
AVR32129: Using the 32-bit AVR UC3 CANIF

Features

- CAN Protocol Description.
- Atmel® AVR® UC3 CANIF Controller Description.
- Atmel AVR UC3 CAN source code inside the AVR Software Framework.
- Configuration and usage of Atmel AVR UC3 CAN source code.
- Example with the AVR UC3 C device.

1 Introduction

The AVR UC3 CANIF module is a Controller Area Network following and compliant with Bosch specification 2.0A and 2.0B.

This application note describes the CAN protocol and its integration inside the CANIF module. Finally, it deals with code examples to get up and running quickly. A driver interface written in C is included as well.



32-bit **AVR**[®]
Microcontrollers

Application Note

Rev. 32152A-AVR-11/10





2 Abbreviations

CAN: Controlled Area Network

MOB: Message Objects

HSB: High Speed Bus

PB: Peripheral Bus

CSMA/CD: Carrier Sense Multiple Access / Collision Detection

OSI/ISO: Open Systems Interconnection

AMP: Open Systems Interconnection

3 References

3.1.1 The AVR Software Framework

<http://asf.atmel.no/readme.html>

All pre-loaded firmware source codes are available in the AVR Software Framework version 2.0 or higher.

3.1.2 The AT32UC3C-EK Getting Started

<http://www.atmel.com/uc3c-ek>

3.1.3 CAN Protocol description

3.1.3.1 Books

- K.Etschberger: CAN Controller Area Network, Grundlagen, Protokolle, Bausteine, Anwendungen; München 2000
- W. Lawrenz: CAN Controller Area Network, Grundlagen Praxis; Heidelberg: Hüthig 1998
- W. Lawrenz: CAN Systems engineering, 1997

3.1.3.2 Specifications

- ISO 11898 available from Beuth/Berlin; <http://www.din.de>
- Review of ISO 11898 by CiA; <http://www.can-cia.de>

3.1.3.3 Internet

- CAN in Automation e.V.; <http://www.can-cia.de>
- OSEK/VDX; homepage <http://www.iiit.etec.uni-karlsruhe.de>
- Open DeviceNet Vendor Association; homepage <http://www.odva.org>

4 CAN Protocol Description

The Control Area Network (CAN) was developed by Bosh for automotive applications in 1985. The CAN is an ISO standard (ISO 11898) for serial communication. The CAN standard includes:

- The Physical layer
- The Data-link layer defines
 - Some message types and
 - Methods for fault detection and fault confinement,

Today CAN has gained widespread use Industrial Automation, Automotive, etc.

4.1 CAN Advantages

The CAN protocol is considered as a mature standard as it is more than 25 years old and numerous CAN products and tools are available on the market.

The main features of the CAN protocol can be listed as below:

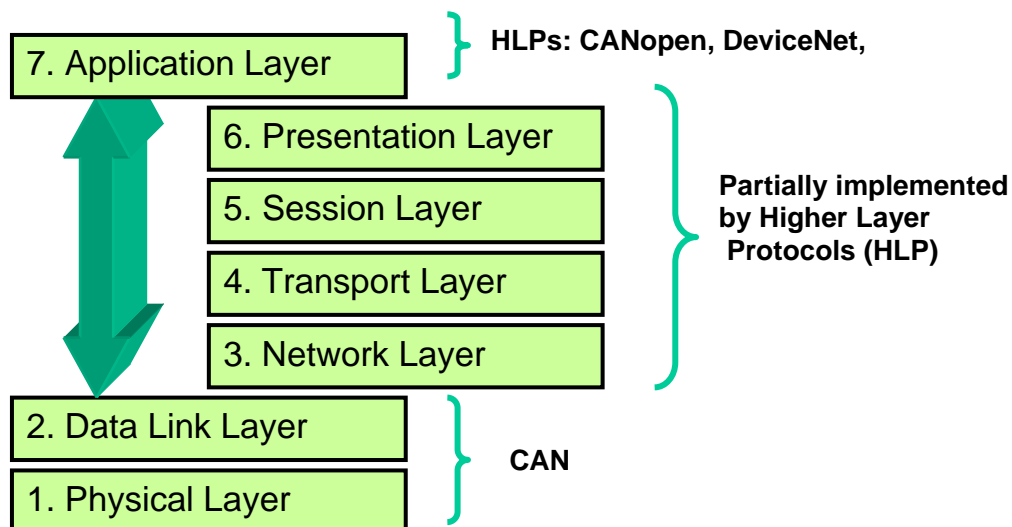
- the error handling takes part of the strong point of the protocol thanks to an extensive error detection mechanism,
- the fault confinement is a built-in feature to prevent faulty node to block system hardware implementation of the protocol,
- the simple transmission medium is a twisted pair of wires considered as a standard, but the single wire mode is also possible (other possible links: Opto or RF links),
- the multi-master / Broadcasting concept,
- the non-destructive arbitration system by CSMA/CD.

4.2 ISO-OSI* Reference Model

The CAN protocol is defined as the 2 first layers of the ISO-OSI model:

- Physical Layer,
- Data Link Layer.

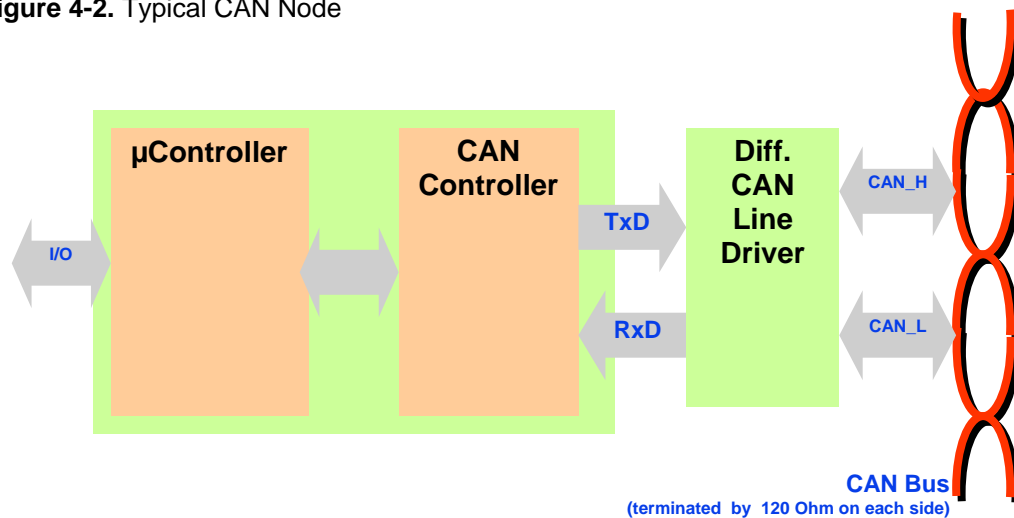
Figure 4-1. ISO-OSI Reference model



4.3 Typical CAN Node

The typical CAN node is made of two parts: a digital section to generate transmission and reception signals (TxD and RxD) and the physical section to generate differential CAN signals (CAN_H and CAN_L).

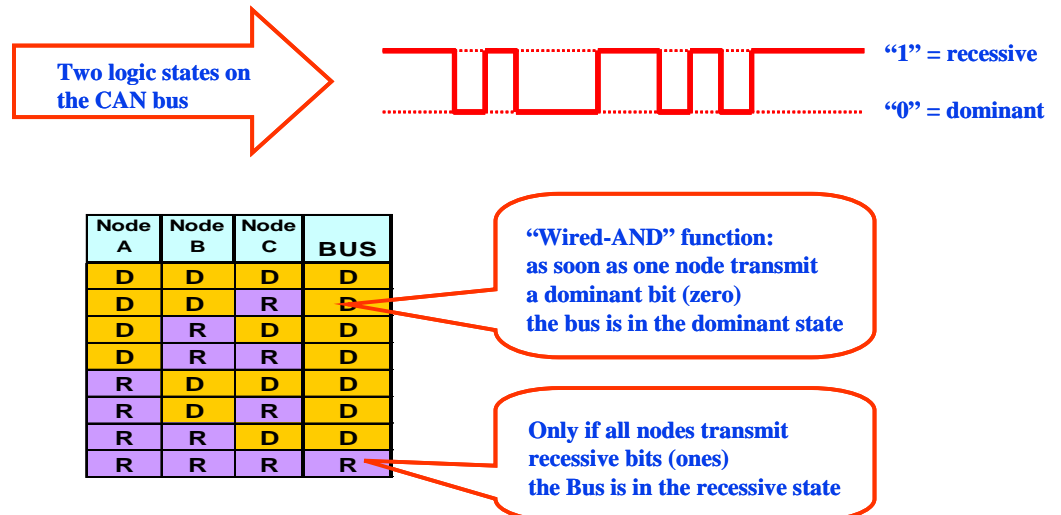
Figure 4-2. Typical CAN Node



4.4 CAN Bus Logic

The CAN bus logic is made of 2 logical levels: '1' for recessive level and '0' for dominant level. The recessive level is the default level. In case of multiple nodes on CAN bus, only the node(s) with the dominant level(s) will win.

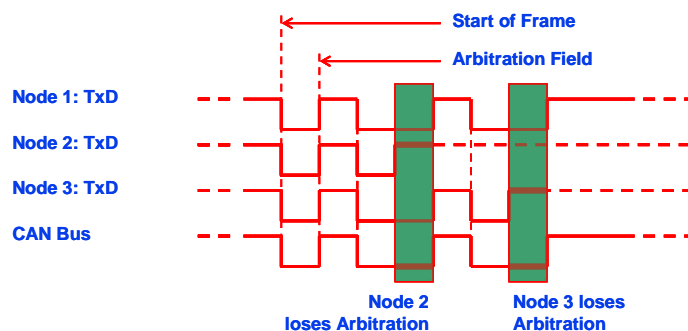
Figure 4-4. CAN Bus Logic



4.5 CAN Bus Access and Arbitration, CSMA/CD and AMP

The selection between a dominant level and recessive level is called the arbitration. In case a node has a recessive level in front of other nodes with dominant level, it loses the arbitration.

Figure 0-4. CAN Bus Access



(*)Carrier Sense Multiple Access/Collision Detection and Arbitration by Message Priority

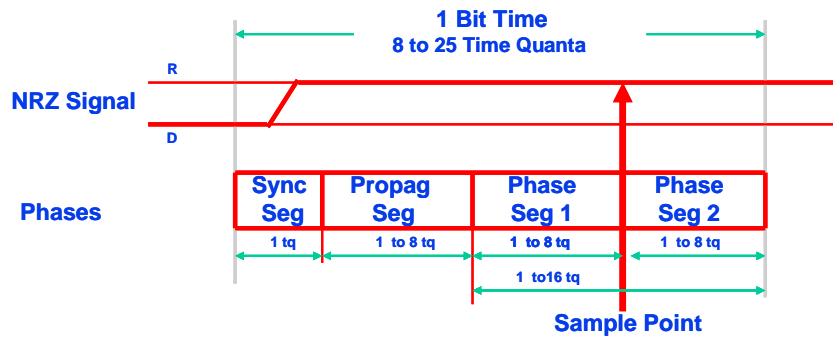
4.6 CAN Bit Construction

The quantum is the minimum ratio of time. The bit time is made of several segments:

- synchronization segment (Sync Seg),
- propagation segment (Propag Seg),
- phase segment 1 (Phase Seg1),
- phase segment 2 (Phase Seg 2)

The number of time quanta in Propag and Phase segments is programmable

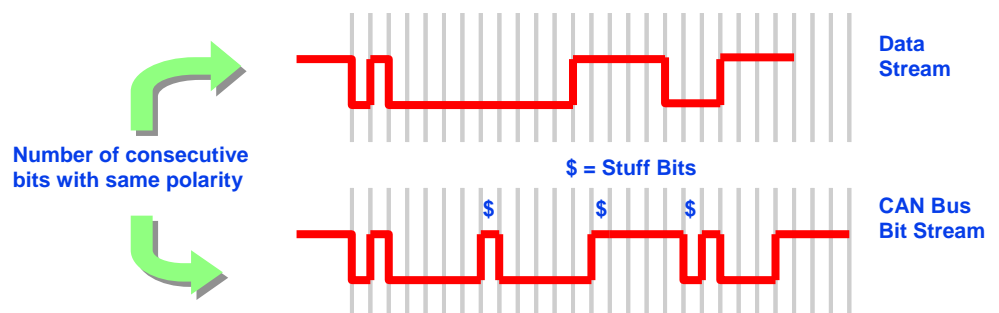
Figure 4-5. CAN Bus Access



4.7 CAN Bit Coding & Bit Stuffing

The CAN bit coding is in NRZ (Non-Return-To-Zero code) but it does not ensure enough edges for synchronization. To do so, stuff bits are inserted after 5 consecutive bits of the same level. These stuff bits have the inverse level of the previous bit transmitted on CAN Bus.

Figure 4-6. CAN Bus Access



4.8 CAN Bus Synchronization

The CAN Bus synchronization is made of two sorts:

- the hard synchronization at the beginning of the reception of a frame,
- the re-synchronization at each dominant to recessive transition.

Figure 4-7. Hard Synchronization

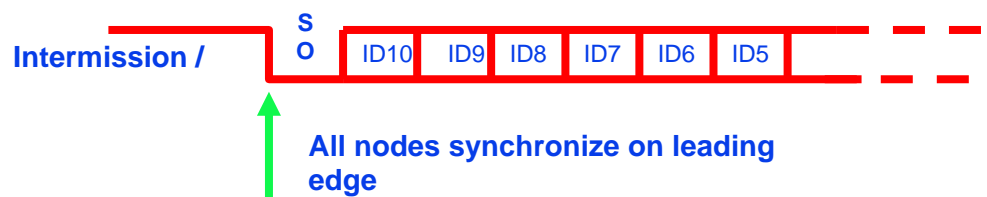
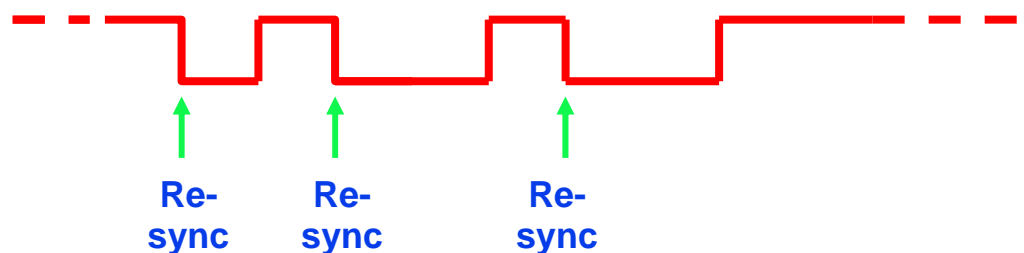


Figure 4-8. Re-Synchronization

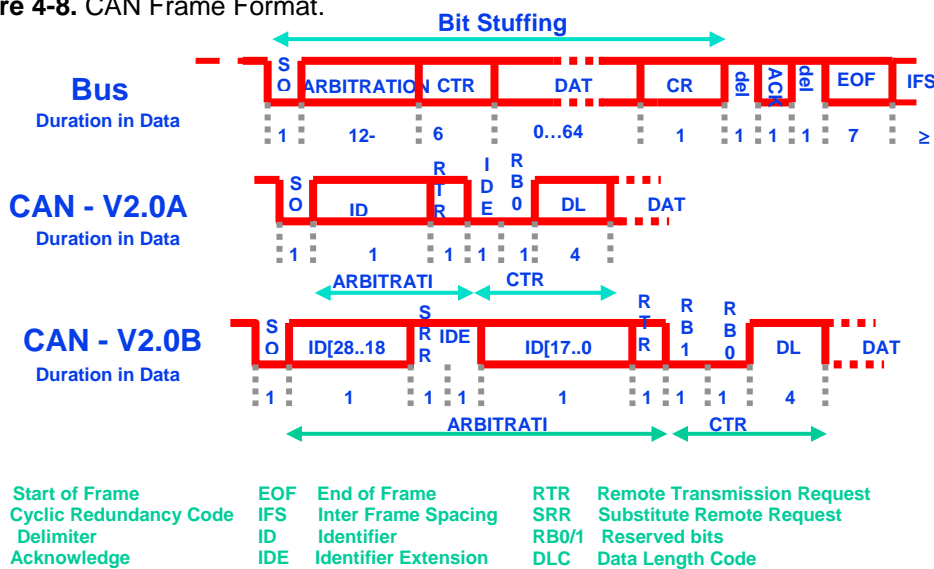


4.9 Frame Formats

The CAN frame format is split in several fields:

- Start of Frame,
- Arbitration with identifier field of 11 bis in 2.0A and 29bits in 2.0B,
- Control (CTR),
- Data.

Figure 4-8. CAN Frame Format.



4.10 Fault Confinement

There are three fundamental states define each node's error signaling:

- Error active: Normal state, node can send all frames incl.error frames
- Error passive: Node can send all frames excluding error frames
- Bus off: Node is isolated from bus

Internal error counts determine the state for each node:

- Transmit error counter (TEC) ,
- Receive error counter (REC).

An error increases the counter by 8 and a successful operation decreases by 1. This mechanism aims to prevent from bus dead-locks by faulty nodes

The CRC is calculated over the non-stuffed bit stream starting with the SOF and ending with the Data field by the transmitting node

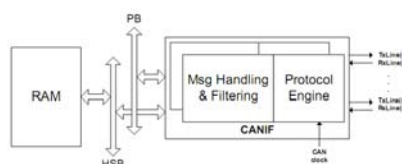
- The CRC is calculated again of the de-stuffed bit stream by the receiving node.
- A comparison of the received CRC and the calculated CRC is made by the receiver.
- In case of mismatch the erroneous data frame is discarded. Instead of sending an acknowledge signal an error frame is sent.

5 CANIF Module Overview

The CAN interface (CANIF) is a 32-bit interface for CAN controllers (channels). Each channel provides following services:

- Message filtering
- Message and status handling
- Fault confinement
- Error detection and signaling
- Message validation and acknowledgement
- Bus arbitration
- Message framing
- Transfer rate and timing

Except message filtering and message handling those services are described in CAN protocol, please refer to *Bosch - CAN Specification* for more details.



5.1 CAN Message Object Structure

The CAN Mailbox (Mob) is split in two fields:

- the control registers (control the CAN module and collect the status of CAN Bus),
- the data registers (contain the CAN frame)

The Mob can be even located in CPU Ram or in HSB Ram. The HSB Ram should be used in order to reduce contention between CPU and Peripheral accesses and optimize performance of CAN access.

5.2 Operating Modes

The CANIF has three operating modes, selectable by the Channel Mode field (CANCFG.CMODE):

- Normal mode (CANCFG.CMODE=00)

Default mode, TX and RX lines are connected to the transceiver. This mode is used to communicate with other nodes on the bus.

- Listening mode (CANCFG.CMODE=01)

The TX line is disconnected from the transceiver. The CAN channel cannot send any message nor acknowledge when a message has been received. The channel is in Error Passive mode and Transmit/Receive Error Counters (TEC/REC) are frozen. This mode is used to listen to CAN bus.

- Loop back mode (CANCFG.CMODE=10)

The TX line is internally connected to the RX line and disconnected from the transceiver. The CAN channel can only send messages or acknowledges to itself. The channel is in Error Passive mode and TEC/REC counters are frozen. This mode is used to detect the bit rate of the CAN bus by successive configuration of bit timing.



CAN clocks

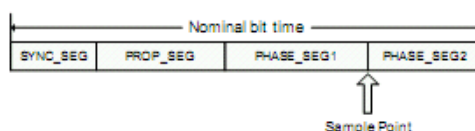
CANIF is connected to both the HSB and the PB, and therefore uses a HSB clock (CLK_CANIF_HSB) and a PB clock (CLK_CANIF_PB). These clocks are generated by the Power Manager, are enabled at reset, and can be disabled in the Power Manager.

CANIF uses a generic clock (GCLK) as clock source (CAN clock) for the CAN bus communication (GCLK_CANIF). User must make sure this clock is running and frequency is correct before any operation.

5.4 CAN Bit Timing

The CAN bit rate is defined by the nominal bit time. Nominal bit time is divided into 4 time segments.

Figure 5-1. Partition of bit Time.



The duration of each time segment is divided into time quanta (TQ). The total number of TQ in a bit time must be in the range [8..25].

The Time Quantum is a fixed unit of time derived from the GCLK_CANIF clock period:

$$TQ = \text{Prescaler} \times GCLK_CANIF = (\text{CANCFG.PRES} + 1) \times GCLK_CANIF$$

Re-synchronization may lengthen or shorten the bit time, the upper bound is given by Synchronization Jump Width field in the Configuration Register (CANCFG.SJW).

The value of all previous parameters are defined in CANCFG register.

Table 5-1. CAN Parameter Settings

Parameter	Range	CANCFG field
SYNC_SEG	1	-
PROP_SEG	[1..8]TQ	PRS+1
PHASE_SEG1	[1..8]TQ	PHS1 + 1
PHASE_SEG2	[1..8]TQ	PHS2 + 1
Prescaler	[2..32]	PRES + 1
Sync Jump	[1..4]	SJW + 1

The bit duration is given by the formula:

$$T_{bit} = (PRS + PHS1 + PHS2 + 4) \times (PRES + 1) \times PGCLK_CANIF$$

Note: PRES should not be set to 0, therefore CAN clock is at least divided by 2.

5.5 Acceptance Filter

The filtering process uses the ID tag (IDT) and ID mask (IDM) values defined in RAM. Comparison is done on the bits IDENTIFIER, RTR and IDE. Messages can therefore be filtered according to the identifier value, frame type (remote or data frame) and the format (standard or extended).

Each received bit is compared with the corresponding bit in the ID tag only if the corresponding bit in ID mask is set. Otherwise the received bit is considered as don't care. The filtering result is true if all comparisons are true.

Examples with 11 bits of identifier ('-' means don't care):

ID received: 000.0010.1001 b 000.0010.1001 b

IDT: 000.0010.1010 b 000.0100.1000 b

IDM: 111.1111.0000 b 111.1111.0000 b

Comparison: 111.1111.- - - b 111.1001.- - - b

Accepted: Y N

The filtering process scans each MOB enabled and configured for reception, from MOB 0, in order to find the MOB that matches the conditions. The first MOB to match is selected for storing the message once received successfully. If no MOB matches, the message is discarded.

5.6 CAN Interrupt Service Structure

There are several sources of interrupts and user can mask each of them. Some sources are grouped into a single interrupt request line. There are 5 interrupt request lines per channel.

- Wake-up interrupt: Wake-up condition detected
- Error interrupt: Any CAN error detected during a communication
- Bus off interrupt: The CAN protocol engine entered in bus off state
- Took interrupt: At least one MOB completed a transmission
- Waxed interrupt: At least one MOB completed a reception

The CANIMR and MOBIMR are used for masking interrupts. These registers are read-only. In

order to set or clear interrupt mask bits, user must write to the following registers:

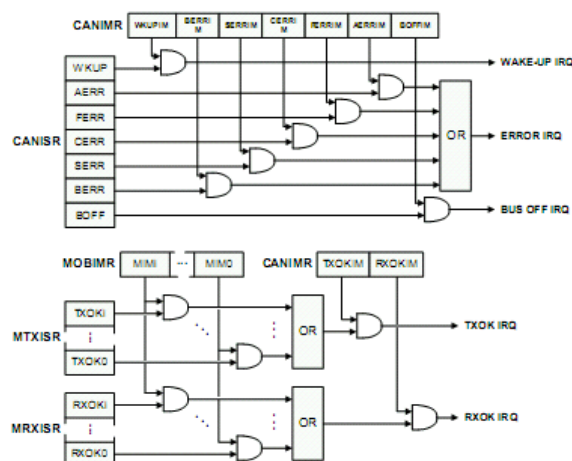
- CANIER / MOBIER: Writing a bit to one sets the corresponding bit in CANIMR / MOBIMR. Writing a bit to 0 has no effect.
- CANIDR / MOBIDR: Writing a bit to one clears the corresponding bit in CANIMR / MOBIMR. Writing a bit to 0 has no effect.

To acknowledge an interrupt request, user must clear the corresponding bit in the corresponding status register (CANISR, MTXISR or MRXISR). To clear status bits, user must access the following write-only registers:

- CANISCR / MTXISCR / MRXISCR: Writing a bit to one clears the corresponding bit in CANISR / MTXISR / MRXISR. Writing a bit to 0 has no effect.

For each MOB, the bits TXOK and RXOK are also accessible in MOBSCR register for clear access and MOBBSR register for read access.

Figure 5-2. Interrupt Channel Structure.








6 AVR UC3 CANIF software driver

The CAN source code is implemented following hardware resources and application point of view. It is available through the AVR Software Framework (ASF) delivered by Atmel (ASF) at <http://asf.atmel.no/readme.html>.

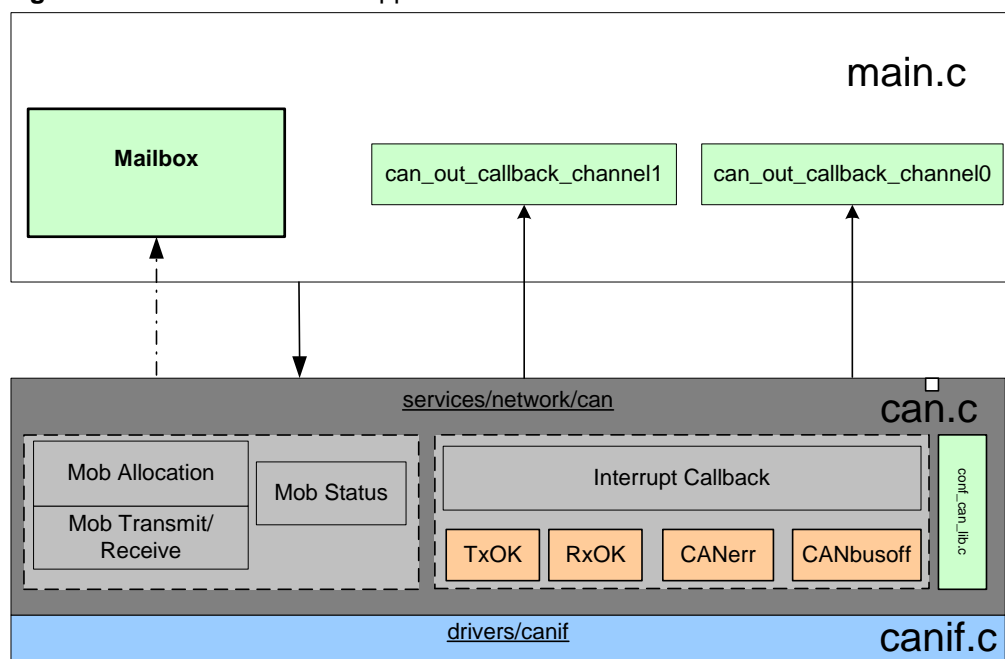
6.1 Files included in the AVR UC3 CAN source code

The CAN driver is split between:

- Two files that define a useful set of functions for the CANIF controller low level driver and provide an abstract layer to CANIF registers. It is located under the *drivers/canif* folder of the [ASF](#):
 -  `canif.c`: Low level driver source file
 -  `canif.h`: Low level driver header file
- Three files that define a useful set of functions for the CAN protocol management and configure it under the *services/network/can* folder of the [ASF](#):
 -  `can.c`: High level source file
 -  `can.h`: High level sheader file
 -  `conf_can.h`: High level configuration header file, especially for fixed bit timing definition.
- The mailbox definition can be defined anywhere but need to respect the following description for CAN message description (`can_msg_t`: `canif.h`) and MOB message description (`can_mob_t`: `can.h`).

6.2 CAN source code overview

Figure 6-1. CAN source code application





6.3 CANIF controller low level driver

The CANIF controller low level driver is based on a set of functions to have access to the hardware resources.

Table 6-1. Macros and declarations for CANIF controller low level driver located in canif.c/.h files

Functions/ Declarations	Category	Reference
CAN_set_phs1(ch, phs1)	Bit Timing	6.3.1
CAN_set_phs2(ch, phs2)	Bit Timing	6.3.1
CAN_set_pres(ch, pres)	Bit Timing	6.3.1
CAN_set_prs(ch, prs)	Bit Timing	6.3.1
CAN_set_sjw(ch, sjw)	Bit Timing	6.3.1
CAN_set_sm(ch, sm)	Bit Timing	6.3.1
CAN_set_channel_mode(ch, mode)	Operating Modes	6.3.2
#define CAN_CHANNEL_MODE_NORMAL	Operating Modes	6.3.2
#define CAN_CHANNEL_MODE_LISTENING	Operating Modes	6.3.2
#define CAN_CHANNEL_MODE_LOOPBACK	Operating Modes	6.3.2
CAN_set_overrun_mode(ch)	Operating Modes	6.3.2
CAN_set_reset(ch)	Channel Handling	6.3.3
CAN_clr_reset(ch)	Channel Handling	6.3.3
CAN_enable(ch)	Channel Handling	6.3.3
CAN_disable(ch)	Channel Handling	6.3.3
CAN_enable_interrupt(ch)	Channel Handling	6.3.3
CAN_disable_interrupt(ch)	Channel Handling	6.3.3
CAN_channel_enable_status(ch)	Channel Handling	6.3.3
CAN_send_overload(ch)	Channel Handling	6.3.3
CAN_get_interrupt_error_status(ch)	Channel Handling	6.3.3
CAN_get_error_mode(ch)	Channel Handling	6.3.3
CAN_get_tec(ch)	Channel Handling	6.3.3
CAN_get_rec(ch)	Channel Handling	6.3.3
can_msg_t;	Message Handling	6.3.4
CAN_set_idemask(ch, mob)	Message Handling	6.3.4
CAN_set_rtrmask(ch, mob)	Message Handling	6.3.4
CAN_set_ide(ch, mob)	Message Handling	6.3.4
CAN_set_rtr(ch, mob)	Message Handling	6.3.4
CAN_set_ext_id(ch, mob, _id)	Message Handling	6.3.4
CAN_set_std_id(ch, mob, _id)	Message Handling	6.3.4
CAN_set_data(ch, mob, _data)	Message Handling	6.3.4
CAN_clr_mob(ch, mob)	Message Handling	6.3.4
CAN_mob_set_dlc(ch, mob, dlc)	Message Handling	6.3.4
CAN_config_tx(ch, mob)	Message Handling	6.3.4

Functions/ Declarations	Category	Reference
CAN_config_rx(ch,mob)	Message Handling	6.3.4
CAN_mob_enable(ch,mob)	Message Handling	6.3.4
CAN_mob_disable(ch,mob)	Message Handling	6.3.4
CAN_mob_set_automode(ch,mob)	Message Handling	6.3.4
CAN_mob_enable_interrupt(ch,mob)	Message Handling	6.3.4

6.3.1 Bit Timing

The CAN_clock reference is based on a generic clock. Each time segments is accessible through the CANCFG register with C macro :

CAN_set_phs1(ch,phs1)
ch: Channel number 0 or 1
phs1: Value for phs1 bit field timing.

* This macro is used to access to PHASE_SEG1.

6.3.2 Operating Modes

The different accesses to operating modes are available through the C macro:

CAN_set_channel_mode(ch,mode)
ch: Channel number 0 or 1
mode: Mode Value.

Each mode is defined through the following #define:

- Normal Mode with CAN_CHANNEL_MODE_NORMAL,
- Listening Mode with CAN_CHANNEL_MODE_LISTENING,
- Loopback Mode with CAN_CHANNEL_MODE_LOOPBACK.

6.3.3 Channel Handling

The initialization and reset of can channel are accessible through C macro as:

CAN_set_reset(ch);
ch: Channel number 0 or 1

To enable a CAN channel, some C macro is used also as:

CAN_enable(ch);
ch: Channel number 0 or 1

6.3.4 Message Handling

6.3.4.1 Message Object Structure

Messages Objects (Mob) are messages descriptor, used to store and handle CAN frame. The message object structure is defined as:

```
typedef struct
{
    union{
        struct{
            volatile U32 id : 32;
            volatile U32 id_mask : 32;
        };
        struct{
            volatile U32 : 1;
            volatile U32 rtr_bit : 1;
            volatile U32 ide_bit : 1;
            volatile U32 id_bit : 29;
            volatile U32 : 1;
            volatile U32 rtr_mask_bit : 1;
            volatile U32 ide_mask_bit : 1;
            volatile U32 id_mask_bit : 29;
        };
    };
    volatile Union64 data;
} can_msg_t;
```

Each field of `can_msg_t` is accessible through C macro. For example to access to the id field, this C macro can be used.

```
CAN_set_std_id(ch,mob,_id)
```

ch: Channel number 0 or 1

mob: Mob number from 0 to 31.

id: Identifier of the message

6.3.4.2 Transmission

In order to setup a Mob in transmission, the C macro can be used.

```
CAN_config_tx(ch,mob)
```

ch: Channel number 0 or 1

mob: Mob number from 0 to 31.

6.3.4.3 Reception

In order to setup a Mob in reception, the C macro can be used.

```
CAN_config_rx(ch,mob)
```

ch: Channel number 0 or 1

mob: Mob number from 0 to 31.

6.3.4.4 Automatic Mode

In order to setup a Mob in automatic mode, the C macro can be used.

```
CAN_set_automode(ch,mob)
```

ch: Channel number 0 or 1

mob: Mob number from 0 to 31.

6.3.4.5 Message Filtering

The message filtering is accessible through the field `id_mask` of the `can_msg_t`.

6.3.4.6 Interrupts

In order to enable interrupt for a Mob, the C macro can be used.

```
CAN_mob_enable_interrupt(ch,mob)
```

ch: Channel number 0 or 1

mob: Mob number from 0 to 31.



6.4 CAN Protocol management description

The CAN Protocol management source files are based on a set of functions to have access to high-level function as allocate Mob, transmit Mob...

Table 6-2. Functions/ Declarations list for CAN driver located in can.c/.h files.

Functions/ Declarations	Category	Reference
U8 can_init(U8 ch, U32 can_msg_ram_add, U8 operating_mode, void (*can_msg_callback)(U8,U8))	Can Initialization	6.4.1
U8 can_mob_alloc(U8 ch)	Mob Allocation	6.4.2
CAN_MOB_NOT_ALLOCATED	Mob Allocation	6.4.2
U8 can_mob_free(U8 ch,U8 handle)	Mob Allocation	6.4.2
U8 can_tx(U8 ch, U8 handle, U8 dlc, U8 req_type, const can_msg_t *can_msg)	Mob Transmit/ Receive	6.4.3
U8 can_rx(U8 ch, U8 handle, U8 req_type, const can_msg_t *can_msg)	Mob Transmit/ Receive	6.4.3
Union64 can_get_mob_data(U8 ch , U8 handle)	Mob Status	6.4.4
U8 can_get_mob_dlc(U8 ch , U8 handle)	Mob Status	6.4.4
U32 can_get_mob_id(U8 ch , U8 handle)	Mob Status	6.4.4
U8 can_mob_get_status(U8 ch, U8 handle)	Mob Status	6.4.4

6.4.1 CAN initialization

The initialization should define the address of the mob location, one of the three operating modes (CAN_CHANNEL_MODE_NORMAL, CAN_CHANNEL_MODE_LISTENING or CAN_CHANNEL_MODE_LOOPBACK) used and the callback functions for the return function called by the hardware interrupts (Mob interrupts and CAN interrupts).

6.4.2 Mob Allocation

```
typedef struct{  
  U8 handle;  
  can_msg_t *can_msg;  
  U8 dlc;  
  U8 req_type;  
  U8 status;  
}can_mob_t;
```

The CAN Mob object is defined to combine CAN messages with applicative properties:

- **handle**: this field define the number of message object currently in use, by default it is initialized with `CAN_MOB_NOT_ALLOCATED` value.
- ***can_msg**: pointer on a CAN message.
- **dlc**: datalength value.
- **req_type**: this field specify if it is a data frame request `CAN_DATA_FRAME` or remote request `CAN_REMOTE_FRAME`.
- **status**: status of a CAN message `CAN_STATUS_COMPLETED` or `CAN_STATUS_NOT_COMPLETED`.

6.4.3 Mob Transmit/ Receive

One function is available for transmit:

```
U8 can_tx( U8 ch,
           U8 handle,
           U8 dlc,
           U8 req_type,
           const can_msg_t *can_msg);
```

and another one for receive:

```
U8 can_rx( U8 ch,
           U8 handle,
           U8 req_type,
           const can_msg_t *can_msg);
```

6.4.4 Mob Status

The 2 status available for a Mob are `CAN_STATUS_NOT_COMPLETED` and `CAN_STATUS_COMPLETED`.

```
U8 can_mob_get_status(U8 ch,
                     U8 handle)

ch: Channel number 0 or 1

handle: number of message object currently in use.
```

and especially access to data,

```
Union64 can_get_mob_data( U8 ch ,
                          U8 handle)

ch: Channel number 0 or 1

handle: number of message object currently in use.
```

id and

```
U32 can_get_mob_id( U8 ch ,
                   U8 handle)

ch: Channel number 0 or 1

handle: number of message object currently in use.
```

dlc field.

```
U8 can_get_mob_dlc( U8 ch ,
                   U8 handle)

ch: Channel number 0 or 1

handle: number of message object currently in use.
```





6.4.5 Interrupt Callback

```
static struct
{
    void (*can_msg_callback_channel0) (U8,U8);
    void (*can_msg_callback_channel1) (U8,U8);
} can_lib_params =
{
    .can_msg_callback_channel0      = NULL,
    .can_msg_callback_channel1      = NULL
};
```

As the high levels of the CAN library have no direct access to the hardware, some callbacks are defined. These callbacks are called at every hardware interrupt with two parameters:

- handle: the number of the selected MOB that produced the event
- event: the name of the event produced (CAN_STATUS_COMPLETED, CAN_STATUS_ERROR, CAN_STATUS_WAKEUP and CAN_STATUS_BUSOFF)

7 Using the CAN library on the UC3C-EK

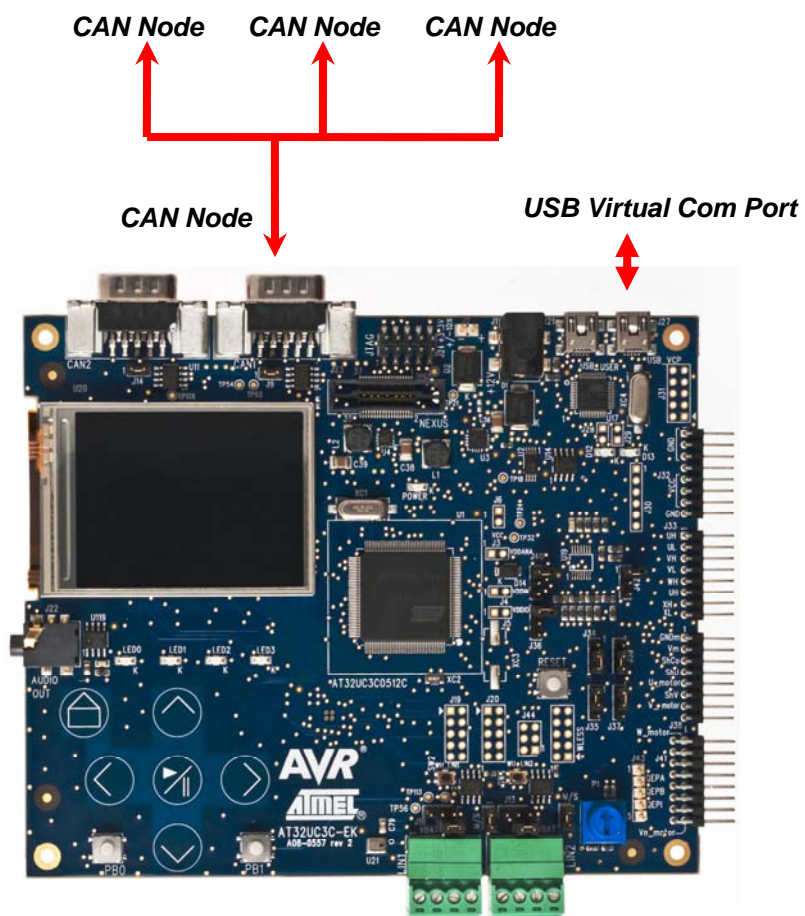
7.1.1 Purpose of the example

This example demonstrates that how to send/receive messages from the CAN using an UC3C-EK development kit. Refer to the [UC3C-EK Getting Started](#) to connect the CAN node and the USB Virtual Com Port to run the example.

7.1.2 Prerequisites

7.1.2.1 Hardware Setup

Figure 7-1. Connect the CAN node.



7.1.2.2 Source Code Location

The source code of the example described in this section is available in `services/network/can/example2/` folder of the [ASF](#).

7.1.2.3 Clock Settings

In order to enable CAN_clock frequency for baudrate generation, here is the following sequence to respect:





- Switch CPU clock to one of oscillator ('0' or '1'),
- Setup PLL if needed,
- Enable PLL if used,
- Setup Generic Clock with right predivisor and right clock input (oscillator or pll),
- Enable Generic Clock,
- Setup internal clock PBA (Register clock), PBB (Memory Clock), CPU and HSB clock with right predivisor.

7.1.2.4 Baudrate Configuration

The baudrate configuration is static. It is configurable through the `conf_can.h` file. A list of baudrate is available using the syntax:

`CAN_BAUDRATE_1MHZ Canclock_xxxMHz`, eg: `CAN_BAUDRATE_1MHZ Canclock_16MHz`

To assign the baudrate and all specific fields to the one chosen by the application, follow this example:

```
#define BAUDRATE          CAN_BAUDRATE_1MHZ_Canclock_16MHz
#define BAUDRATE_SJW      CAN_BAUDRATE_1MHZ_Canclock_16MHz_SJW
#define BAUDRATE_PRES     CAN_BAUDRATE_1MHZ_Canclock_16MHz_PRES
#define BAUDRATE_PRS      CAN_BAUDRATE_1MHZ_Canclock_16MHz_PRS
#define BAUDRATE_PHS1     CAN_BAUDRATE_1MHZ_Canclock_16MHz_PHS1
#define BAUDRATE_PHS2     CAN_BAUDRATE_1MHZ_Canclock_16MHz_PHS2
```

7.1.2.5 MOB Allocation

It is possible to allocate MOB at the memory area requested by the application:

- In CPU Ram

```
mob_ram_t mob_ram_ch0[NB_MOB0];
```

- In HSB Ram

```
__no_init mob_ram_t mob_ram_ch0[NB_MOB0] @ AVR32_HRAM_ADDRESS;
```

7.1.2.6 Mailbox Definition

In order to define a mailbox for CAN application, please check the `conf_can_example.h` file.

In this file, all CAN messages are defined with custom properties.

For example, in order to define a CAN message in transmission:

```
can_msg_t msg_tx_sot = {
    0x130, //id
    0x1ff, //id mask
    0x0102030405060708 // data
};
```


The CAN message object is then defined:

```
can_mob_t appli_tx_msg = {
    {
        CAN_MOB_NOT_ALLOCATED, //Handle
        &msg_tx_sot, // CAN Message
        8, // DLC
        CAN_DATA_FRAME, // Data Frame Type
        CAN_STATUS_NOT_COMPLETED // Status
    }
};
```

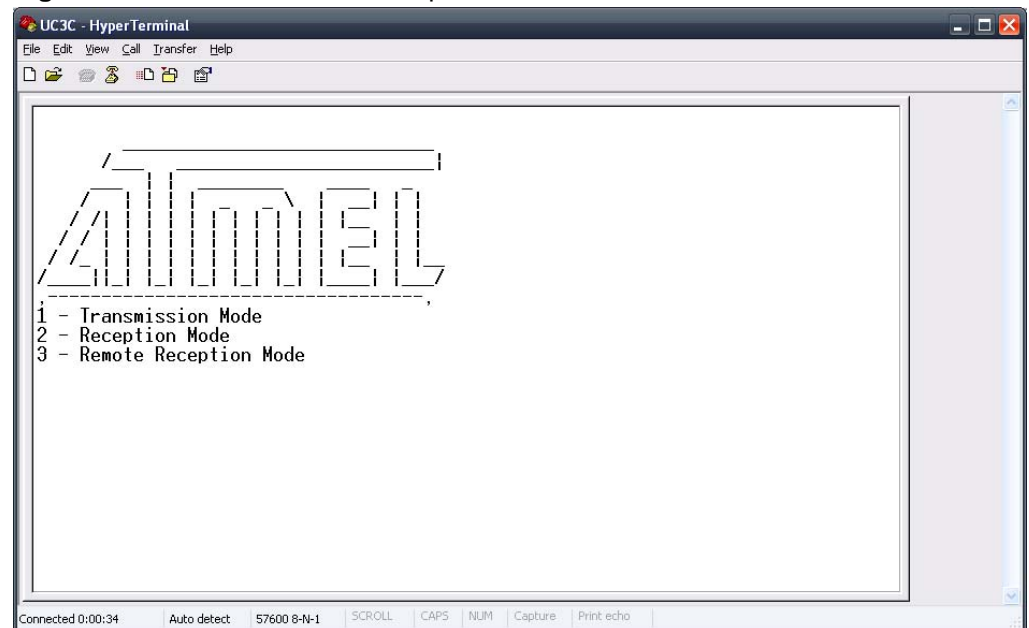
Here it means, one CAN message is initialized as a data frame with a dlc of 8. Note that as it is not allocated and not treated, it is initialized with the status CAN_MOB_NOT_ALLOCATED.

The other messages `appli_rx_msg` and `appli_remote_rx_msg` for the application are defined with the same rule.

7.1.3 Case 1: Normal Mode

In normal mode case, a list of message in transmission is previously defined. The list of message in reception that can be received is also defined.

Figure 7-2. Demo Hardware Startup Screen



Press '1' to send a message on the CAN bus



Figure 7-3. Normal Mode Startup Screen

```
com1_57600 - HyperTerminal
File Edit View Call Transfer Help
[Icons]

  /-----\
 /         \
|  Atmel  |
|         |
 \-----/

1 - Transmission Mode
2 - Reception Mode
3 - Remote Reception Mode

-0- TxCAN @ 307: 0x133(Std.), L=8, 01-02-03-04-05-06-07-08;

===Start demo now? Press y or Y - start
===                          Press n or N - Not start

=== OK: UC3C CAN Communication Success ===

Connected 0:56:29  ANSIW  57600 8-N-1  SCROLL  CAPS  NUM  Capture  Print echo
```

Once the message is correctly sent, press 'q' to return to the main menu.

Figure 7-4. Normal Mode Running

```
com1_57600 - HyperTerminal
File Edit View Call Transfer Help
[Icons]

  /-----\
 /         \
|  Atmel  |
|         |
 \-----/

1 - Transmission Mode
2 - Reception Mode
3 - Remote Reception Mode

===Start demo now? Press y or Y - start
===                          Press n or N - Not start
-1- RxCAN @ 307: 0x133, L=8, 0-0-0-0-0-0-0-0
-1- RxCAN @ 307: 0x133, L=8, 0-0-0-0-0-0-0-0
-1- RxCAN @ 307: 0x133, L=8, 0-0-0-0-0-0-0-0
-1- RxCAN @ 307: 0x133, L=8, 0-0-0-0-0-0-0-0
-1- RxCAN @ 307: 0x133, L=8, 0-0-0-0-0-0-0-0
-1- RxCAN @ 307: 0x133, L=8, 0-0-0-0-0-0-0-0
-1- RxCAN @ 307: 0x133, L=8, 0-0-0-0-0-0-0-0
-1- RxCAN @ 307: 0x133, L=8, 0-0-0-0-0-0-0-0

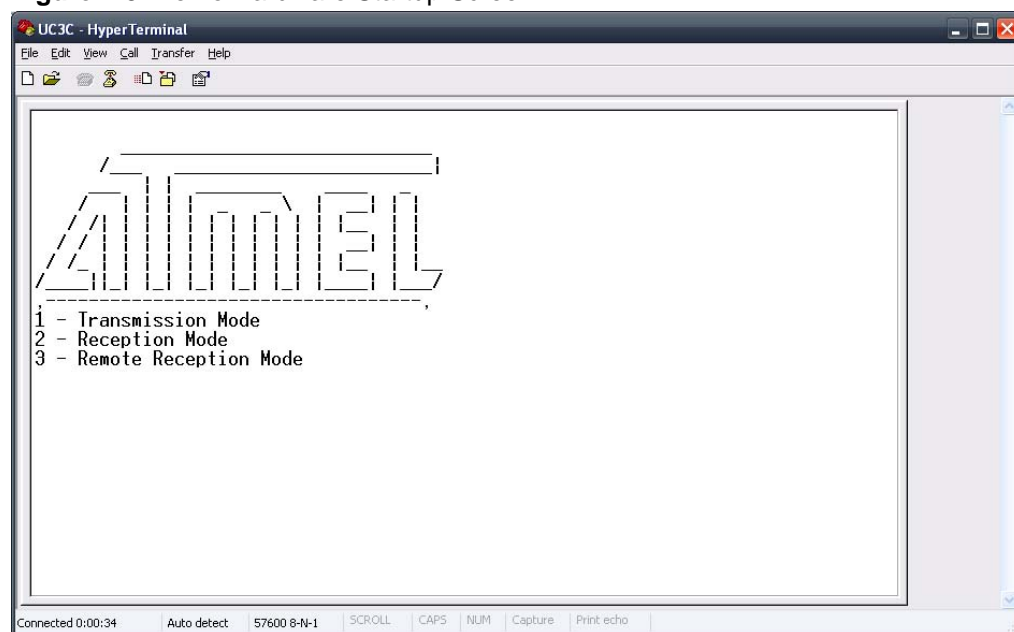
Connected 1:10:01  ANSIW  57600 8-N-1  SCROLL  CAPS  NUM  Capture  Print echo
```

Press button '2' to receive a message on CAN bus. The content of the message (ID, DLC and DATA) is also displayed.

7.1.4 Case 2: Remote Reception Mode

In Remote Reception mode, a CAN remote request is sent and the node 0 will reply to this message.

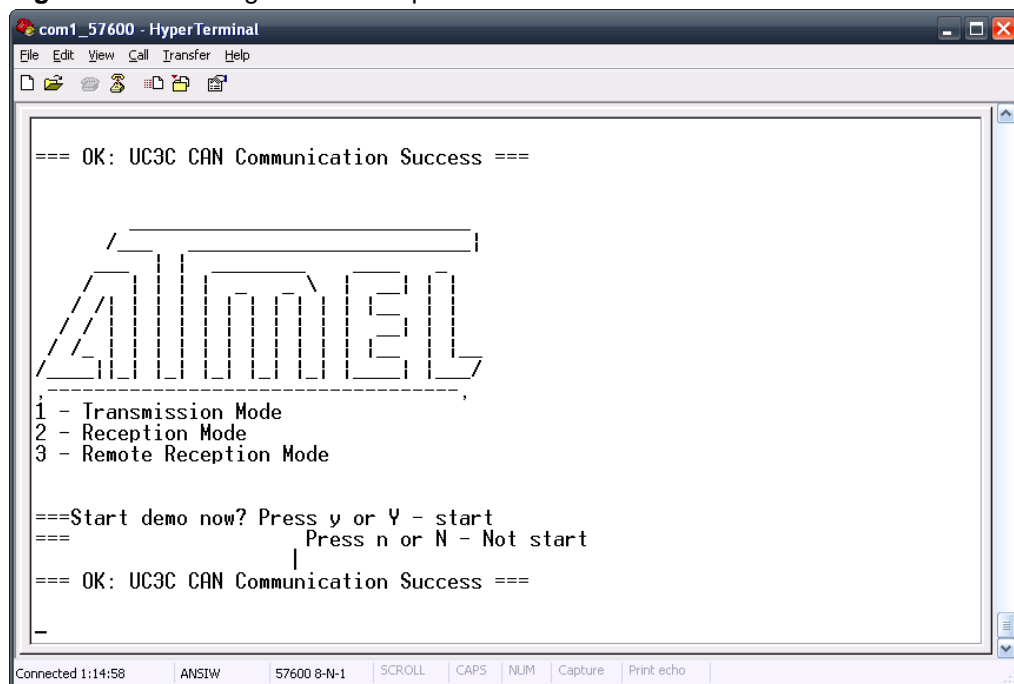
Figure 7-5. Demo Hardware Startup Screen



Press '3' to setup the a Remote Reception request on the CAN bus



Figure 7-6. Listening Mode Startup Screen





Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. **EXCEPT AS SET FORTH IN ATMEL'S TERMS AND CONDITIONS OF SALE LOCATED ON ATMEL'S WEB SITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.** Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel's products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

© 2010 Atmel Corporation. All rights reserved. Atmel®, Atmel® logo and combinations thereof, AVR®, AVR® logo and others, are the registered trademarks or trademarks of Atmel Corporation or its subsidiaries. Other terms and product names may be trademarks of others.