CANcrypt Software Manual

ESAcademy's implementation of CANcrypt as described in "Implementing scalable CAN Security with CANcrypt"

ISBN 978-0-9987454-1-1

for version 1.0 of 12-APR-2017

A manual from

EMBEDDED SYSTEMS ACADEMY

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 CANcrypt commercial software implementation as provided by Embedded Systems Academy.

It does not contain 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 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 CANcrypt. 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.

Cc_[IMPLEMENTATION]

These directories contain CANcrypt demo implementations for different target hardware. 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 CANcrypt parameters

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

3.1.1 Device numbering and addressing

Address, Cc_DEVICE_ID

In all CANcrypt request or command messages, a 4-bit value addresses the target CANcrypt device. A value of zero broadcasts to all devices (for example, used by the identify request). Values 1–14 are for CANcrypt devices 1–14. Address 15 is reserved for the CANcrypt configurator.

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

3.1.2 The Keys

CANcrypt 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, CANcrypt 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-crypt).

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 CANCRYPT

3.1.3 Status

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

Status

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

- Bits 0–1: Pairing status
  0: not paired
  1: pairing in progress
  2: paired
  3: pairing error
- 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

3.1.4 Controls

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

Request and commands

The 4-bit request value is used in most CANcrypt 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	request, response	1–15	5
Flip bits in key	request	1–15	6
Bit or key generation	request, response	1–15	7
Bit generation trigger	request	0	8
Secure heartbeat	Event	0	9
Generic data read	secure exchange	1–15	10
Generic data write	secure exchange	1–15	11
Generic data ID	secure exchange	1–15	12
Extended Identify	request, response	1-15	13
Preamble, Epilog	event	0	14
Save last session key	event	0	15

REQUESTS USED BY CANCRYPT DEVICES AND CONFIGURATOR

The requests and commands in the table “Requests used by CANcrypt devices and configurator” are used by both devices and the configurator in the same manner.

There is one exception: the identify request and response. When used by the configurator, these requests have extra parameters. For details, see the protocol definitions in Chapter 6.

3.1.5 Methods

CANcrypt 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, CANcrypt 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
- Bit 2: Bit generation method, set to 1 for random delay, otherwise direct, immediate reply to trigger message.
- Bit 3: Number of bit generation messages used. When set, 16 bit generation messages are used, else 2.

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

Bit generation:

After each bit-generation cycle, the customizable function UpdateBit() is called and can flip the bit generated, for example depending on the permanent key. This increases security for cases where an intruder has physical access to the CAN system as the intruder cannot easily determine when a new bit generated is 0 or 1. In addition, bit stuffing is used.. This ensures that the Mixup() function used for authentication and encryption does not use a value with all the same bits.

Authentication:

The signature used for messages is 16 bits. The signature is generated by the combination of a checksum that is encrypted using a bit mixup of the current dynamic key and the message counter. In basic mode, the checksum is calculated in Fletcher style with the initialization generated from the permanent or dynamic key. In regular mode or higher a 16bit CRC checksum is used. In advanced mode AES-128 is used for encryption/decryption.

Encryption:
The encryption is based on a mixup of the current dynamic key.

3.1.6 Functionality

Individual CANcrypt 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

3.1.7 Timings

CANcrypt 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 CANcrypt 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 CANCrypt). Value 3 selects custom, manufacturer-specific timings.

CANcyrpt message timeout:
If a CANcrypt message contains a request, requiring a response, then the transmitter uses this timeout to wait for an 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
CANcrypt 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 (slowest repetition)	250 ms	500 ms	1000 ms
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 CANCRYPT

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 CANcrypt 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.

3.1.8 CANcrypt 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 communication 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

CANCRYPT ERROR COUNTER CHANGES

In the same manner, CANcrypt uses an UNSIGNED8 error counter to determine the health of the CANcrypt connection. The table below (CANCrypt Error Counter Changes) shows which events influence the error counter. Once the error counter reaches 128 or higher, the CANcrypt 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 CANcrypt secure message table

Datatype	Name	Use
UNSIGNED32	CAN ID	CAN ID of the secure message. Set bit 30 to indicate that a 29-bit CAN ID is used.
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	CANcrpyt functionality used for this message.
UNSIGNED4	Method	CANcrypt methods used for this message.
UNSIGNED4	Producer	CANcrypt 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 CANcrypt. 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.

A typical use case would be that the table initially gets set by the configurator using secure communication as defined in **Error! Reference source not found.Error! Reference source not found.**. Once the table is programmed, modifications should only be possible by a configurator with the same permanent key access used to program the table in the first place.

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 requires 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 CANcrypt customizable functions

CANcrypt uses a value in the range of 0–3 to select one of four security function levels as shown in the table below (selecting CANcrypt 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 with limited rounds (7). Only 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 Speck Cipher with full rounds (27)
Advanced	2	Highest security level, potentially suitable to also provide safety functionality. Uses AES-128.
Custom	3	Allows customization of all security relevant functions.

SELECTING CANCRYPT DEFAULT METHODS AND ALGORITHMS

All elementary CANcrypt 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 CANcrypt 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, CANcrypt 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], // dest: array with len of dyn. key
  UNSIGNED32 psrc[12]             // array with 0 and rnd nrs (3*15)
)
{
UNSIGNED32 pad[2];
UNSIGNED8 lp_s = 0;
UNSIGNED8 lp_d = 0;
UNSIGNED8 all_ins_used = 0;
UNSIGNED8 all_outs_used = 0;

  // init destination with permanent key
  memcpy(pdest,pkey,Cc_KEY_LEN8);

  while ( (all_ins_used == 0) || (all_outs_used == 0) )
  { // repeat until all input and output values have been used

    // ensure non-zero source
    if (psrc[lp_s] == 0)
    {
      psrc[lp_s] = pdest[lp_d];
    }
    if (psrc[lp_s+1] == 0)
    {
      psrc[lp_s+1] = pdest[lp_d+1];
    }

    // generate pad from permanent key and source
    Ccuser_Mix64(&(pkey[lp_d]),&(psrc[lp_s]),pad,
                                        (Cc_FUNCTIONALITY+1)*7);
    // add pad to key
    pdest[lp_d]   += pad[0];
    pdest[lp_d+1] += pad[1];

    // increment offsets
    lp d += 2;
    if (lp_d >= Cc_KEY_LEN32)
    { // end of destination reached
      lp_d = 0;
      all_outs_used = 1;
    }
    lp s += 2;
    if (lp_s >= 12)
    { // end of source reached
      lp_s = 0;
      all_ins_used = 1;
    }
  }
}
```

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 CANcrypt implementation uses variations of the 32-bit and 64-bit Speck Cipher. 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^=b,b=ROL32(b,3),b^=a)
// extended CANcrypt version, create a disturbance in algorithm
#define MIXROUND32x(a,b,k,j)
               (a=ROR32(a,9-(j&3)),a+=b,a^=b,b=ROL32(b,1+(j&7)),b^=a)


#if (Cc FUNCTIONALITY >= Cc SECFCT REGULAR)
/*********************************************************************
Same as above for 16bit
*********************************************************************/
#define ROR16(x,r) ( (x >> (r & 0x0F)) | (x << (16 - (r & 0x0F))) )
#define ROL16(x,l) ( (x << (l & 0x0F)) | (x >> (16 - (l & 0x0F))) )
#define MIXROUND16(a,b,k)  (a=ROR16(a,8),a+=b,a^=b,b=ROL16(b,3),b^=a)
#endif


/*********************************************************************
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
RETURNS: Value pmixed[] returns the mixed bits
*********************************************************************/
void Ccuser_Mix64(
  UNSIGNED32 pkey[2],   // key input
  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
)
{
  UNSIGNED32 d0 = pdat[0];
  UNSIGNED32 d1 = pdat[1];
  UNSIGNED32 key[2];
  UNSIGNED8 lp;

  key[0] = pkey[0];
  key[1] = pkey[1];

  MIXROUND32(d1, d0, key[0]); // apply key
```

```
  for (lp = 0; lp < rounds - 1; lp++)
  { // execute a round
    // key expansion
    MIXROUND32(key[1],key[0],lp);
    if ((pkey[0] & 0x33) == 0x11)
    { // CANcrypt specific distortion of Speck cipher
      // apply in 1/16th of the cases
      MIXROUND32x(d1,d0,key[0],key[1]);
    }
    else
    { // regular Speck
      MIXROUND32(d1,d0,key[0]); // apply key
    }
  }
  // return result
  pmixed[0] = d0;
  pmixed[1] = d1;
}


#if (Cc_FUNCTIONALITY >= Cc_SECFCT_REGULAR)
/*********************************************************************
Same as above, but for 16 bit instead of 32bit
*********************************************************************/
void Ccuser_Mix32(
  UNSIGNED16 pkey[2],   // key input
  UNSIGNED16 pdat[2],   // data input of 64 bit
  UNSIGNED16 pmixed[2], // mixed bits output of 64 bit
  UNSIGNED8 rounds      // number of mixing rounds to execute
)
{
  UNSIGNED16 d0 = pdat[0];
  UNSIGNED16 d1 = pdat[1];
  UNSIGNED16 key[2];
  UNSIGNED8 lp;

  key[0] = pkey[0];
  key[1] = pkey[1];

  MIXROUND16(d1, d0, key[0]); // apply key
  for (lp = 0; lp < rounds - 1; lp++)
  { // execute a round
    // key expansion
    MIXROUND16(key[1],key[0],lp);
    MIXROUND16(d1,d0,key[0]); // apply key
  }
  // return result
  pmixed[0] = d0;
  pmixed[1] = d1;
}
#endif
```

4.3 Generate dynamic key

When a CANcrypt 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: pter to 2nd input array
  UNSIGNED32 factor,    // input 3: optional, set zero if not used
                        // used for serial number, counter
  UNSIGNED32 pout[Cc_KEY_LEN32] // output: the dyn key or onetime pad
)
{
UNSIGNED8 lp;
UNSIGNED8 ret_val = FALSE;

  if ( (pin1 != NULL) && (pin2 != NULL) && (pout != NULL))
  { // parameter check ok

    // for length of key
    for (lp = 0; lp < Cc_KEY_LEN32; lp += 2)
    { // for length of keys
      // introduce factor
      pin2[lp] += factor;
      pin2[lp+1] ^= factor;
      // number of rounds depending on security level
      // Speck defined value for 64/128bit keys: 27

#if (Cc_FUNCTIONALITY <= Cc_SECFCT_REGULAR)
      Ccuser_Mix64(&(pin1[lp]),&(pin2[lp]),&(pout[lp]),
                                        (Cc_FUNCTIONALITY+1)*13);
#elif (Cc_FUNCTIONALITY == Cc_SECFCT_ADVANCED)
      // Using AES, see below;
#else
```

```
  #error "Functionality selection not supported"
#endif
    }

#if (Cc_FUNCTIONALITY == Cc_SECFCT_ADVANCED)
  #if (Cc_KEY_LEN32 != 4)
    #error "key length not supported"
  #endif
  // in advanced mode, now use AES
  AESxxx_Encryption(pin1,pin2,pout);
#endif

    ret_val = TRUE;
  }

  return ret_val;
}
```

4.3.1 Pairing: Generate a random key first

When two CANcrypt devices are paired, they use the pairing messages to exchange initial random 24-bit values. The dynamic key is initialized by making a copy of the permanent key and then using the *Ccuser_MakeKey()* function with the exchanged random value. After that, at least 16 bit-generation cycles to update the key should run before the dynamic key is used.

4.3.2 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.

4.3.3 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.

Note that the message counter is not used with the secure heartbeat, as here the key is updated automatically with every cycle.

4.4 Updating the dynamic shared key

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

4.4.1 Pairing: Key update using a single bit

In pairing mode, the bit-generation cycle is executed periodically in the background. The bit generated is used to modify the shared key.

The generated bit is shifted in from the left. Additional input from the permanent key determines if the current bit is flipped or not.

```
/*********************************************************************
BOOK:    Section 6.4.1 "Pairing: Key update using a single bit"
DOES:    Called to update a dynamic key by introducing a new bit.
RETURNS: TRUE if key update completed,
         FALSE if not possible due to parameters
*********************************************************************/
UNSIGNED8 Ccuser_UpdateDynKey(
  UNSIGNED8 bit,         // new bit to introduce to dynamic key
  UNSIGNED32 ppermkey[Cc_PERMKEY_LEN32], // pointer to perm key used
  UNSIGNED32 pdynkey[Cc_KEY_LEN32] // pointer to dynamic key
)
{
UNSIGNED8 lp;
UNSIGNED8 *pB;
UNSIGNED8 ret_val = FALSE;

  if ((ppermkey != NULL) && (pdynkey != 0))
  {
    // save last key
    memcpy(&(pdynkey[Cc_KEY_LEN32]),pdynkey,Cc_KEY_LEN8);

    // shift entire dynamic key right by 1 bit
    for(lp=Cc_KEY_LEN32-1;lp>0;lp--)
    { // loop backwards, from highest to lowest
      // shift right
      pdynkey[lp] >>= 1;
      // merge in bit from next record
      pdynkey[lp] |= (pdynkey[lp-1] << 31);
    }
    pdynkey[0] >>= 1;

    // Check if bit needs flipping
    pB = (UNSIGNED8 *) (ppermkey);
    lp = pdynkey[0] & (Cc_PERMKEY_LEN8-1);
    if ((pB[lp]) & 1)
    { // flip bit
      bit ^= 1;
    }

    // add highest bit to new 32bit
    if (bit)
    { // set bit
      pdynkey[0] |= 0x80000000ul;
    }
```

```
#if (Cc_FUNCTIONALITY >= Cc_SECFCT_REGULAR)
    if (((pB[lp]) & 1) == 0)
    { // bit
      // additional mix up
 #if (Cc_KEY_LEN_BITS >= 128)
      Ccuser_Mix64(&(pdynkey[4]),&(pdynkey[2]),&(pdynkey[0]),
                                                  Cc_FUNCTIONALITY*3);
 #else
      pB = (UNSIGNED8 *) (pdynkey);
      Ccuser_Mix32((UNSIGNED16 *)&(pB[4]),(UNSIGNED16 *)&(pB[2]),
                          (UNSIGNED16 *)&(pB[0]),Cc_FUNCTIONALITY*3);
 #endif
    }
#endif

    ret_val = TRUE;
  }

  return ret_val;
}
```

4.4.2 Grouping: Key update based on secure heartbeat

In grouping mode, a secure heartbeat is produced.

Every secure heartbeat includes three 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

The secure heartbeat contains a 32-bit signature consisting of a 24-bit random value and an 8-bit checksum. The entire value is encrypted based on the current/last dynamic key used.

4.5.1 Generate secure Heartbeat value

The secure heartbeat is device specific, so the input parameters include the random value, the dynamic key and the address of the device sending the heartbeat.

```
/*********************************************************************
BOOK:    Section 6.5.1 "Generate signature value"
DOES:    Generates a signature for this device
RETURNS: The 32bit signature
*********************************************************************/
UNSIGNED32 Ccuser_MakeSignature(
  UNSIGNED8 address,    // device ID (1-15) of this device
  UNSIGNED32 pdyn[Cc_KEY_LEN32], // pointer to dynamic key used
  UNSIGNED32 rnd        // random value, 24bit used
)
{
UNSIGNED32 secHB = 0;
UNSIGNED8 sum;
UNSIGNED8 offset;
UNSIGNED8 *p8;
#if (Cc FUNCTIONALITY >= Cc_SECFCT_REGULAR)
UNSIGNED32 key;
#endif

  if (pdyn != NULL)
  { // parameter check ok

    // initialize checksum with value from dynamic key
    p8 = (UNSIGNED8 *) pdyn;
    offset = p8[address&(Cc_KEY_LEN8-1)];
    sum = p8[offset&(Cc_KEY_LEN8-1)] + offset;

    // add first 3 bytes of checksum
    sum += (rnd & 0xFF);
    sum += ((rnd >> 8)& 0xFF);
    sum += ((rnd >> 16)& 0xFF);

    // prepare result
    secHB = sum;
    secHB <<= 24;
    secHB += (rnd & 0x00FFFFFFul);

#if (Cc FUNCTIONALITY >= Cc SECFCT REGULAR)
    Ccuser_Mix32((UNSIGNED16 *)&(pdyn[address&(Cc_KEY_LEN32-1)]),
                 (UNSIGNED16 *)&(pdyn[(address+1)&(Cc KEY_LEN32-1)]),
                 (UNSIGNED16 *)&key,Cc_FUNCTIONALITY*7);
    // encrypt with Speck cipher
    secHB ^= key;
#else
    // encrypt with XOR
    secHB ^= pdyn[address&(Cc_KEY_LEN32-1)];
#endif

  }

  return secHB;
}
```

4.5.2 Verify secure Heartbeat value

To verify a secure heartbeat, we need the value received, the device address from which the heartbeat was received and the dynamic key used to create it.

```
/*********************************************************************
BOOK:    Section 6.5.2 "Verify signature value"
DOES:    Verifies a signature received from a device
RETURNS: TRUE, if signature was verified
*********************************************************************/
UNSIGNED8 Ccuser_VerifySignature(
  UNSIGNED8 address,    // device ID (1-15) of device sending the
  UNSIGNED32 *sHB,      // signature
                        // on return, decrypted value is at location
  UNSIGNED32 pdyn[Cc_KEY_LEN32] // pointer to dynamic key used
)
{
UNSIGNED8 add;
UNSIGNED8 offset;
UNSIGNED8 *p8;
UNSIGNED8 ret_val = FALSE;
#if (Cc FUNCTIONALITY >= Cc_SECFCT_REGULAR)
UNSIGNED32 key;
#endif

  if (pdyn != NULL)
  { // parameter check ok
    // initialize checksum with value from dynamic key
    p8 = (UNSIGNED8 *) pdyn;
    offset = p8[address&(Cc KEY LEN8-1)];
    add = p8[offset&(Cc_KEY_LEN8-1)] + offset;

#if (Cc FUNCTIONALITY >= Cc SECFCT REGULAR)
    Ccuser_Mix32((UNSIGNED16 *)&(pdyn[address&(Cc_KEY_LEN32-1)]),
                 (UNSIGNED16 *)&(pdyn[(address+1)&(Cc_KEY_LEN32-1)]),
                 (UNSIGNED16 *)&key,Cc_FUNCTIONALITY*7);
    // decrypt with Speck cipher
    *sHB ^= key;
#else
    // decrypt with XOR
    *sHB ^= pdyn[address&(Cc_KEY_LEN32-1)];
#endif

    // add first 3 bytes of checksum
    add += (*sHB & 0xFF);
    add += ((*sHB >> 8)& 0xFF);
    add += ((*sHB >> 16)& 0xFF);

    // verify result
    if (add == (*sHB >> 24))
    { // sum matches
      ret_val = TRUE;
    }
  }

  return ret_val;
}
```

4.6 Secure message checksum generation

The checksum used for the secure messages are used for authentication. Therefore, we recommend to use a key dependent initializer instead of the typical 0 or all Fh. In CANcrypt, checksums are always encrypted.

The default checksum calculation is in Fletcher style, see the examples provided. In this mode a 32-bit checksum is calculated and in a final step a 16-bit value is extracted from that. For custom mode we recommend to use CRC algorithms.

4.6.1 Checksum init

This function initializes the checksum mechanism. Initializer must come from a key.

```
/*******************************************************************
BOOK:     Section 6.6.1 "Checksum init"
DOES:     Generates an initial value for the checksum, depending on
          a key passed.
RETURNS: 32 bit initial value
*******************************************************************/
UNSIGNED32 Ccuser_ChecksumInit(
  UNSIGNED32 pkey[Cc_KEY_LEN32] // pointer to a key (pad, dyn or
perm)
)
{
UNSIGNED32 sum;

  if (pkey != NULL)
  {
    sum = pkey[0] ^ pkey[1];
  }
  else
  { // not good, consider reporting an error
    sum = 0x5BADC0DE;
  }

  return sum;
}
```

4.6.2 Checksum step

This function adds a single 16-bit value to the checksum. Note that parameter passed and returned is 32-bit, as initial calculation is based on 32-bit.

```
/*******************************************************************
BOOK:     Section 6.6.2 "Checksum step"
DOES:     Calculates a 16bit checksum, adding one value at the time
RETURNS: Checksum value in lowest 16bit, highest 16bit is carry-over
*******************************************************************/
```

```
UNSIGNED32 Ccuser_ChecksumStep16(
  UNSIGNED32 last,       // initial value or last calculated value
                         // higher 16bit may include a carry-over
  UNSIGNED16 *pdat       // next 16bit value to add
)
{
UNSIGNED32 sum;

  if (pdat != NULL)
  { // parameter check ok
    // Basic version: Fletcher style checksum
    sum = last + *pdat;
  }

  return sum;
}
```

4.6.3 Checksum final

This function performs the last step of the checksum calculation. Here it generates a 16-bit value from the calculated 32-bit checksum.

```
/*******************************************************************
BOOK:    Section 6.6.3 "Checksum final"
DOES:    When checksum calculation is completed, merges 16bit
         checksum with 16bit carry ove rto final 16bit checksum.
RETURNS: Final checksum value
*******************************************************************/
UNSIGNED16 Ccuser_ChecksumFinal(
  UNSIGNED32 last        // last calculated checksum value
)
{
UNSIGNED32 sum1;
UNSIGNED32 sum2;

  // reduce sums (adding hi and low word)
  sum2 = (last >> 16);
  sum1 = (last & 0x0000FFFFul) + sum2;
  sum2 = (sum2 & 0x0000FFFFul) + (sum2 >> 16);
  // return merged sums
  return ((sum2 << 12) | sum1) & 0x0000FFFFul;
}
```

4.7 Encryption and decryption

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

4.7.1 Secure message encryption

When it comes to secure messaging, CANcrypt 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
)
{
UNSIGNED8 ret_val = FALSE;
UNSIGNED8 lp;
UNSIGNED8 *p8pad;
UNSIGNED8 *p8dat;
#if (Cc FUNCTIONALITY >= Cc SECFCT REGULAR)
UNSIGNED32 yap[2]; // yet another pad
#endif

  if ( (ppad != NULL) && (pdat != NULL) &&
       (first < Cc_KEY_LEN8) && ((first + bytes) <= Cc_KEY_LEN8)
     )
  { // parameter check ok

#if (Cc FUNCTIONALITY >= Cc_SECFCT_REGULAR)
    // init yet another pad
    yap[0] = ppad[1];
    yap[1] = yap[0] ^ ppad[0];
```

```
    p8pad = (UNSIGNED8 *) yap;
#else
    p8pad = (UNSIGNED8 *) ppad;
#endif
    p8dat = (UNSIGNED8 *) pdat;

    for (lp = 0; lp < bytes; lp++)
    { // for length of data
#if (Cc_FUNCTIONALITY >= Cc_SECFCT_REGULAR)
      // further mixup with yet another one-time pad
      Ccuser_Mix64(&(ppad[lp&((Cc_KEY_LEN32-1)>>1)]),
                                  yap,yap,Cc_FUNCTIONALITY+1);
      // XOR data with key, here with yet another pad
      p8dat[first+lp] ^= p8pad[(first+lp) & 0x07];
#else
      // XOR data with key, here entire pad
      p8dat[first+lp] ^= p8pad[(first+lp) & (Cc_KEY_LEN8-1)];
#endif
    }

    ret_val = TRUE;
  }

  return ret_val;
}
```

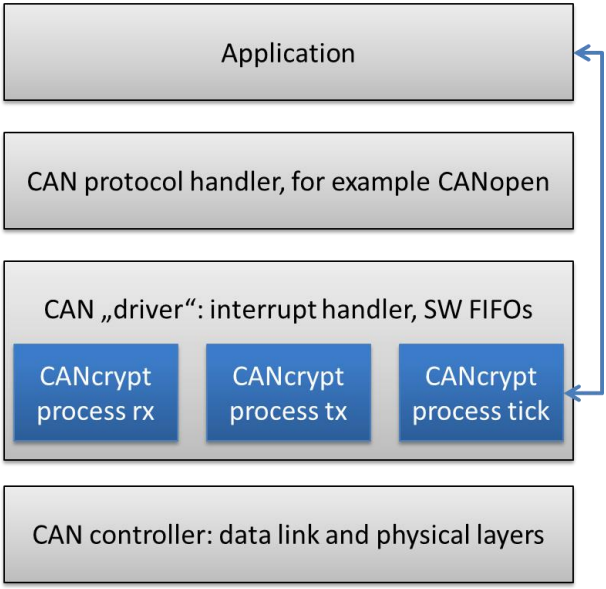
4.7.2 Secure message decryption

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
)
{
  // use encrypt function, it is symetric
  return Ccuser_Encrypt(ppad,pdat,first,bytes);
}
```

5 CANcrypt Programming

The following diagram illustrates the simplified integration of CANcrypt 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 CANcrypt security system and to handle call backs from events reported by the CANcrypt security system.



SIMPLIFIED CANCRYPT INTEGRATION

5.1 C include files definitions

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

5.1.1 CC_user_config.h

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

```
/*********************************************************************
MODULE:     Cc user config.h, CANcrypt global user configuration
CONTAINS:   Configuration parameters for CANcrypt
COPYRIGHT:  2016-2017 Embedded Systems Academy, Inc (USA) and
            Embedded Systems Academy, GmbH (Germany)
HOME:       www.cancrypt.eu
LICENSE:    LIMITED COMMERCIAL USE. FOR DETAILS SEE
            www.cancrypt.eu/docs/CANcryptLicense.pdf

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:    1.00, 12-APR-2017
            $LastChangedRevision: 301 $
*********************************************************************/


#ifndef  CC USER CONFIG H
#define _CC_USER_CONFIG_H


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

// If defined, monitor CAN for unexpected messages
#define Cc_USE_MONITORING

// If defined, implement grouping functionality
#define Cc_USE_GROUPING

// If defined, implement key generation and pairing functionality
#define Cc_USE_PAIRING

// If defined, implement secure messaging
#define Cc_USE_SECURE_MSG

// If defined, switch output pins for performance measurements
// #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, 512 or 1024
Must be greater or equal to the dynamic key length
*********************************************************************/
#define Cc_PERMKEY_LEN_BITS     128
#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         64
#define Cc_KEY_LEN32            (Cc_KEY_LEN_BITS >> 5)
#define Cc_KEY_LEN8             (Cc_KEY_LEN_BITS >> 3)

// Default length for key/bit generation cycle (<= 32)
#define Cc_KEY_LEN_INIT         32


/*********************************************************************
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

// Secure message timeout
#define Cc_SECMSG_TIMEOUT       50


/*********************************************************************
Key generation parameters
*********************************************************************/
// Bit generation cycle timeout: 10, 25, 50, 100
#define Cc_BIT_CYCLE_TIMEOUT    50

// Bit generation max random delay time, 0 for immediate/direct
//                                       or 0x0F, 0x1F, 0x3F
#define Cc_BIT_CYCLE_RANDTIME   0x1F

// Bit generation method used: random delay or direct
#define Cc_BITMETHOD            Cc_BITMETHOD_DELAY

// Number of bit claiming messages used: 2 (default) or 16
#define Cc_BITMETHOD_CLAIMS     Cc_BITMETHOD_16CLAIMS
```

```
// Maximum number of ignore states allowed for bit select cycles
#if (Cc BITMETHOD CLAIMS == Cc BITMETHOD_16CLAIMS)
 #define Cc_MAX_ignore_cnt       4
#else
 #define Cc_MAX_ignore_cnt       12
#endif

// CANcrypt Method combination
#define Cc METHOD               Cc TIMINGS + \
     (((Cc_BITMETHOD + Cc_BITMETHOD_CLAIMS + Cc_FUNCTIONALITY) << 4))


/*******************************************************************
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


#ifdef Cc_USE_SECURE_MSG
/*******************************************************************
Secure messages used
*******************************************************************/
// CAN ID, first byte encrypted, number of bytes encrypted,
// methods used, device address of producer
#define Cc secMSG 1(prd)        {0x0180 + prd, 0, 4, Cc METHOD, prd}
#define Cc_secMSG_last          {COBID_MASKID,0xFF,0xFF,0xFF,0xFF}
#endif


/*******************************************************************
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.

```
/*********************************************************************
MODULE:     CANcrypt_types.h, global type definitions
CONTAINS:   The global type definitions required for CANcrypt
AUTHORS:    Embedded Systems Academy, Inc (USA) and
            Embedded Systems Academy, GmbH (Germany)
HOME:       www.cancrypt.eu
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:    0.900, 17-MAR-2017
*********************************************************************/


#ifndef _CANCRYPT_TYPES_H
#define _CANCRYPT_TYPES_H

#include "Cc user types.h"
#include "Cc user config.h"
#include "CANcrypt_can.h"


/*********************************************************************
CANcrypt 16bit version info
6bit version, 6bit revision, 2bit reserved (zero), 2bit functionality
Version zero reserved for demo and evaluation
*********************************************************************/
// Version 1.00
#define Cc VERSION NR  1
#define Cc REVISION NR 0
#define CANcrypt VERSION ( (Cc VERSION NR << 10) + \
                           (Cc_REVISION_NR << 4) + Cc_FUNCTIONALITY )
```

```
/*********************************************************************
CANcrypt key hierarchy
Book section 5.3.2 "The Keys"
*********************************************************************/
#define Cc_PERM_KEY_MANUFACTURER  0x06
#define Cc_PERM_KEY_INTEGRATOR    0x05
#define Cc_PERM_KEY_OWNER         0x04
#define Cc_PERM_KEY_USER          0x03
#define Cc_PERM_KEY_SESSION       0x02
#define Cc_DYN_KEY_GROUP          0x01
#define Cc_DYN_KEY_PAIR           0x00


/*********************************************************************
CANcrypt status byte
Book section 5.3.3 "Status"
*********************************************************************/
// Bits 0-1: status of pairing process
#define Cc_PAIR_STAT_BITS         0x03
#define Cc_PAIR_STAT_NONE         0x00
#define Cc_PAIR_STAT_PROGRESS     0x01
#define Cc_PAIR_STAT_PAIRED       0x02
#define Cc_PAIR_STAT_FAIL         0x03
// Bits 2-3: status of grouping process
#define Cc_GROUP_STAT_BITS        0x0C
#define Cc_GROUP_STAT_NONE        0x00
#define Cc_GROUP_STAT_PROGRESS    0x04
#define Cc_GROUP_STAT_GROUPED     0x08
#define Cc_GROUP_STAT_FAIL        0x0C
// Bits 4-5: status of last command
#define Cc_CMD_STAT_BITS          0x30
#define Cc_CMD_STAT_NONE          0x00
#define Cc_CMD_STAT_SUCCESS       0x10
#define Cc_CMD_STAT_IGNORE        0x20
#define Cc_CMD_STAT_FAIL          0x30
// Bit 6: reserved
// Bit 7: status of key generation
#define Cc_KEY_STAT_BITS          0x80
#define Cc_KEY_STAT_NONE          0x00
#define Cc_KEY_STAT_GENERATION    0x80
```

```
/*********************************************************************
CANcrypt request, response, event identification
Book section 5.3.4 "Controls"
*********************************************************************/
#define Cc_CTRL_BITS              0x0F
#define Cc_CTRL_ABORT             0x00
#define Cc_CTRL_ACK               0x01
#define Cc_CTRL_ALERT             0x02
#define Cc_CTRL_IDENTIFY          0x03
#define Cc_CTRL_PAIR              0x04
#define Cc_CTRL_UNPAIR            0x05
#define Cc_CTRL_FLIPBIT           0x06
#define Cc_CTRL_KEYGEN            0x07
#define Cc_CTRL_BITNOW            0x08
#define Cc_CTRL_SECHB             0x09
#define Cc_CTRL_DATRD             0x0A
#define Cc_CTRL_DATWR             0x0B
#define Cc_CTRL_DATMSG            0x0C
#define Cc_CTRL_XTDID             0x0D
#define Cc_CTRL_RESERVED          0x0E
#define Cc_CTRL_SAVEKEY           0x0F


/*********************************************************************
CANcrypt security functionality
Book section 5.3.5 "Methods"
*********************************************************************/
// Security functionality: basic, regular, advanced or custom
#define Cc_SECFCT_BITS            0x03
#define Cc_SECFCT_BASIC           0x00
#define Cc_SECFCT_REGULAR         0x01
#define Cc_SECFCT_ADVANCED        0x02
#define Cc_SECFCT_CUSTOM          0x03
// Bit generation method: random delay or direct response
#define Cc_BITMETHOD_DIRECT       0x00
#define Cc_BITMETHOD_DELAY        0x04
// Bit claiming messages: 2 or 16
#define Cc_BITMETHOD_2CLAIMS      0x00
#define Cc_BITMETHOD_16CLAIMS     0x08


/*********************************************************************
CANcrypt timings and timeouts
Book section 5.3.7 "Timings"
*********************************************************************/
#define Cc_TIMING_BITS            0x03
#define Cc_TIMING_FAST            0x00
#define Cc_TIMING_MEDIUM          0x01
#define Cc_TIMING_SLOW            0x02
#define Cc_TIMING_CUSTOM          0x03
```

```
/*********************************************************************
CANcrypt call back event and alert / error codes
Below 0x80: status, event  -  0x80 and higher: error, alert
*********************************************************************/
#define Cc_EVENT_GENSTAT          0x00
#define Cc_EVENT_KEYSTAT          0x10
#define Cc_EVENT_PARSTAT          0x20
#define Cc_EVENT_GRPSTAT          0x30
#define Cc_EVENT_MSGSTAT          0x40
#define Cc_EVENT_GASTAT           0x50
#define Cc_EVENT_IDENT            0x60
#define Cc_EVENT_GENERR           0x80
#define Cc_EVENT_KEYERR           0x90
#define Cc_EVENT_PARERR           0xA0
#define Cc_EVENT_GRPERR           0xB0
#define Cc_EVENT_MSGERR           0xC0
#define Cc_EVENT_GAERR            0xD0

// 0x0X: unexpected ACK, signed
#define Cc_EVENT_GENACK           Cc_EVENT_GENSTAT + 1
// ungrouping event with instruction to
// save current dynamic key as session key
#define Cc_EVENT_UNGROUP_SAVE     Cc_EVENT_GENSTAT + 2
// ungrouping / unpairing without saving keys
#define Cc_EVENT_UNPAIR           Cc_EVENT_GENSTAT + 3
#define Cc_EVENT_UNGROUP          Cc_EVENT_GENSTAT + 3

// 0x1X: new key generation initated
#define Cc_EVENT_KEYGEN_INIT      Cc_EVENT_KEYSTAT + 1
// key generation completed OK
#define Cc_EVENT_KEYGEN_OK        Cc_EVENT_KEYSTAT + 2
// key generation requested flip bits
#define Cc_EVENT_KEYGEN_FLIP      Cc_EVENT_KEYSTAT + 3
// key generation flip bit acknowledge
#define Cc_EVENT_KEYGEN_FLIP_ACK  Cc_EVENT_KEYSTAT + 4
// key generation complöeted, save key
#define Cc_EVENT_KEYGEN_SAVE_KEY  Cc_EVENT_KEYSTAT + 5

// 0x2X: new pairing started
// call-back parameters: key id, key length
#define Cc_EVENT_PAIR_INIT        Cc_EVENT_PARSTAT + 1
// key generation completed OK
#define Cc_EVENT_PAIRED           Cc_EVENT_PARSTAT + 2

// 0x3X: new grouping cycle
// call-back parameters: group info, HB event time, HB inhibit time
#define Cc_EVENT_GROUP_INIT       Cc_EVENT_GRPSTAT + 1
// grouping completed
// call-back parameters: group info, HB event time, HB inhibit time
#define Cc_EVENT_GROUPED          Cc_EVENT_GRPSTAT + 2
// secure heartbeat cycle complete
#define Cc_EVENT_HBSECURED        Cc_EVENT_GRPSTAT + 3

// 0x5X: generic read access requested (device only)
// call-back parameters: index+subindex
#define Cc_GACC_READ_REQUEST      Cc_EVENT_GASTAT + 1
// generic read access completed (configurator only)
// call-back parameters: device, data length, data
```

```
#define Cc GACC READ OK           Cc EVENT GASTAT + 2
// generic write access requested (device only)
// call-back parameters: index+subindex, data length, data
#define Cc GACC WRITE REQUEST     Cc EVENT GASTAT + 3
// generic write access completed (configurator only)
// call-back parameters: device, status
#define Cc_GACC_WRITE_OK          Cc_EVENT_GASTAT + 4

// 0x6X: Configurator: identification response received
// call-back parameters: key id and size, version and method
#define Cc_CONFIG_IDENT           Cc_EVENT_IDENT + 1
// call-back parameters: length and data
#define Cc_CONFIG_XTDIDENT        Cc_EVENT_IDENT + 2


// 0x8X: general errors
// unexpected ACK, failed signature
#define Cc EVENT GENACKFAIL       Cc_EVENT_GENERR + 1
// intruder alert
#define Cc ERR INTRUDER           Cc_EVENT_GENERR + 2
// error counter overflow
#define Cc_ERR_ERRCOUNT           Cc_EVENT_GENERR + 3

// 0x9X: error in key generation
// call-backparameters: key work, key id, key position
#define Cc ERR KEY GEN TIME       Cc EVENT KEYERR + 1
// different bit expected in key generation
// call-backparameters: key work, key id, key position
#define Cc_ERR_KEY_GEN_BIT        Cc_EVENT_KEYERR + 2
// too many bits has to be ignored
// call-back parameters: key work, key id, key position
#define Cc ERR KEY GEN_IGNORE     Cc_EVENT_KEYERR + 3
// signature failed
// call-back parameters: key work, key id, key position
#define Cc_ERR_KEY_GEN_SIGFAIL    Cc_EVENT_KEYERR + 4
// Key not available
// call-back parameters: key id
#define Cc ERR KEY NO KEY         Cc_EVENT_KEYERR + 5
// flip request timeout
#define Cc ERR KEY FLIP_TIMEOUT   Cc_EVENT_KEYERR + 6
// flip request timeout
#define Cc_ERR_KEY_FLIP_FAIL      Cc_EVENT_KEYERR + 7

// 0xAX: pairing error
// trying to group and pair at the same time
#define Cc_ERR_PAIRGOUPED         Cc_EVENT_PARERR + 1
// pairing lost
#define Cc_ERR_PAIR_LOST          Cc_EVENT_PARERR + 2

// 0xBX: grouping error
// grouping failed before secure heartbeat started
#define Cc_ERR_GROUP_SESS         Cc_EVENT_GRPERR + 1
// group partner lost
// call-back parameters: group expected, group found, 0
#define Cc ERR GROUP LOST         Cc_EVENT_GRPERR + 2
// secure heartbeat failure
// call-back parameters: device with sec HB failure, 0, 0
#define Cc_ERR_SECHB_FAIL         Cc_EVENT_GRPERR + 3
```

```
// 0xCX: secure messaging error
// unexpected message failure
// call-back parameters: msg id
#define Cc_ERR_MSG_UNEXPECT       Cc_EVENT_MSGERR + 1
// secure message failure, not paired
// call-back parameters: msg id
#define Cc_ERR_MSG_NOPAIR         Cc_EVENT_MSGERR + 2
// secure message failure, not secure
// call-back parameters: msg id
#define Cc_ERR_MSG_NOSEC          Cc_EVENT_MSGERR + 3

// 0xDX: generic read/write error
// generic read access failed
#define Cc_GACC_READ_FAIL         Cc_EVENT_GAERR + 1
// generic read access timeout
#define Cc_GACC_READ_TIME         Cc_EVENT_GAERR + 2
// generic write access failed
#define Cc_GACC_WRITE_FAIL        Cc_EVENT_GAERR + 3
// generic write access timeout
#define Cc_GACC_WRITE_OK_TIME     Cc_EVENT_GAERR + 4


/*******************************************************************
CANcrypt control for Cc Restart
*******************************************************************/
// Bits 0-1: control for pairing process
#define Cc_PAIR_CTRL_NONE         0x00
#define Cc_PAIR_CTRL_RESTART      0x01
#define Cc_PAIR_CTRL_STOP         0x02
// Bits 2-3: control for grouping process
#define Cc_GROUP_CTRL_NONE        0x00
#define Cc_GROUP_CTRL_RESTART     0x04
#define Cc_GROUP_CTRL_STOP        0x08


#ifdef Cc_USE_MONITORING
/*******************************************************************
CAN message monitoring, list of CAN IDs not expected for receive
*******************************************************************/
typedef struct {
  COBID_TYPE   lst[Cc_CANIDLIST_LEN]; // list of CAN IDs
  COBID_TYPE   self_rx;     // last tx with self receive
  COBID_TYPE   unx_alert;   // unx message received
  UNSIGNED16   cur_num;     // current number of entries in list
} Cc_CANIDLIST_HANDLE;
#endif
```

```
#ifdef Cc_USE_SECURE_MSG
/*********************************************************************
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;


/*********************************************************************
For each entry in the secure message list, CANcrypt also needs a set
of
variables to track processing of those.
*********************************************************************/
typedef struct {
  CAN_MSG     preamble;      // the preamble CAN message
  UNSIGNED16  time;          // timestamp of last message
  UNSIGNED8   count;         // message counter
  UNSIGNED8   pre_rx;        // TRUE, if preamble received
} Cc_SEC_MSG_TRACK_ENTRY;
#endif


#ifdef Cc_USE_PAIRING
/*********************************************************************
CANcrypt internal state machine for key generation
*********************************************************************/
typedef enum
{
  CCkeySTATE_IDLE,           // no action, waiting for command
  CCkeySTATE_DYN_KEY_INIT,   // start initialization of dynamic key
  CCkeySTATE_START_KEY_REQ,  // configurator: start key gen. request
  CCkeySTATE_NEXT,           // now activating next random bit cycle
  CCkeySTATE_BITCYCLE,       // bit cycle now active
  CCkeySTATE_GOT_BIT,        // a bit for key is determined
  CCkeySTATE_ADD_BIT,        // the bit is added to key
  CCkeySTATE_IGNORE_BIT,     // no bit determined, ignore
  CCkeySTATE_DELAY_NEXT,     // configurator: delay until next cycle
  CCkeySTATE_OVER_OK,        // all random bit cycles complete, key ok
  CCkeySTATE_OVER_DELAY,     // delay after completion, before ACK
  CCkeySTATE_CLOSED          // channel closed not using anymore
} CCkeySTATE_TYPE;
```

```
/*********************************************************************
CANcrypt internal state machine for pairing
*********************************************************************/
typedef enum
{
  CCparSTATE_IDLE,           // not paired, wait for pairing action
#if (Cc DEVICE ID == 1)
  CCparSTATE_WAIT_RESP,      // waiting for pairing response
  CCparSTATE_RESPONSE,       // pairing response received, sending ack
  CCparSTATE_WAIT_ACK,       // wait for ack
#else
  CCparSTATE_REQ_RX,         // initial pairing request transmitted
#endif
  CCparSTATE_KEYUPDATE,      // Updating key with bit cycles
  CCparSTATE_ACKED,          // acknowledged
  CCparSTATE_PRE_PAIRED,     // delay to start
  CCparSTATE_PAIRED,         // paired
  CCparSTATE_ABORT,          // aborted
  CCparSTATE_CLOSED          // pairing closed, do not re-try
} CCparSTATE_TYPE;
#endif


#ifdef Cc USE GROUPING
/*********************************************************************
CANcrypt internal state machine for grouping
*********************************************************************/
typedef enum
{
  CCgrpSTATE_INIT,           // first initialization
  CCgrpSTATE_IDLE,           // not grouped, try to pair
  CCgrpSTATE_REQ_TX,         // initial grouping request transmitted
  CCgrpSTATE_HBINIT,         // grouping completed, init secure HB
  CCgrpSTATE_HBCYCLE1,
  CCgrpSTATE_HBCYCLE2,       // 3 states fpr each cycle
  CCgrpSTATE_HBCYCLE3,
  CCgrpSTATE_CLOSED          // grouping closed
} CCgrpSTATE_TYPE;
#endif


/*********************************************************************
DOES:    Call-back function, reports a system status change to the
         application
RETURNS: Nothing
*********************************************************************/
typedef void (*Cc_CB_EVENT) (
  UNSIGNED8   event,         // event that happened (Cc_EVENT_xxx)
  UNSIGNED32  param1,        // depending on the event, up to 3
  UNSIGNED32  param2         // event related parameters
);
```

```
/*******************************************************************
DOES:     Call-back 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
);
```

Generic, secure read and write access

CANcrypt supports a secure request and response message system. The data or parameters accessed are indentified using a 16bit index and an 8bit subindex value and is limited to a maximum of 4 bytes per request.

The index/subindex is adopted from CANopen. If you wish to define your own parameters, we recommend to use index values from 2000h to 5FFFh as in CANopen these are used for manufacturer specific parameters.

The call back functions *Cc_GA_READ()* and *Cc_GA_WRITE()* are called from within CANcrypt processing if a read/write access came in from another device. Fill-in your own application code to react to these read/write requests.

```
/*******************************************************************
DOES:     Call back function to read a generic parameter. Only called
          when request comes from apaired device and is secure.
RETURNS:  Length of data in bytes, 0 for not available, else 1-4
*******************************************************************/
typedef UNSIGNED8 (*Cc_GA_READ) (
  UNSIGNED16  index,          // index to read
  UNSIGNED8   sub,            // subindex to read
  UNSIGNED8   dat[4]          // pointer to data buffer, copy
);                            // data to here


/*******************************************************************
DOES:     Call back function to write a generic parameter. Only called
          when request comes from apaired device and is secure.
RETURNS:  Number of bytes written, 0 on error
*******************************************************************/
typedef UNSIGNED8 (*Cc_GA_WRITE) (
  UNSIGNED16  index,          // index to write
  UNSIGNED8   sub,            // subindex to write
  UNSIGNED8   length,         // length of data (1 - 4)
  UNSIGNED8   dat[4]          // data to write
);
```

```
/*******************************************************************
CANcrypt channel for generic access
*******************************************************************/
typedef struct
{
  // Generic access functions
  Cc_GA_READ  fct_GARead;   // generic read access
  Cc_GA_WRITE fct_GAWrite;  // generic write access
  CAN_MSG     can1;         // CAN message pair
  CAN_MSG     can2;
  UNSIGNED8   partner;      // device communicating with
  UNSIGNED8   gencnt;       // message counter
} Cc_GEN_ACC_HANDLE;
```

5.1.3 CANcrypt_api.h

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

```
/*******************************************************************
MODULE:     CANcrypt api.h, API functions
CONTAINS:   Definitions for CANcrypt Application Programming Interface
AUTHORS:    Embedded Systems Academy, Inc (USA) and
            Embedded Systems Academy, GmbH (Germany)
HOME:       www.cancrypt.eu
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:    0.900, 17-MAR-2017
*******************************************************************/

#ifndef _CANCRYPT_API_H
#define _CANCRYPT_API_H

#include "CANcrypt_types.h"
```

CANcrypt system restart

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
);
```

Restart of generic access

Use the *Cc_Restart_GenAccess()* function to start handling secure, generic read/write accesses made to this device. You need to pass pointers to the two call back functions handling the read and write accesses as well as the device ID of the communication partner.

In the current CANcrypt version, the secure generic read write access is limited to one communication partner. Typically the configurator and one device. The devices must be paired for this communication mode to be used.

```
/*******************************************************************
DOES:    Re-start of a generic access communication channel.
RETURNS: TRUE, if completed
         FALSE, if error in parameters passed
*******************************************************************/
UNSIGNED8 Cc_Restart_GenAccess(
  Cc_GEN_ACC_HANDLE  *pGA,     // pointer to general access handle
  Cc_GA_READ  fct_readacc,     // generic read access
  Cc_GA_WRITE fct_writeacc,    // generic write access
  UNSIGNED8   partner          // communication partner (1-15)
);
```

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
);
```

Message monitoring

In CANcrypt, active participants should monitor the network for suspicious (injected?) messages. After a CANcrypt system is started by *Cc_Restart()*, use the

following two functions to inform the system of the messages transmitted by this device.

```
#ifdef Cc USE MONITORING
/*********************************************************************
         CAN receive monitoring for unexpected messages
DOES:    Init (erase) the list of CAN IDs
RETURNS: TRUE, if list was erased
         FALSE, if list could not be erased
*********************************************************************/
UNSIGNED8 Cc Init IDList (
  Cc_CANIDLIST_HANDLE *pIDs      // pointer to list and length
);


/*********************************************************************
         CAN receive monitoring for unexpected messages
DOES:    Adds a CAN ID to the maintained list of IDs.
RETURNS: TRUE, if ID was added
         FALSE, if ID could not be added
*********************************************************************/
UNSIGNED8 Cc_Addto_IDList(
  Cc CANIDLIST HANDLE *pIDs,     // pointer to list and length
  COBID_TYPE can_id              // CAN ID to add
);
#endif
```

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
);
```

```
/*******************************************************************
DOES:    Transmits a secure message pair as used for secure mesg and
         generic access. CAN messages must be pre-filled with data.
         Generates checksum, encrypts messages and transmits them.
NOTE:    Must be grouped or paired
RETURNS: TRUE, when queued
*******************************************************************/
UNSIGNED8 Cc_TxGenAccess(
  Cc_HANDLE *pCc,               // pter to a CANcrypt handle record
  CAN_MSG *pcan1,               // pter to pair of CAN messages with
  CAN_MSG *pcan2,               // generic access request/response
  UNSIGNED8 *pcnt,              // message counter used
  UNSIGNED8 first,              // first data byte to encrypt in can2
  UNSIGNED8 num                 // number of bytes to encrypt in can2
);
```

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 sub-system.

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:

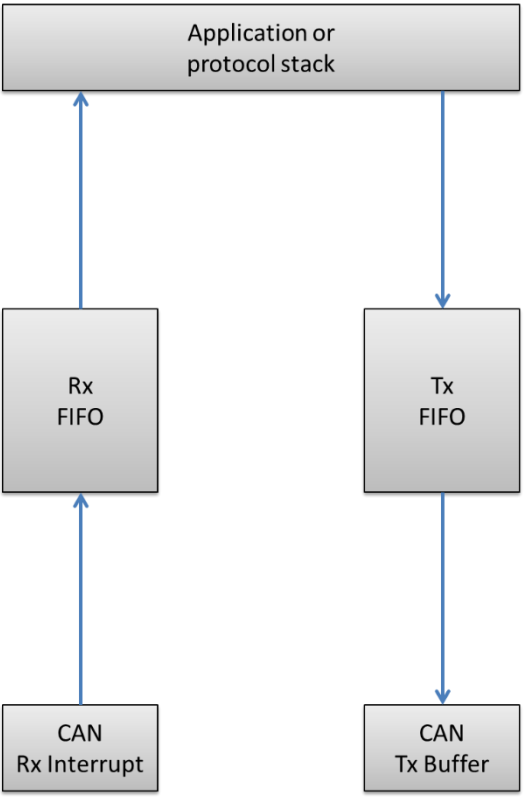
1.) The CAN communication interface
2.) A one Millisecond timer
3.) Non-volatile memory for storage of configuration and keys

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.

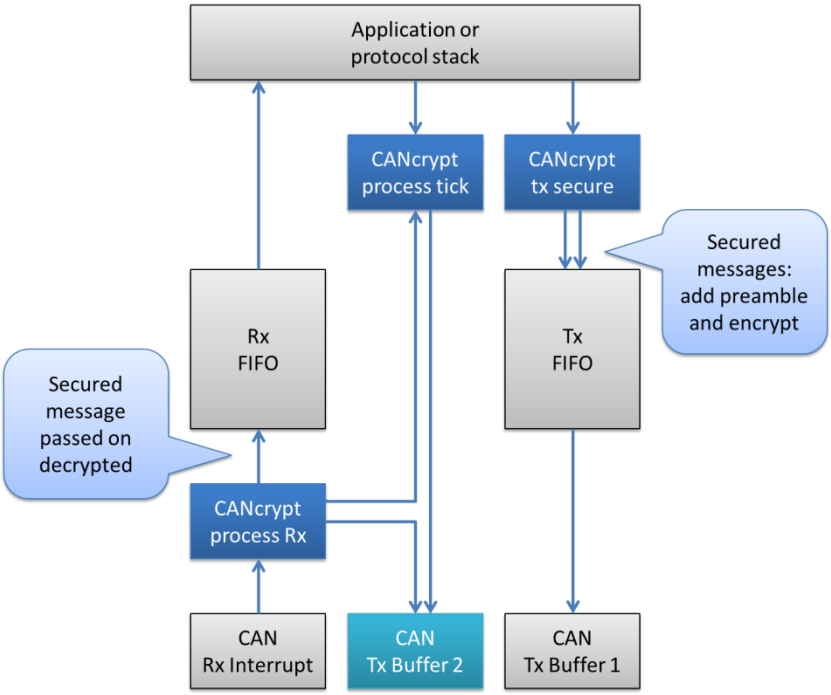
The diagram below illustrates a typical CAN driver with FIFOs (First-In, First-Out buffers). A CAN interrupt service routine copies received CAN messages into a receive FIFO. The messages are processed later by a protocol stack (such as CANopen) or directly by the application. Messages transmitted by 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. Sometimes an additional time and or control filed is added. Here we use the minimal version as shown below.

```
/*********************************************************************
Data structure for a single CAN message
*********************************************************************/
typedef struct
{ // order optimized for allignment
  UNSIGNED8 BUF[8];              // Data buffer
  COBID_TYPE ID;                 // Message Identifier
  UNSIGNED8 LEN;                 // Data length (0-8)
} CAN_MSG;
```

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. This function is only needed, if the direct bit-claiming mode is used instead of the random delay method.

```
/*********************************************************************
MODULE:    CANcrypt_can.h, CAN definitions and functions
CONTAINS:  CAN interface definitions for the CANcrypt system
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_CAN_H
#define _CANCRYPT_CAN_H

#include "Cc_user_types.h"


/*********************************************************************
Data structure for a single CAN message
*********************************************************************/
typedef struct
{ // order optimized for allignment
  UNSIGNED8 BUF[8];              // Data buffer
  COBID_TYPE ID;                 // Message Identifier
  UNSIGNED8 LEN;                 // Data length (0-8)
} CAN_MSG;


/*********************************************************************
DOES:    This function initializes the CAN interface.
RETURNS: 1 if init is completed
         0 if init failed
*********************************************************************/
UNSIGNED8 CCHW_Init (
  UNSIGNED16 BaudRate   // 1000, 800, 500, 250, 125 or 50
  );


/*********************************************************************
DOES:    This function implements a CAN transmit FIFO. With each
         function call a message is added to the FIFO.
RETURNS: 1 Message was added to the transmit FIFO
         0 If FIFO is full, message was not added
*********************************************************************/
UNSIGNED8 CCHW_PushMessage (
  CAN_MSG MEM_FAR *pTransmitBuf // CAN message to send
  );
```

```
/*********************************************************************
DOES:    This function retrieves a CAN message from the receive FIFO.
RETURNS: 1 Message was pulled from receive FIFO
         0 Queue empty, no message received
*********************************************************************/
UNSIGNED8 CCHW_PullMessage (
  CAN_MSG MEM_FAR *pReceiveBuf // Buffer for received message
  );

/*********************************************************************
DOES:    Transmission of a CANcrypt high priority message.
NOTE:    Must be transmit immediately, no delay. Bypasses Tx FIFO.
         ONLY REQUIRED FOR DIRECT BIT CLAIMING MODE
RETURNS: Nothing
*********************************************************************/
UNSIGNED8 CCHW_TransmitNow(
  CAN_MSG *pMsg
);


/*********************************************************************
DOES:    Enable/Disable of CAN receive interrupt
RETURNS: Nothing
*********************************************************************/
void CCHW_EnableCANrx (void);
void CCHW_DisableCANrx (void);


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

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"
```

Grouping information

When grouping is used, devices need to know which address/device number they have and which communication partners are expected. CANcrypt configuration supports a list of mandatory and a list of optional partners. Grouping initialization ends when all mandatory partners were found.

```
/*******************************************************************
DOES:    This function retrieves the last saved grouping parameters
RETURNS: Both my_addr and grp_info
*******************************************************************/
void Ccnvol_GetGroupInfo(
  UNSIGNED8 *my_addr,   // address (1-15) to use by this device
  UNSIGNED16 *grp_mand, // Mandatory devices to start up
                        // bits 1-15 set for each device in group
                        // bit 0 set if grouping is disabled
  UNSIGNED16 *grp_opt   // Optional devices to start up
                        // bits 1-15 set for each device in group
                        // bit 0 set if grouping is disabled
);
```

```
/*********************************************************************
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 Demo and driver example

The CAN blocks on various microcontrollers may be quite different. This section shows implementation details for the NXP LPC1768 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 ----------------------------*/
```

6 CANcrypt Application Demo

The provided application demos show different CANcrypt configurations.

5.5 Pairing Demo

The pairing demo consists of one configurator (device ID 1) and one device (default configuration device ID 3).

The configurator runs an endless loop in which it performs the following steps.

BEGIN OF LOOP

Time controlled (25, 50 and 75ms after start):

1) Restart CANcrypt and send generic identification request
2) Send extended identification request (vendor ID at 1018h,1)
3) Send extended identification request (serial number at 1018h,4)

Upon reaching 100ms after start, the configurator initiates a pairing request *Ccconfig_Request_Pair()*. On first call, based on the manufacturer key. On subsequent calls based on the user key.

Controlled by the *action* parameter, the configurator performs the following actions:

1) Generic read access of 32bit user key identifier
2) Generate user key request (section of 32bit)
3) Use flip bit command to set user key, repeat until 4 sections completed
4) Update the dynamic key (bit generation cycle)
5) Send request to save user key permanently
6) Update the dynamic key (bit generation cycle)
7) Save a 32bit user key identifier
8) Update the dynamic key (bit generation cycle)
9) Send unpair request

END OF LOOP

Repeat the loop over and over.

5.6 Grouping Demo

The grouping demo comes in two variations. The default configuration generates a cyclic secure message transmitting a counter. In terminal mode (Cc_USE_Terminal must be defined) a traditional plaintext/cyphertext communi-

cation mode is implemented where participating devices implement a terminal mode which is routed through the CANcrypt secure messaging.

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

Upon start, devices Restart the CANcrypt system, load the secure messaging table and inform the monitoring unit of the CAN IDs to watch out for.

```
Cc_Restart(&CcH,0,Cc_GROUP_CTRL_RESTART,Cccb_Event,CCHW_PushMessage,
           CCHW_TransmitNow,my_ident);

Cc_Load_Sec_Msg_Table(&CcH,RxSecMg,RxTrk,TxSecMg,TxTrk);
Cc_Addto_IDList(&(CcH.unx_lst),0x080 + Cc_DEVICE_ID);
Cc_Addto_IDList(&(CcH.unx_lst),0x170 + Cc_DEVICE_ID);
Cc_Addto_IDList(&(CcH.unx_lst),0x180 + Cc_DEVICE_ID);
Cc_Addto_IDList(&(CcH.unx_lst),0x280 + Cc_DEVICE_ID);
Cc_Addto_IDList(&(CcH.unx_lst),0x380 + Cc_DEVICE_ID);
Cc_Addto_IDList(&(CcH.unx_lst),0x480 + Cc_DEVICE_ID);
Cc_Addto_IDList(&(CcH.unx_lst),0x700 + Cc_DEVICE_ID);
```

In default mode, the secure transmit message contains a counter, in terminal mode parts of a plaintext.

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 of about 2s devices re-group based on the last saved session key.