UNSIGNED16 first,     // first byte to decrypt
  UNSIGNED16 bytes      // number of bytes to decrypt
);
```

5 CANcryptFD Programming

The following diagram illustrates the simplified integration of CANcryptFD into existing CAN systems. Integration happens at the driver level and thus is independent from additional layers and the software using CAN communications. Applications need only make a few function calls to activate the CANcryptFD security system and to handle call backs from events reported by the CANcryptFD security system.



SIMPLIFIED CANCRYPTFD INTEGRATION

5.1 C include files definitions

The example demo projects provided contain the C include/definition files used for all major CANcryptFD definitions and settings.

5.1.1 CC_user_config.h

This module is used to configure the CANcryptFD operation mode, including the selected methods and timeouts.

```
/*********************************************************************
MODULE:    Cc_user_config.h, CANcrypt global user configuration
CONTAINS:  Configuration parameters for CANcrypt
AUTHORS:   Embedded Systems Academy, Inc (USA) and
           Embedded Systems Academy, GmbH (Germany)
HOME:      https://www.cancrypt.net
LICENSE:   See below, APPLIES TO THIS FILE ONLY.
           See individual file headers for applicable license.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at
  www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  implied. See the License for the specific language governing
  permissions and limitations under the License.

VERSION:   1.03, 06-AUG-2018
           $LastChangedRevision: 466 $
*********************************************************************/


#ifndef _CC_USER_CONFIG_H
#define _CC_USER_CONFIG_H


/*********************************************************************
CANcrypt code selection
*********************************************************************/

// If defined, implement grouping functionality
#define Cc_USE_GROUPING

// If defined, implement secure messaging
#define Cc_USE_SECURE_MSG

// If defined, switch output pins for performance measurements,
// as well as for debug and test
#define Cc_USE_DIGOUT


/*********************************************************************
Security CAN Functionality
Cc_SECFCT_BASIC           0x00
Cc_SECFCT_REGULAR         0x01
Cc_SECFCT_ADVANCED        0x02
Cc_SECFCT_CUSTOM          0x03
*********************************************************************/
#define Cc_FUNCTIONALITY        Cc_SECFCT_REGULAR


/*********************************************************************
Permanent key length used by this version: 128, 256 or 512
Must be greater or equal to the dynamic key length
*********************************************************************/
```

```
#define Cc_PERMKEY_LEN_BITS     256
#define Cc_PERMKEY_LEN32        (Cc_PERMKEY_LEN_BITS >> 5)
#define Cc_PERMKEY_LEN8         (Cc_PERMKEY_LEN_BITS >> 3)
// Default permanent key available
#define Cc_PERMKEY_DEFAULT      Cc_PERM_KEY_USER


/*********************************************************************
Dynamic key length used by this version: 64 or 128
*********************************************************************/
#define Cc_KEY_LEN_BITS         128
#define Cc_KEY_LEN32            (Cc_KEY_LEN_BITS >> 5)
#define Cc_KEY_LEN16            (Cc_KEY_LEN_BITS >> 4)
#define Cc_KEY_LEN8             (Cc_KEY_LEN_BITS >> 3)


/*********************************************************************
Timings used
IN THIS VERSION, INDIVIDUAL TIMINGS MUST STILL BE SET BELOW
*********************************************************************/
#define Cc_TIMINGS              Cc_TIMING_MEDIUM


/*********************************************************************
Secure heartbeat timings
*********************************************************************/
// Secure Heartbeat event time
#define Cc_SECHB_EVENT_TIME     500

// Secure Heartbeat inhibit time
#define Cc_SECHB_INHIBIT_TIME   250

// Secure Heartbeat timeout
#define Cc_SECHB_TIMEOUT        1000

// Message sequence counter increment value, ensure multiple bits
change
#define Cc_SEC_INC              0x0861


/*********************************************************************
Default key generation parameters
*********************************************************************/
// CANcrypt Method combination
#define Cc METHOD               ( Cc TIMINGS + \
   (((Cc_BITMETHOD + Cc_BITMETHOD_CLAIMS + Cc_FUNCTIONALITY) << 4)) )

// Custom bit generation cycle timeout
#define Cc_CUST_BITCYC_TIMEOUT   20

// Custom bit generation max random delay time, 0 for immedi-
ate/direct
#define Cc_CUST_BITCYC_RANDTIME  0x0F


/*********************************************************************
Enable monitoring of unexpected CAN message IDs received.
*********************************************************************/
```

```
// Maximum number of CAN IDs in list monitored, 0 to disable
#define Cc_CANIDLIST_LEN        16


/*********************************************************************
CAN IDs used
*********************************************************************/
// CANcrypt messages for devices and configurator, plus next 15 IDs
#define Cc_CANID_CONFIG         0x0171
// Bit claiming messages start with this ID, plus next 1 or 15 IDs

#define Cc_CANID_BITBASE        0x06F0

// CANcrypt messages for debug messages, plus next 15 IDs
#define Cc_CANID_DEBUG          0x06E1


/*********************************************************************
Size of event queue for call backs (used from Rx thread)
Limit selection to 4 or 8
*********************************************************************/
#define Cc_CBEVENT_QUEUE          0x04


/*********************************************************************
DEFINES: CAN HARDWARE DRIVER DEFINITIONS
*********************************************************************/

// Tx FIFO depth (must be 0, 4, 8, 16 or 32)
#define TXFIFOSIZE              16

// Rx FIFO depth (must be 0, 4, 8, 16 or 32)
#define RXFIFOSIZE              16
```

5.1.2 CANcrypt_types.h

This definition file contains the main bit and type definitions for the CANcrypt parameters, variables and configurations. See the file for details.

5.1.3 CANcrypt_api.h

This module contains the function definitions for actively controlling CANcrypt from an application.

CANcrypt system restart

The *Cc_SelectGroup()* function can be called BEFORE *Cc_Restart()* to pre-select the key used for the next grouping process and to specify the group participants expected.

```
/*****************************************************************
DOES:    Selects the next grouping event
         If none specified, last saved session key and group is used.
RETURNS: TRUE, if key availanle, else FALSE
*****************************************************************/
UNSIGNED8 Cc_SelectGroup(
  Cc_HANDLE  *pCc,       // pointer to CANcrypt handle record
  UNSIGNED8  key_major, // key ID major info (2-6)
  UNSIGNED8  key_minor, // key ID minor info (size in 32bits)
  UNSIGNED16 grp_exp ,  // Expected devices (bit 0 = unused,
                        // bit 1 = Device1, bit 2 = Device 2, etc.)
  UNSIGNED16 grp_opt    // Optional devices
);
```

The *Cc_Restart()* function is used to re-start the CANcrypt system. The entire handle is erased and re-initialized. The parameters include the address (device ID) and a control byte. Values for the control byte are defined as Cc_PAIR_CTRL_xxx and Cc_GROUP_CTRL_xxx.

The function pointers passed are the generic event call back and the CAN transmit functions used for this CANcrypt connection. The ”TxNow” is only required if the “direct” key generation method is used.

The device identification string consists of 4 values of each 32bit and is adopted from CANopen. The four values are a vendor ID, a product code, a revision number and a serial number. If you do not have a CANopen vendor ID, then set the vendor ID field to zero.

```
/*****************************************************************
DOES:    Re-start of the CANcrypt system.
RETURNS: TRUE, if completed
         FALSE, if error in parameters passed
*****************************************************************/
UNSIGNED8 Cc_Restart(
  Cc_HANDLE  *pCc,       // pointer to CANcrypt handle record
  UNSIGNED8   address,  // address of this device, set to zero if
                        // taken from config Ccnvol_GetGroupInfo()
  UNSIGNED32  control,  // Bit0-1: 00: No change to pairing
                        //         01: Restart pairing
                        //         10: Stop pairing
                        // Bit2-3: 00: No change to grouping
                        //         01: Restart grouping
                        //         10: Stop grouping
  // Call-back functions
  Cc_CB_EVENT fct_event,      // change of status, alert
  Cc_CAN_PUSH fct_pushTxFIFO, // put CAN message into Tx FIFO
```

```
  Cc_CAN_PUSH fct_pushTxNOW,  // transmit this CAN message now
  // Device identification
  UNSIGNED32 id_1018[4] // Vendor ID, product code, revision, serial
);
```

Identification

The two identification functions can also be used when devices are not yet paired or grouped. Devices may deny access to some extended identification requests if that data is only available when paired or grouped.

The application receives the responses through the generic event call back function passed on *Cc_Restart()* of the CANcrypt system.

```
/*****************************************************************
DOES:    Generate an identify message.
RETURNS: nothing
*****************************************************************/
void Cc_TxIdentify(
  Cc_HANDLE *pCc,        // pointer to CANcrypt handle record
  UNSIGNED16 version,    // CANcrypt version
  UNSIGNED8 key_id,      // key id desired
  UNSIGNED8 key_len,     // key len desired
  UNSIGNED8 cc_method,   // method desired (Cc_SECFCT_xxx)
  UNSIGNED8 cc_timing    // timing desired (Cc_TIMING_xxx)
);


/*****************************************************************
DOES:    Generates an extended identify message
RETURNS: nothing
*****************************************************************/
void Cc_TxXtdIdentify(
  Cc_HANDLE *pCc,               // pter to a CANcrypt handle record
  UNSIGNED8 device,             // device to send request to
  UNSIGNED16 index,             // index to data
  UNSIGNED8 subindex            // subindex to data
);
```

Closing a CANcrypt connection

The Cc_TxDisconnect() function can be used to end a paired or grouped connection. It informs the communication partner of the discontinuation of the connection and closes the CANcrypt secure communication channel.

```
/*****************************************************************
DOES:    Disconnect from the CANcrypt communication partners,
         sends a request to end pairing / grouping.
RETURNS: nothing
*****************************************************************/
```

```
void Cc_TxDisconnect(
  Cc_HANDLE *pCc,       // pointer to CANcrypt handle record
  UNSIGNED8 dest_addr,  // paired device ID (1-15) or 0 for group
  UNSIGNED8 reason      // reason for disconnecting, event/aler code
);
```

Secure messaging

In order to use secure messaging, all nodes transmitting or receiving secure messages must have the secure message table implemented. The table is passed to CANcrypt using the function *Cc_Load_Sec_Msg_Table()*. When grouped or paired, *Cc_Init_Sec_Msg_Table_Counter()* must be called synchronized on all devices to re-init the message counters.

```
#ifdef Cc_USE_SECURE_MSG
/*********************************************************************
DOES:    Installs the secure message handlers by passing the
         appropriate secure message tables for transmit and receive.
RETURNS: TRUE, if completed
         FALSE, if error in parameters passed
*********************************************************************/
UNSIGNED8 Cc_Load_Sec_Msg_Table(
  Cc_HANDLE *pCc,       // pointer to CANcrypt handle record
  Cc_SEC_MSG_TABLE_ENTRY *pMsgTblRx, // secure messages to receive
  Cc_SEC_MSG_TRACK_ENTRY *pMsgTrkRx, // variables for above
  Cc_SEC_MSG_TABLE_ENTRY *pMsgTblTx, // secure messages to receive
  UNSIGNED8             *pMsgTrkTcnt // counter for above
);


/*********************************************************************
DOES:    Initialize the transmit and receive counters for the secure
         messages, may only be called "synchronized" for all
         paired / grouped devices, e.g. directly with pairing /
         grouping confirmation.
RETURNS: nothing
*********************************************************************/
void Cc_Init_Sec_Msg_Table_Counter(
  Cc_HANDLE *pCc        // pointer to CANcrypt handle record
);
#endif
```

The function *Cc_Process_secMsg_Rx()* determines if a message received requires security treatment. If a CANcrypt preamble is received, then it gets copied to a buffer. Processing only continues if the data message belonging to the preamble is received.

```
/*********************************************************************
DOES:    This is the secure CAN Rx function of CANcrypt, needs to be
         called before a message received goes into receive FIFO.
RETURNS: 0: message ignored by CANcrypt,                ADD TO FIFO
         1, message is a preamble,              DO NOT ADD TO FIFO
         2, secure message, and it is secure,           ADD TO FIFO
```

```
          3, message requires security, but we are not paired
                                                    DO NOT ADD TO FIFO
*********************************************************************/
UNSIGNED8 Cc_Process_secMsg_Rx(
  Cc_HANDLE *pCc,        // pointer to CANcrypt handle record
  CAN_MSG *pCANrx        // pointer to CAN message received
);
```

The function *Cc_Process_secMsg_Tx()* must be called if a message to be transmitted should be only transmitted as a secure message. If the message is in the list of secure messages, then the CANcrypt system applies the security features and generates the preamble for the message.

```
/*********************************************************************
DOES:     This is the secure CAN transmit function of CANcrypt. It
          must be called before the transmit message is copied
          to the transmit FIFO, as a preamble might need to be
          inserted first.
RETURNS: 0: message ignored (not handled) by CANcrypt     ADD TO FIFO
          1: message is secured by CANcrypt,  ADD PREAMBLE&MSG TO FIFO
          2: message requires security, but we are not paired
                                                    DO NOT ADD TO FIFO
*********************************************************************/
UNSIGNED8 Cc_Process_secMsg_Tx(
  Cc_HANDLE *pCc,        // pointer to CANcrypt handle record
  CAN_MSG *preamble,     // pointer to CAN buffer for preamble
  CAN_MSG *pCANtx        // pointer to CAN message to transmit
);
```

Misc functions

The Cc_TxDisconnect() function can be used to end a paired or grouped connection. It informs the communication partner of the discontinuation of the connec

```
/*********************************************************************
DOES:     Generate a response message of type acknowledge or abort.
RETURNS: nothing
*********************************************************************/
void Cc_TxAckAbort(
  Cc_HANDLE *pCc,        // pointer to CANcrypt handle record
  UNSIGNED8 ack,         // TRUE for Ack, FALSE for Abort
  UNSIGNED8 dest_addr,   // destination device ID (1-15)
                         // or 0 for broadcast
  UNSIGNED8 key_id,      // key id for this acknowledge, 0 if unused
  UNSIGNED8 key_len      // key len for this acknowledge, 0 if unused
);


/*********************************************************************
DOES:     Generate an alert message.
RETURNS: nothing
*********************************************************************/
```

```
void Cc_TxAlert(
  Cc_HANDLE *pCc,        // pointer to CANcrypt handle record
  UNSIGNED8 dest_addr,   // destination device ID (1-15)
                         // or 0 for broadcast
  UNSIGNED16 alert       // 16bit alert or error code
);
```

Cyclic processes

The function *Cc_Process_Tick()* needs to be called cyclically, preferably multiple times per millisecond. If called less frequent or integrated in a Real-Time Operating System, the call should be

```
while (Cc_Process_Tick(pCc)
{
}
```

This ensures that it keeps processing as long as there are some CANcrypt tasks to execute.

This function calls the sub-tasks of the CANcrypt system.

```
/*****************************************************************
DOES:    Main CANcrypt householding functionality. Call cyclicly.
         Primarily monitors timeouts and satet transitions.
RETURNS: TRUE, if there was something to process
         FALSE, if there was nothing to do
*****************************************************************/
UNSIGNED8 Cc_Process_Tick(
  Cc_HANDLE *pCc         // pointer to CANcrypt handle record
);

/*****************************************************************
Same as above, but for individual tasks within CANcrypt:
bit and key generation process, pairing, grouping, monitoring
*****************************************************************/
UNSIGNED8 Cc_Process_Key_Tick(Cc_HANDLE *pCc);
UNSIGNED8 Cc_Process_Pair_Tick(Cc_HANDLE *pCc);
UNSIGNED8 Cc_Process_Group_Tick(Cc_HANDLE *pCc);
UNSIGNED8 Cc_Process_secMsg_Tick(Cc_HANDLE *pCc);
UNSIGNED8 Cc_Process_Monitor_Tick(Cc_HANDLE *pCc);
```

CAN receive triggered processes

The *Cc_Process_Rx()* function needs to be called directly from the CAN receive interrupt. From here, an incoming message is distributed to the appropriate subsystem.

At the end, TRUE is returned, if this message should not be passed on to the application. In this case, the driver/interrupt has to dismiss/ignore it.

```
/*****************************************************************
DOES:    This is the main CAN receive function of CANcrypt, must be
         called directly from CAN receive interrupt. Distributes a
         message to the other Cc_Process_xxx_Rx() functions.
RETURNS: TRUE, if this message was processed by CANcrypt
         FALSE, if this message was ignored by CANcrypt
*****************************************************************/
UNSIGNED8 Cc_Process_Rx(
  Cc_HANDLE *pCc,        // pointer to CANcrypt handle record
  CAN_MSG *pCANrx        // pointer to CAN message received
);

/*****************************************************************
Same as above, but for individual tasks within CANcrypt:
bit and key generation process, pairing, grouping, monitoring
*****************************************************************/
UNSIGNED8 Cc_Process_Key_Rx(Cc_HANDLE *pCc, CAN_MSG *pCANrx);
UNSIGNED8 Cc_Process_Pair_Rx(Cc_HANDLE *pCc, CAN_MSG *pCANrx);
UNSIGNED8 Cc_Process_Group_Rx(Cc_HANDLE *pCc, CAN_MSG *pCANrx);
UNSIGNED8 Cc_Process_Monitor_Rx(Cc_HANDLE *pCc, CAN_MSG *pCANrx);
```

5.2 Low-level driver interfacing

CANcrypt requires access to the following system resources:

1.) The CAN communication interface
2.) A one Millisecond timer
3.) Non-volatile memory for storage of configuration and keys

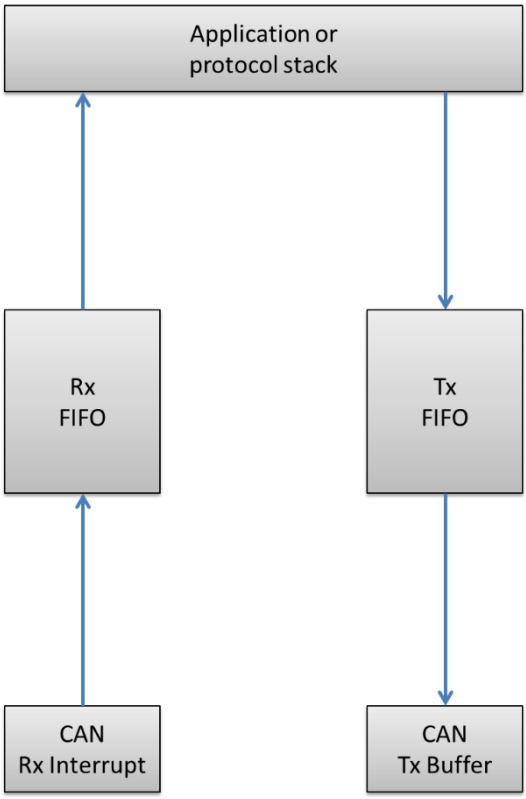
This chapter describes the detailed requirements for these interfaces.

5.2.1 CAN interface access

To simplify required code changes to existing implementations, the programming interface provides hooks to typical CAN driver processing functions. Several functions that are time critical typically need to be integrated at the lowest driver level, directly in the CAN receive interrupt routine. When integrated at this level, the changes to the user or application level are minimal.

The diagram below illustrates a typical CAN driver with FIFOs (First-In, First-Out buffers). A CAN interrupt service routine copies received CAN messages into a receive FIFO. The messages are processed later by a protocol stack (such as CANopen) or directly by the application. Messages transmitted by the application or

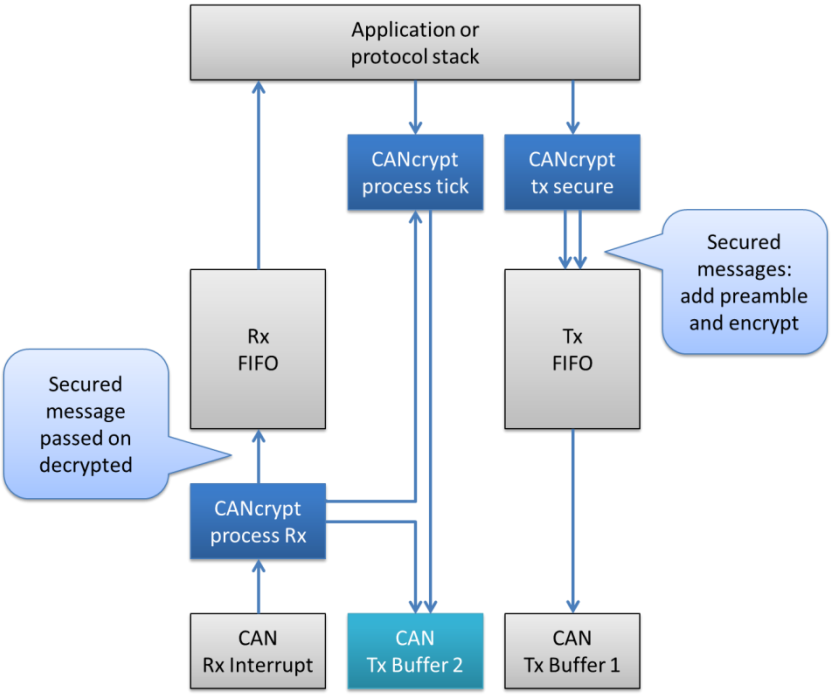
protocol stack are added to a transmit FIFO and from there go into a CAN controller transmit buffer.



TYPICAL FIFO CAN DRIVER

CANcrypt functions can be fully integrated into this scheme, minimizing the impact on the application or the protocol stack.

Once the CANcrypt system is integrated and active (paired device found), no changes are required in regard to receiving CAN messages. The CANcrypt receive handler (CANcrypt process Rx) is integrated into the CAN receive interrupt handler and copies received secure messages only after they have been authenticated and decrypted.



FUNCTION "HOOKS" BETWEEN CANCRYPT AND DRIVER

In regard to transmission of secure messages, the application or protocol stack can add unsecured messages to the transmit FIFOs in the same manner as without CANcrypt. However, for a secure transmission, CANcrypt generates the appropriate preamble and encrypted message.

If used with the option for the fastest bit-generation cycle (direct response to trigger, not random delay), a dedicated CAN transmit buffer (CAN Tx buffer 2 in figure above) is required.

The background handler (process tick) manages the pairing of devices and updating the dynamic key.

Moving CAN messages

In CANcrypt a CAN message is defined as a structure of the CAN ID (data type of 16 bits or 32 bits depending if CAN is used with 11-bit or 29-bit CAN message identifiers), the data length and the data. This is a common definition used by many drivers.

When a CAN message is passed on to a FIFO or another handler, then the parameters passed are only a pointer to the CAN message structure and the value returned is a Boolean. The return value is TRUE, if the message was passed on without errors.

```
/*****************************************************************
DOES:    Function to pass on a CAN message to a buffer
RETURNS: TRUE, if message was accepted
         FALSE, if message could not be processed
*****************************************************************/
typedef UNSIGNED8 (*Cc_CAN_PUSH) (
  CAN_MSG *pCAN                   // CAN message to transfer
);
```

When initializing CANcrypt via the *Cc_Restart()* function, a total of two such driver functions need to be passed on to CANcrypt. These are:

fct_pushTxFIFO
This function places the CAN message passed into the regular transmit FIFO/queue. If there are already messages in this transmit FIFO it will go out after all the messages in the FIFO went out.

fct_pushTxNOW
This function bypasses the transmit FIFO and directly places this CAN message into a CAN transmit buffer of the CAN controller.

Function currently NOT used by CANcryptFD!

5.2.2 Random numbers, timer and timeout

CANcrypt uses timings based on milliseconds. Only two functions are required.

```
/*********************************************************************
MODULE:    CANcrypt_hwsys.h, Misc HW system functions
CONTAINS:  Random number generation and timers
AUTHOR:    2017 Embedded Systems Academy, GmbH
HOME:      www.esacademy.com/cancrypt

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at
  www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  implied. See the License for the specific language governing
  permissions and limitations under the License.

VERSION:   0.10, 19-JAN-2017
*********************************************************************/

#ifndef _CANCRYPT_HWSYS_H
#define _CANCRYPT_HWSYS_H


#include "Cc_user_types.h"


/*********************************************************************
DOES:    Generates a random value.
NOTE:    Must be suitable for security use, shall not produce the
         same sequence of numbers on every reset or power up!!
RETURNS: Random value
*********************************************************************/
INTEGER32 CCHW_Rand(
  void
);


/*********************************************************************
DOES:    This function reads a 1 millisecond timer tick. The timer
         tick must be a UNSIGNED16 and must be incremented once per
         millisecond.
RETURNS: 1 millisecond timer tick
*********************************************************************/
UNSIGNED16 CCHW_GetTime (
  void
  );
```

```
/*********************************************************************
DOES:    This function compares a UNSIGNED16 timestamp to the
         internal timer tick and returns 1 if the timestamp
         expired/passed.
RETURNS: 1 if timestamp expired/passed
         0 if timestamp is not yet reached
NOTES:   The maximum timer runtime measurable is 0x7FFF
*********************************************************************/
UNSIGNED8 CCHW_IsTimeExpired (
  UNSIGNED16 timestamp // Timestamp to be checked for expiration
  );


#endif
/*---------------------- END OF FILE ----------------------------*/
```

5.2.3 Non-Volatile memory access

Non-volatile memory, like EEPROM, is needed to store keys and configurations. If a system is hard-coded, these parameters could also all be stored in code memory area.

```
/*********************************************************************
MODULE:    CANcrypt_nvol.h, access to non volatile memory
CONTAINS:  Functions that access data stored in NVOL memory
AUTHOR:    2017 Embedded Systems Academy, GmbH
HOME:      www.esacademy.com/cancrypt

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at
  www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  implied. See the License for the specific language governing
  permissions and limitations under the License.

VERSION:   0.10, 19-JAN-2017
*********************************************************************/


#ifndef _CANCRYPT_NVOL_H
#define _CANCRYPT_NVOL_H

#include "CANcrypt_types.h"

/*********************************************************************
DOES:    This function saves the current grouping parameters
RETURNS: TRUE, if saved, FALSE if failed
*********************************************************************/
UNSIGNED8 Ccnvol_SaveGroupInfo(
  UNSIGNED8 my_addr,    // address (1-15) to use by this device
  UNSIGNED16 grp_info   // bits 1-15 set for each device in group
                        // bit 0 set if grouping is disabled
);
```

Key hierarchy access

These functions provide access to the key hierarchy. Erase or save commands are only called by CANcrypt, if a request with the appropriate authorization has been received.

```
/*********************************************************************
BOOK:    Section 6.1 "Key hierarchy access"
DOES:    This function directly returns a key from the key hierarchy
RETURNS: Pointer to the key or NULL if not available
*********************************************************************/
```

```
UNSIGNED32 *Ccnvol_GetPermKey(
  UNSIGNED8 key_ID      // key major ID, 2 to 6
);


/*********************************************************************
BOOK:    Section 6.1 "Key hierarchy access"
DOES:    This function erases a key from the key hierarchy. Will only
         be called from CANcrypt, if called from authorized
         configurator.
RETURNS: TRUE, if key was erased, else FALSE
*********************************************************************/
UNSIGNED8 *Ccnvol_ErasePermKey(
  UNSIGNED8 key_ID      // key major ID, 2 to 6
);


/*********************************************************************
BOOK:    Section 6.1 "Key hierarchy access"
DOES:    This function saves a key to the key hierarchy. Will only
         be called from CANcrypt, if called from authorized
         configurator.
RETURNS: TRUE, if key was erased, else FALSE
*********************************************************************/
UNSIGNED8 *Ccnvol_SavePermKey(
  UNSIGNED8 key_ID,     // key major ID, 2 to 6
  UNSIGNED32 *pkey      // pointer to key data
);

#endif
/*---------------------- END OF FILE ----------------------------*/
```

5.3 Secure message configuration

The secure message list record is defined in CANcrypt_types.h.

```
/*********************************************************************
Book section 5.4 "CANcrypt secure message table"
*********************************************************************/
typedef struct {
  COBID_TYPE  CAN_ID;       // CAN message ID of the secure message
  UNSIGNED8   EncryptFirst; // first byte to encrypt
  UNSIGNED8   EncryptLen;   // number of bytes to encrypt
  UNSIGNED8   FunctMethod;  // function and methods
  UNSIGNED8   Producer;     // address (1-15) of the producer
} Cc_SEC_MSG_TABLE_ENTRY;
```

The application needs to provide two arrays with these records, one array with the secure messages to receive, and one with the secure messages to transmit. The last entry in the list needs to be all values FFh to indicate the end of the table.

The example below shows lists with two secure transmit messages and three secure receive messages.

```
// secure message table for transmit
Cc_SEC_MSG_TABLE_ENTRY TxSecMg[3] = {
  0x0183, 2, 4, 0, 3,
  0x0283, 0, 6, 0, 3,
  0xFFFF, 0xFF, 0xFF, 0xFF, 0xFF
};

// secure message table for receive
Cc_SEC_MSG_TABLE_ENTRY RxSecMg[4] = {
  0x0181, 2, 4, 0, 1,
  0x0182, 2, 4, 0, 1,
  0x0281, 0, 6, 0, 1,
  0xFFFF, 0xFF, 0xFF, 0xFF, 0xFF
};
```

When passing these lists on to CANcrypt via the *Cc_Load_Sec_Msg_Table()* function, CANcrypt also requires lists with state tracking information. For the example above they can be allocated as follows:

```
// secure message tracking info required by CANcrypt
UNSIGNED8 TxTrk[2];
Cc_SEC_MSG_TRACK_ENTRY RxTrk[3];
```

The call to activate the lists above is:

```
Cc_Load_Sec_Msg_Table(&CcH,RxSecMg,RxTrk,TxSecMg,TxTrk);
```

5.4 Driver implementation

The CAN blocks on various microcontrollers may be quite different. This section shows implementation details for the NXP LPC54618 microcontroller.

5.4.1 CAN queue / FIFO

Often a CAN software FIFO buffer (**f**irst-**i**n-**f**irst-**o**ut) queue is used as a hardware abstraction layer. If an application uses the same FIFO on different target micro-controllers, than it can be independent from the various differences of the CAN controllers. This sections shows the simple implementation of a CAN receive and transmit FIFO.

There are separate FIFOs for transmit and receive. The functions for each FIFO are a flush (erase), getting the current “in” or “out” pointers and a “done” call when data was copied. See the next chapter for a usage example.

```
/*****************************************************************
MODULE:    canfifo.c, CAN FIFO demo - message queues
CONTAINS:  CAN transmit and receive FIFO
COPYRIGHT: Embedded Systems Academy GmbH, 2016-2017
HOME:      www.esacademy.com/cancrypt
LICENSE:   FOR EDUCATIONAL AND EVALUATION PURPOSE ONLY!
CONTACT:   Contact info@esacademy.de for other available licenses

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.

VERSION:   0.10, 19-JAN-2017
*****************************************************************/

#include "CANcrypt_includes.h"


#if (TXFIFOSIZE != 0) && (TXFIFOSIZE != 4) && (TXFIFOSIZE != 8) &&
(TXFIFOSIZE != 16) && (TXFIFOSIZE != 32) && (TXFIFOSIZE != 64)
 #error "TXFIFOSIZE must be 0 (deactivated), 4, 8, 16, 32 or 64"
#endif
#if (RXFIFOSIZE != 0) && (RXFIFOSIZE != 4) && (RXFIFOSIZE != 8) &&
(RXFIFOSIZE != 16) && (RXFIFOSIZE != 32) && (RXFIFOSIZE != 64)
 #error "RXFIFOSIZE must be 0 (deactivated), 4, 8, 16, 32 or 64"
#endif


/*****************************************************************
MODULE VARIABLES
*****************************************************************/
```

```
typedef struct
{
#if (TXFIFOSIZE > 0)
  CAN_MSG TxFifo[TXFIFOSIZE];
#endif
#if (RXFIFOSIZE > 0)
  CAN_MSG RxFifo[RXFIFOSIZE];
#endif
#if (TXFIFOSIZE > 0)
  UNSIGNED8 TxIn;
  UNSIGNED8 TxOut;
#endif
#if (RXFIFOSIZE > 0)
  UNSIGNED8 RxIn;
  UNSIGNED8 RxOut;
#endif
} CANFIFOINFO;

// Module variable with all FIFO information
CANFIFOINFO mCF;


/***************************************************************************
PUBLIC FUNCTIONS
****************************************************************************/
```

Transmit FIFO / queue

```
#if (TXFIFOSIZE > 0)
/***************************************************************************
DOES:    Flushes / clears the TXFIFO, all data stored in FIFO is lost
RETURNS: nothing
****************************************************************************/
void CANTXFIFO_Flush (
  void
  )
{
  mCF.TxIn = 0;
  mCF.TxOut = 0;
}


/***************************************************************************
DOES:    Returns a CAN message pointer to the next free location in
         the FIFO. The application may then copy a CAN message to the
         location given by the pointer and MUST call
         CANTXFIFO_InDone() when copy completed.
RETURNS: CAN message pointer into FIFO
         NULL if FIFO is full
****************************************************************************/
```

```
CAN_MSG *CANTXFIFO_GetInPtr (
  void
  )
{
UNSIGNED8 ovr; // check if FIFO is full

  ovr = mCF.TxIn + 1;
  ovr &= (TXFIFOSIZE-1);

  if (ovr != mCF.TxOut)
  {// FIFO is not full
    return &(mCF.TxFifo[mCF.TxIn]);
  }
  return 0;
}

/*****************************************************************
DOES:    Must be called by the application after data was copied into
         the FIFO, this increments the internal IN pointer to the
         next free location in the FIFO.
RETURNS: nothing
*****************************************************************/
void CANTXFIFO_InDone (
  void
  )
{
  // Increment IN pointer
  mCF.TxIn++;
  mCF.TxIn &= (TXFIFOSIZE-1);
}

/*****************************************************************
DOES:    Returns a CAN message pointer to the next OUT message in the
         FIFO. The application may then copy the CAN message from the
         location given by the pointer to the desired destination and
         MUST call CANTXFIFO_OutDone() when done.
RETURNS: CAN message pointer into FIFO
         NULL if FIFO is empty
*****************************************************************/
CAN_MSG *CANTXFIFO_GetOutPtr (
  void
  )
{
  if (mCF.TxIn != mCF.TxOut)
  { // message available in FIFO
    return &(mCF.TxFifo[mCF.TxOut]);
  }
  return 0;
}
```

```
/*****************************************************************
DOES:    Must be called by application after data was copied from the
         FIFO, this increments the internal OUT pointer to the next
         location in the FIFO.
RETURNS: nothing
*****************************************************************/

void CANTXFIFO_OutDone (

  void
  )
{
  mCF.TxOut++;
  mCF.TxOut &= (TXFIFOSIZE-1);
}
#endif // (TXFIFOSIZE > 0)
```

Receive FIFO / queue

```
#if (RXFIFOSIZE > 0)
/*****************************************************************
DOES:    Flushes / clears the RXFIFO, all data stored in FIFO is lost
RETRUNS: nothing
*****************************************************************/
void CANRXFIFO_Flush (
  void
  )
{
  mCF.RxIn = 0;
  mCF.RxOut = 0;
}

/*****************************************************************
DOES:    Returns a CAN message pointer to the next free location in
         the FIFO. The application may then copy a CAN message to the
         location given by the pointer and MUST call
         CANRXFIFO_InDone() when copy completed.
RETURNS: CAN message pointer into FIFO, NULL if FIFO is full
*****************************************************************/
CAN_MSG *CANRXFIFO_GetInPtr (
  void
  )
{
UNSIGNED8 ovr; // check if FIFO is full

  ovr = mCF.RxIn + 1;
  ovr &= (RXFIFOSIZE-1);

  if (ovr != mCF.RxOut)
  {// FIFO is not full
    return &(mCF.RxFifo[mCF.RxIn]);
  }
  return 0;
}
```

```
/*****************************************************************
DOES:    Must be called by the application after the data was copied
         into the FIFO, this increments the internal IN pointer to
         the next free location in the FIFO.
RETURNS: nothing
*****************************************************************/

void CANRXFIFO_InDone (
  void
  )
{
  // Increment IN pointer
  mCF.RxIn++;
  mCF.RxIn &= (RXFIFOSIZE-1);
}


/*****************************************************************
DOES:    Returns a CAN message pointer to the next OUT message in the
         FIFO. The application may then copy the CAN message from the
         location given by the pointer to the desired destination and
         MUST call CANRXFIFO_OutDone() when done.
RETURNS: CAN message pointer into FIFO, NULL if FIFO is empty
*****************************************************************/
CAN_MSG *CANRXFIFO_GetOutPtr (
  void
  )
{
  if (mCF.RxIn != mCF.RxOut)
  { // message available in FIFO
    return &(mCF.RxFifo[mCF.RxOut]);
  }
  return 0;
}


/*****************************************************************
DOES:    Must be called by application after data was copied from the FIFO,
         this increments the internal OUT pointer to the next location
         in the FIFO.
RETURNS: nothing
*****************************************************************/
void CANRXFIFO_OutDone (
  void
  )
{
  mCF.RxOut++;
  mCF.RxOut &= (RXFIFOSIZE-1);
}
#endif // (RXFIFOSIZE > 0)

/*---------------------- END OF FILE ----------------------------*/
```

5.5 Grouping Demo

The grouping demo uses 3 devices with the device IDs 2, 3 and 7.

The messages chosen to mimic a CANopen style behavior with bootup, heart-beats, emergencies and process data objects.

The secure messaging system is enabled for one transmit message of each device using CAN IDs 182h, 183h and 187h (default IDs used for CANopen Transmit PDO).

Messages contain a mixture of fixed data as well as a counter. The message 187h varies its length to include various parts of a fixed string "Plain text part, not changing.".

All devices are configured to receive the two secure PDO messages from the other two devices. In debugging mode, the echo the received data unencrypted using CAN IDs 2xxh and 3xxh to reveal the unencrypted data.

This demo does not make use of true NVOL memory storage for keys, instead they are stored in RAM. Keys are only stored until next reset or power cycle.

First initialization uses:

```
Cc_SelectGroup(&CcH,Cc_PERM_KEY_SESSION+Cc_GRPKEY_GENERATE,Cc_PERMKEY
_LEN32,0x008C,0x008C); // 2,3,7
Cc_Restart(&CcH,Cc_DEVICE_ID,Cc_GROUP_CTRL_RESTART,Cccb_Event,CCHW_Pu
shMessage,NULL,my_ident);
```

The setting “`Cc_GRPKEY_GENERATE`” requests that a new grouping session key is generated.

Note that no special functions are used for transmit/receive of the secure messages. The CANcrypt system is hooked into the transmit/receive FIFO and detects and handles secure messages without any extra intervention by the application.

After a random delay of a bout 30s, one of the grouped nodes will initiate a disconnect with save of the current session key. All devices save the session key and after a delay re-group based on the last saved session key.