AVR076: AVR® CAN - 4K Boot Loader

1. Features

- CAN Protocol
  - Controller Area Network (CAN) used as Physical Layer
  - 7 re-programmable ISP CAN identifiers
  - Auto-bitrate
- In-System Programming
  - Read/Write Flash and EEPROM memories
  - Read Device ID
  - Full chip Erase
  - Read/Write configuration bytes
  - Security setting from ISP command
  - Remote application start command
- In-Application Programming
  - Up to 255 nodes
  - 16 Re-locatable Reserved Identifiers
- Application Programming Interface
  - Write Flash API (application section)
- Opened for Other Protocols :
  - LIN,
  - RS232,
  - SPI,
  - TWI,
  - ...
- Flip Interfacing

2. Description

This document describes the **“Slim” CAN boot loader** functionality as well as its protocol to efficiently perform operations on the on chip Flash & EEPROM memories.

This boot loader implements the “In-System Programming” (ISP). The ISP allows the user to program or re-program the microcontroller on-chip Flash & EEPROM memories without removing the device from the system and without the need of a pre-programmed application.

The CAN boot loader can manage a communication with an host through the CAN network. It can also access and perform requested operations on the on-chip Flash & EEPROM memories.

In-application programming feature is available to manage up to 255 CAN nodes.

A special entry (Flash API) is available for users.

8-bit AVR® Microcontroller

AT90CAN32
AT90CAN64
AT90CAN128

- - - - - - - - - -

ATmega16M1
ATmega32M1
ATmega32C1
ATmega64M1
ATmega64C1

4 Kbytes (Slim) CAN Boot Loader

Doc 8247A–CAN–08/09

3. Boot Loader Environment

The **CAN boot loader** is loaded in the “Boot Loader Flash Section” of the on-chip Flash memory. The boot loader size is less than 4K bytes, so the physical “Boot Loader Flash Section” only is half-full. The application program size must be lower or equal the “Application Flash Section” plus 4K bytes(c.f. Table 3-1 on page 2 and Table 3-2 on page 2).

Table 3-1. AT90CANxx Family - Memory Mapping (byte addressing)

Memory		AT90CAN128	AT90CAN64	AT90CAN32
FLASH	Size	128 K bytes	64 K bytes	32 K bytes
	Add. Range	0x00000 - 0x1FFFF	0x00000 - 0x0FFFF	0x00000 - 0x07FFF
“Application Flash Section”	Size	120 K bytes	56 K bytes	24 K bytes
	Add. Range	0x00000 - 0x1DFFF	0x00000 - 0xDFFF	0x00000 - 0x05FFF
“Boot Loader Flash Section”	Size	8 K bytes		
	Add. Range	0x1E000 - 0x1FFFF	0x0E000 - 0x0FFFF	0x06000 - 0x07FFF
“Boot Loader Reset Addresses” (1)	Small (1st) Boot	0x1FC00	0x0FC00	0x07C00
	Second Boot	0x1F800	0x0F800	0x07800
	Third Boot	0x1F000 (2)	0x0F000 (2)	0x07000 (2)
	Large (4th) Boot	0x1E000	0x0E000	0x06000
EEPROM	Size	4 K bytes	2 K bytes	1 K bytes
	Add. Range	0x0000 - 0x0FFF	0x0000 - 0x07FF	0x0000 - 0x03FF

Note: 1. The “Boot Loader Reset Address” depends on the fuse bits “BOOTSZ”.
Refer to the data sheet for more details on Flash memories (Flash, EEPROM, ...) behaviors.
2. CAN Boot Loader reset address.

Table 3-2. ATmegaxxM1/C1 Family - Memory Mapping (byte addressing)

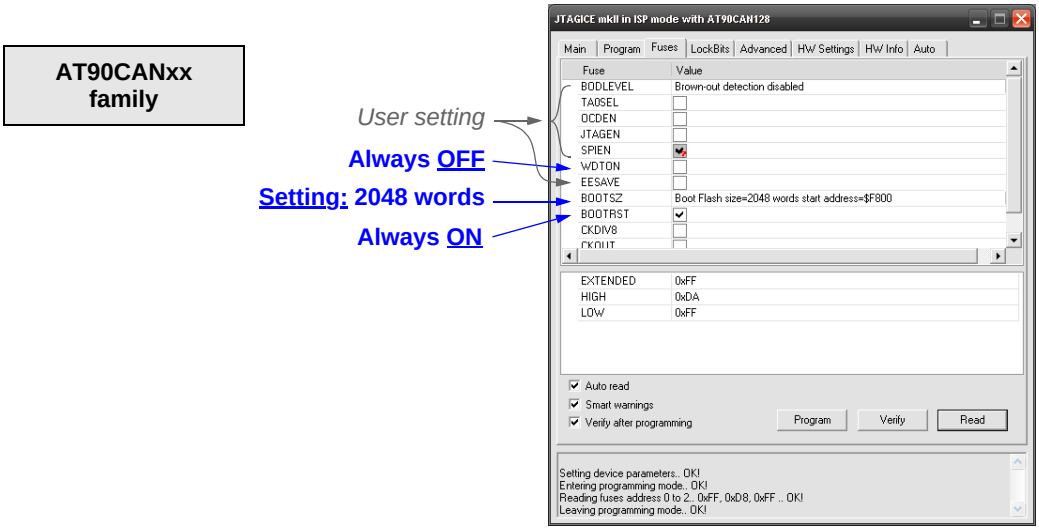
Memory		ATmega64M1/C1	ATmega32M1/C1	ATmega16M1
FLASH	Size	64 K bytes	32 K bytes	16 K bytes
	Add. Range	0x00000 - 0x0FFFF	0x00000 - 0x07FFF	0x00000 - 0x03FFF
“Application Flash Section”	Size	56 K bytes	28 K bytes	12 K bytes
	Add. Range	0x00000 - 0x0DFFF	0x00000 - 0x6FFF	0x00000 - 0x02FFF
“Boot Loader Flash Section”	Size	8 K bytes	4 K bytes	4 K bytes
	Add. Range	0x0E000 - 0x0FFFF	0x07000 - 0x07FFF	0x03000 - 0x03FFF
“Boot Loader Reset Addresses” (1)	Small (1st) Boot	0x0FC00	0x07E00	0x03E00
	Second Boot	0x0F800	0x07C00	0x03C00
	Third Boot	0x0F000 (2)	0x07800	0x03800
	Large (4th) Boot	0x0E000	0x07000 (2)	0x03000 (2)
EEPROM	Size	2 K bytes	1 K bytes	512 bytes
	Add. Range	0x0000 - 0x07FF	0x0000 - 0x03FF	0x0000 - 0x01FF

Note: 1. The “Boot Loader Reset Address” depends on the fuse bits “BOOTSZ”.
Refer to the data sheet for more details on Flash memories (Flash, EEPROM, ...) behaviors.
2. CAN Boot Loader reset address.

3.1 Device Fuse Setting

Please, refer to the device Data Sheet for further explanation.

Figure 3-1. Device Fuses Setting - Part 1



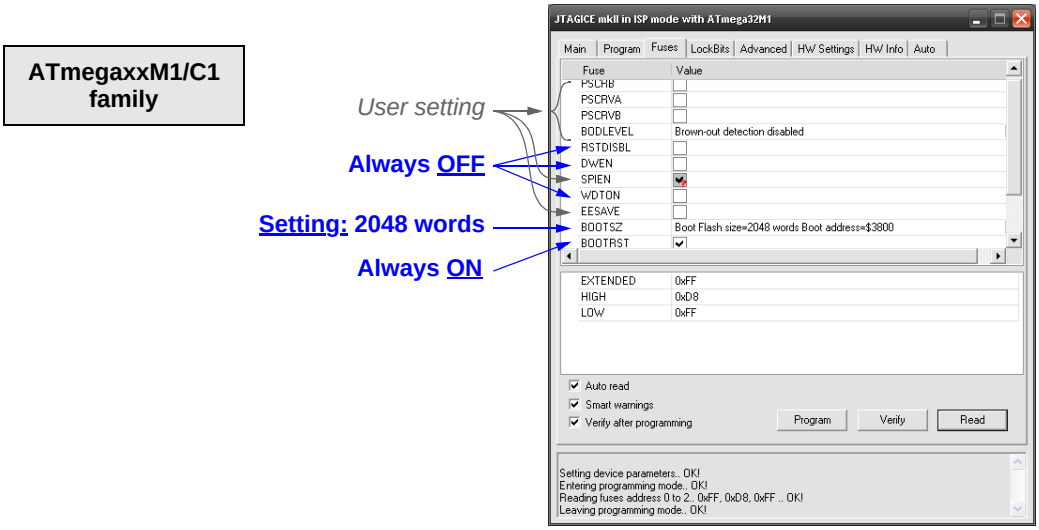
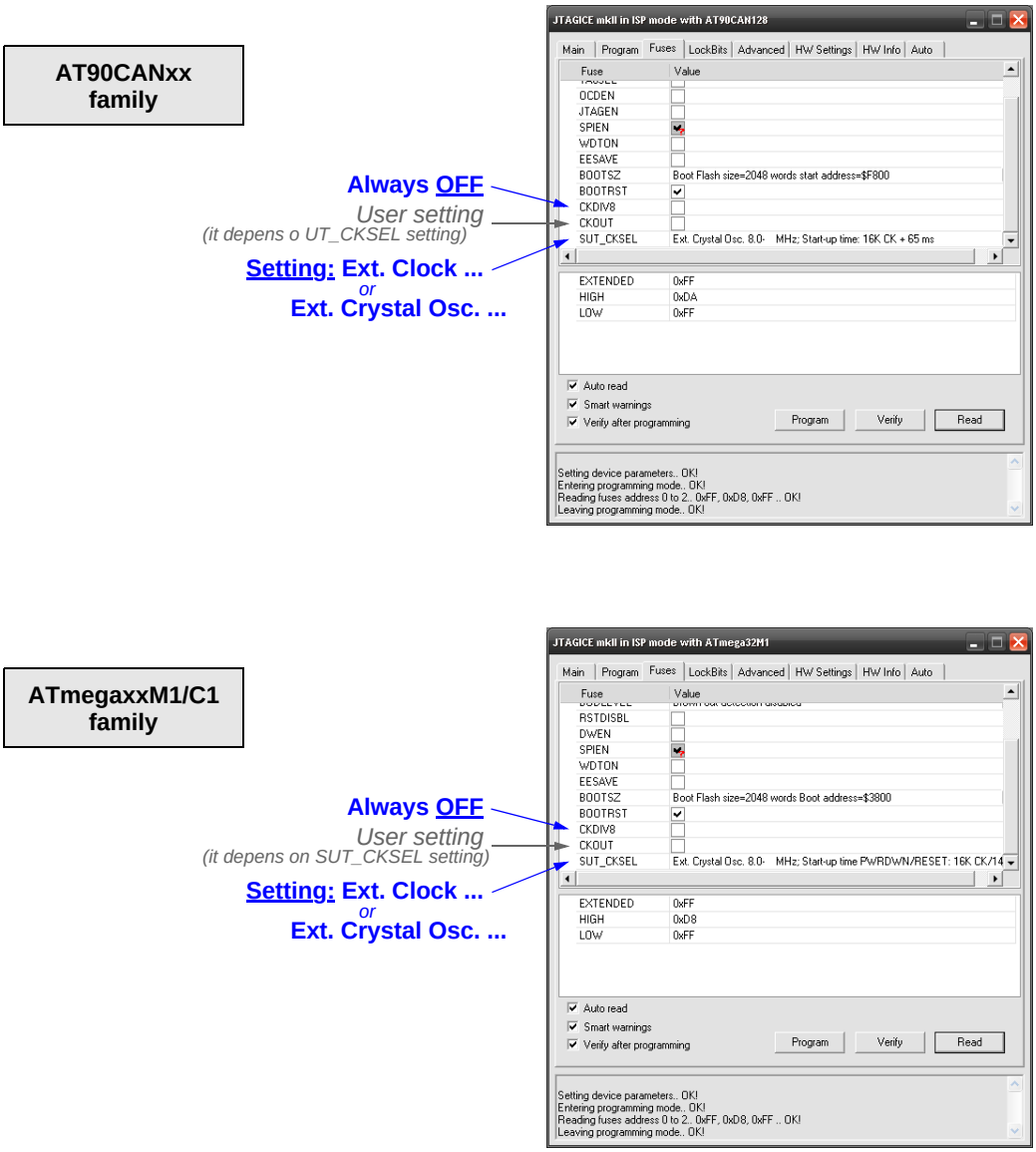


Figure 3-2. Device Fuses Setting - Part 2



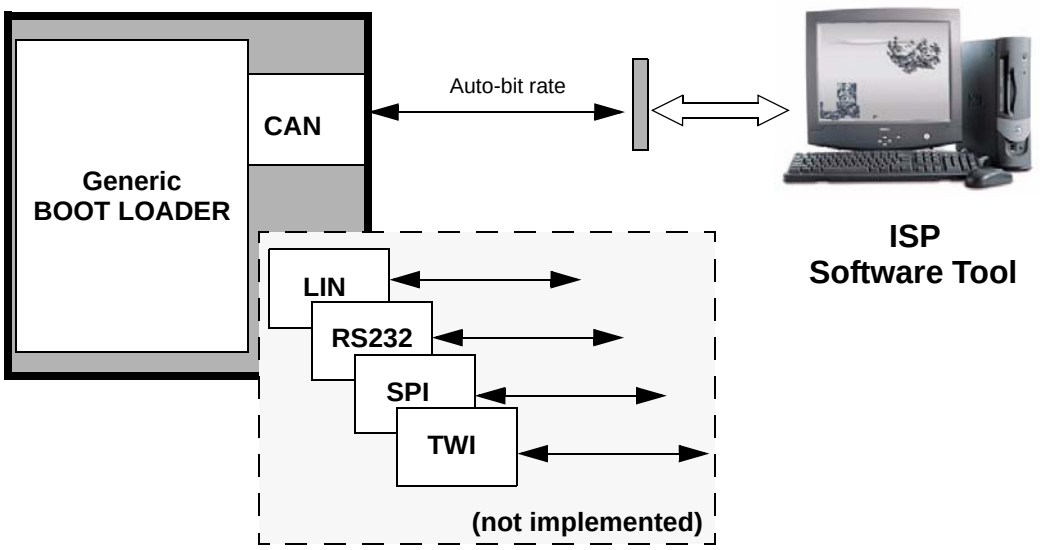
Summary of mandatory fuse setting:

- **Fuse High Byte:** - **BOOTRST** programmed,
  - **BOOTSZ [1:0]** programmed for **2048** words,
  - **WDTON** unprogrammed.
- **Fuse Low Byte:** - **CKSEL [3:0]** programmed to select a clock with an high accuracy to match with CAN requirement (the internal RC oscillator doesn't match).

3.2 Physical Environment

A generic boot loader deals with the host (or PC) through a CAN interface. The generic boot loader is a service able to be connected to other interfaces (LIN, RS232, SPI, TWI, ...).

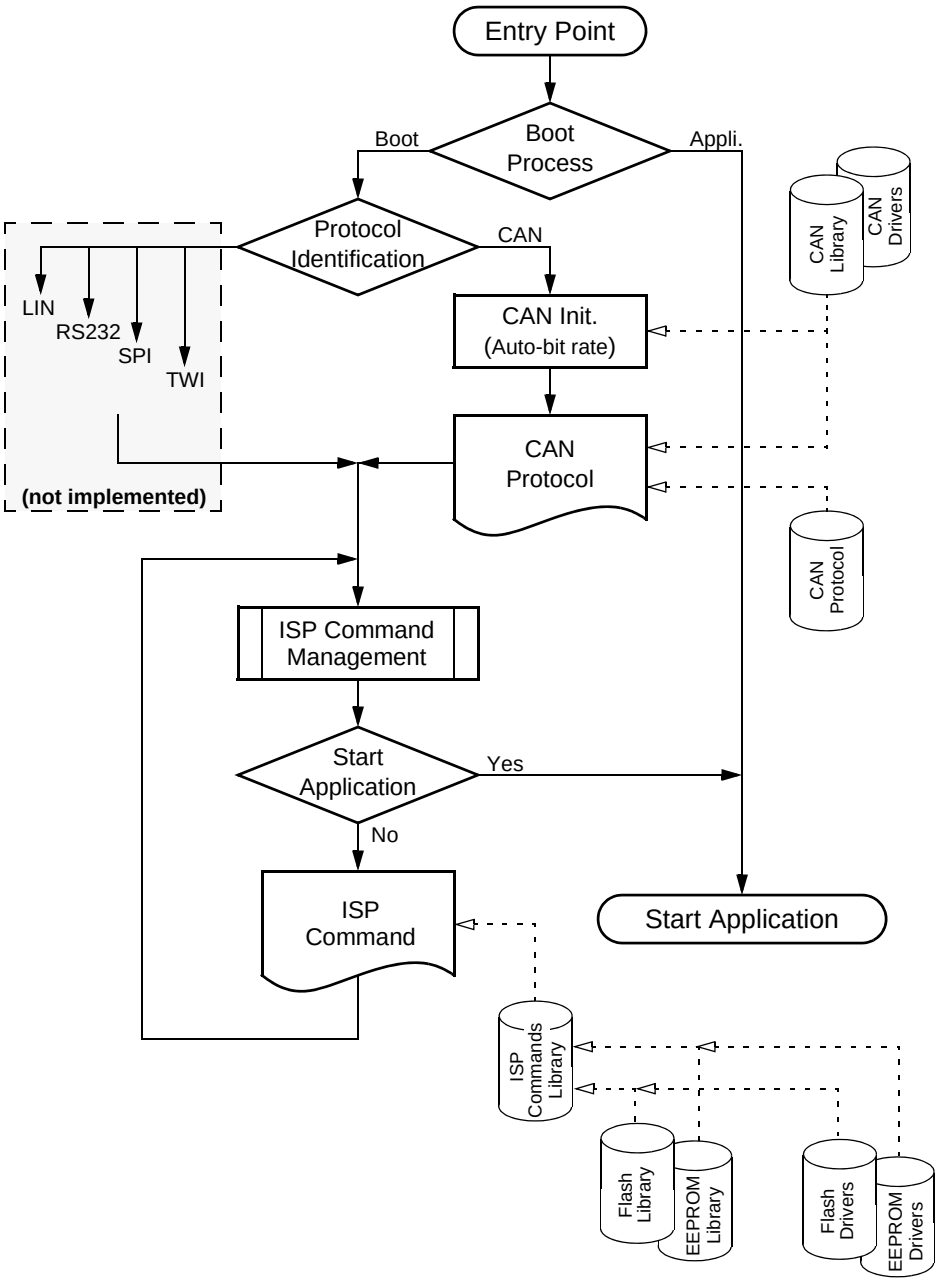
Figure 3-3. Physical Environment



3.3 Boot Loader Description

3.3.1 Overview

Figure 3-4. Boot Loader Diagram



3.3.2 Entry Point

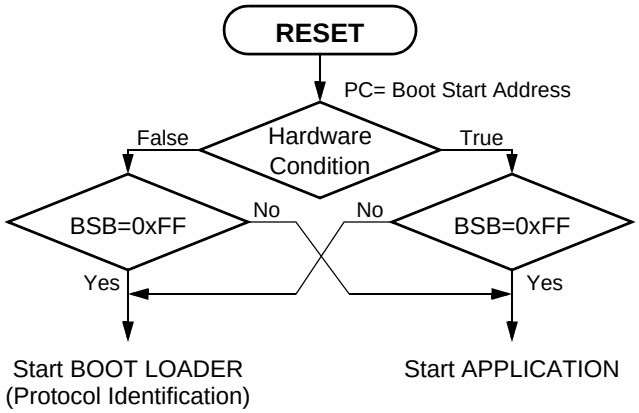
Only **one** "*Entry Point*" is available, it is the entry point to the boot loader. The "BOOTRST" fuse of the device have to be set. After Reset, the "Program Counter" of the device is set to "Boot Reset Address" (c.f. Table 3-1 "AT90CANxx Family - Memory Mapping (byte addressing)" on page 2 and Table 3-2 "ATmegaxxM1/C1 Family - Memory Mapping (byte addressing)" on page 2). This "*Entry Point*" initializes the "*boot process*" of the boot loader.

3.3.3 Boot Process

The "*boot process*" of the boot loader allows to start the application or the boot loader itself. This depends on two variables:

- The "**Hardware Condition**".
  The Hardware Condition is defined by a device input PIN (named HWCB in this boot loader) and its activation level (Ex: INT0/PIND.0, active low). It is set in the board header file.
- The "**Boot Status Byte**".
  The Boot Status Byte "**BSB**" belongs to the "Boot Loader Configuration Memory" (c.f. Section 4.5.4.1 "Boot Status Byte - "BSB"" on page 13). Its default value is 0xFF. An ISP command allows to change its value.

Figure 3-5. Boot Process Diagram



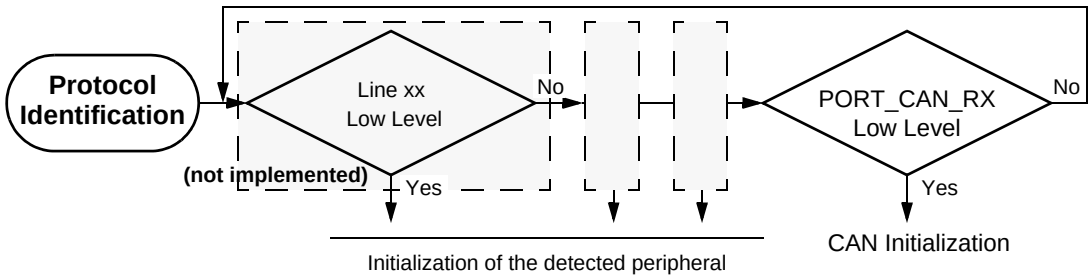
3.3.4 Protocol Identification

The "*Protocol Identification*" of the boot loader select what protocol to use, CAN or other protocol. A polling of the physical lines is done to detect an activity on the media. These lines are:

- **PORT_CAN_RX**: The polling is be done on:
  - *RXCAN/PIND.6* for AT90CANxx Family,
  - *RXCAN/PINC.3* for ATmegaxxM1/C1 Family,
- *(user defined interface)*.

A low level on **PORT_CAN_RX** line starts the initialization of the CAN peripheral.

Figure 3-6. Protocol Identification Diagram



3.3.5 CAN Initialization

The CAN, used to communicate with the host, has the following configuration:

– **Standard:** CAN format 2.0A (11-bit identifier).
– **Frame**: Data frame.
– **Bit rate**: Depends on Extra Byte - "**EB**" (see "Extra Byte - "EB"" on page 14):
  - "**EB**" = 0xFFH: Use the software auto-bit rate.
  - "**EB**" **!=** 0xFFH: Use Bit-Timing Control[1..3] bytes to set the CAN bit rate (see "Bit-Timing Control [1..3] - "BTC[1..3]"" on page 14).

The initialization process must be performed after each device Reset. The host initiates the communication by sending a data frame to select a node. In case of auto-bitrate, this will help the boot loader to find the CAN bitrate. The CAN standard says that a frame having an acknowledge error is re-sent automatically. This feature and the capability of the CAN peripheral to be set in "LISTEN" mode are used by the auto-bitrate. Once the synchronization frame is received without any error, a recessive level is applied on the acknowledge slot by releasing the "LISTEN" mode.

The software auto-bit rate supports a wide range of baud rates according with the system clock (CKIO) set on the device (c.f. "**FOSC**" definition in "config.h " file). This functionality (auto-bit rate) is not guaranteed on a CAN network with several CAN nodes. A fixed baud-rate ("**EB**" **!=** 0xFFH) is recommended in this case.

3.3.6 CAN Protocol Overview

The "*CAN Protocol*" is an higher level protocol over serial line (CAN Bus).

It is described in specific paragraphs in this document (See "CAN Protocol & ISP Commands" on page 17.).

3.3.7 ISP Commands Overview

The "*CAN Protocol*" decodes "*ISP commands*". The set of "*ISP commands*" obviously is independent of any protocol.

It is described in a specific paragraph in this document (See "CAN Protocol & ISP Commands" on page 17.).

3.3.8 Output From Boot Loader

The output from the boot loader is performs after receiving the ISP command: "*Start Application*" (See "CAN Protocol & ISP Commands" on page 17.).

4. Memory Space Definition

The boot loader supports up to six (**6**) separate memory spaces. Each of them receives a code number (value to report in the corresponding protocol field) because low level access protocols (drivers) can be different.

The access to memory spaces is a byte access (i.e. the given addresses are byte addresses).

Table 4-1. Memory Space Code Numbers

Space [1]	Code Number	Access
Flash Memory	**0**	Read & Write
EEPROM Data Memory	**1**	Read & Write
Signature	**2**	Read only
Boot Loader Information	**3**	Read only
Boot Loader Configuration	**4**	Read & Write
Device registers	**5**	Read only

Note: 1. Sometimes, the discriminating is not physical (ex: “Signature” is a sub-set of the code of the boot loader Flash Section” as well as “Boot Loader Information”).

4.1 Flash Memory Space

The Flash memory space managed by the boot loader is a sub-set of the device Flash. It is the “*Application Flash Section*”.

Table 4-2. Flash Memory Space (Code Number **0**)

Flash Memory Space	AT90CAN128	AT90CAN64 ATmega64M1 ATmega64C1	AT90CAN32 ATmega32M1 ATmega32C1	ATmega16M1
Size	124 K bytes	60 K bytes	28 K bytes	12 K bytes
Address Range	0x00000 - 0x1EFFF	0x0000 - 0xEFFF	0x0000 - 0x6FFF	0x0000 - 0x2FFF
Number of page(s)[1]	2	1	1	1

Note: 1. Page parameter is different in the boot loader and in the device itself.

4.1.1 Reading or Programming

The “*ISP Read*” or “*ISP Program*” commands only access to Flash memory space in byte addressing mode into a page of 64K bytes (c.f. Table 4-2 ”Flash Memory Space (Code Number 0)” on page 9). Specific ISP commands allows to select the different pages.

The boot loader will return a “*Device protection*” error if the Software Security Byte “**SSB**” is set while read or write command occurs (c.f. Section 4.5.4.2 ”Software Security Byte - “SSB”” on page 13).

4.1.2 Erasing

The “*ISP Erase*” command is a full erase (all bytes=0xFF) of the Flash memory space. This operation is available whatever the Software Security Byte “**SSB**” setting. A the end of the operation, the Software Security Byte “**SSB**” is reset to level 0 of security (Section 4.5.4.2 ”Software Security Byte - “SSB”” on page 13).

4.1.3 Limits

The ISP commands on the Flash memory space has no effect on the boot loader (no effect on "*Boot Loader Flash Section*").

The sizes of the Flash memory space (code number **0**) for ISP commands are given in Table 4-2 "Flash Memory Space (Code Number 0)" on page 9.

4.2 EEPROM Data Memory

The EEPROM data memory space managed by the boot loader is the device EEPROM.

Table 4-3. EEPROM Data Memory Space (Code Number **1**)

EEPROM Data Memory Space	AT90CAN128	AT90CAN64 ATmega64M1 ATmega64C1	AT90CAN32 ATmega32M1 ATmega32C1	ATmega16M1
Size	4 K bytes	2 K bytes	1 K bytes	512 bytes
Address Range	0x0000 - 0x0FFF	0x0000 - 0x07FF	0x0000 - 0x03FF	0x0000 - 0x01FF
Number of page(s)	-- No paging --			

4.2.1 Reading or Programming

The EEPROM data memory space is used as non-volatile data memory. The "*ISP Read*" or "*ISP Program*" commands access byte by byte to this space (no paging).
The boot loader will return a "*Device protection*" error if the Software Security Byte "**SSB**" is set while read or write command occurs (c.f. Section 4.5.4.2 "Software Security Byte - "SSB"" on page 13).

4.2.2 Erasing

The "*ISP Erase*" command is a full erase (all bytes=0xFF) of the EEPROM Data Memory space. This operation is available whatever only if the Software Security Byte "**SSB**" is reset (Section 4.5.4.2 "Software Security Byte - "SSB"" on page 13).

4.2.3 Limits

The sizes of the EEPROM Data Memory space (code number **1**) for ISP commands are given in Table 4-3 "EEPROM Data Memory Space (Code Number 1)" on page 10.

4.3 Signature

The Signature space managed by the boot loader is included the code of the boot loader. It is in the "*Boot Loader Flash Section*".

Table 4-4. Signature Space (Code Number **2**)

Signature Space		AT90CAN128	AT90CAN64	AT90CAN32	ATmega64M1	ATmega32M1	ATmega16M1	ATmega64C1	ATmega32C1
Manufacturer Code	Address: 0x00 (Read only)	0x1E							
Family Code	Address: 0x01 (Read only)	0x81			0x84			0x86	
Product Name	Address: 0x02 (Read only)	0x97	0x96	0x95	0x96	0x95	0x94	0x96	0x95
Product Revision	Address: 0x03 (Read only)	≥ 0x00							
Number of page(s)		-- No paging --							

4.3.1 Reading or Programming

The “I*SP Read*” command accesses byte by byte to this space (no paging).

No access protection is provided on this read only space.

4.3.2 Erasing

Not applicable for read only space.

4.3.3 Limits

Details on the Signature space (code number 2) for ISP commands are given in Table 4-4 ”Signature Space (Code Number 2)” on page 11.

4.4 Boot Loader Information

The Boot loader information space managed by the boot loader is included the code of the boot loader. It is in the “*Boot Loader Flash Section*”.

Table 4-5. Boot Loader Information Space (Code Number **3**)

Signature Space		AT90CAN128 AT90CAN64 AT90CAN32	ATmega64M1/C1 ATmega32M1/C1 ATmega16M1
Bootloader Revision	Address: 0x00 (Read only)	□ 0x01	
Boot ID1	Address: 0x01 (Read only)	0xD1	
Boot ID2	Address: 0x02 (Read only)	0xD2	
Number of page(s)		-- No paging --	

4.4.1 Reading or Programming

The “*ISP Read*” command accesses byte by byte to this space (no paging).
No access protection is provided on this read only space.

4.4.2 Erasing

Not applicable for this read only space.

4.4.3 Limits

Details on the Boot loader information space (code number **3**) for ISP commands are given in Table 4-5 ”Boot Loader Information Space (Code Number 3)” on page 11.

4.4.4 Boot Loader Information Byte Description

4.4.4.1 Boot Revision

Boot Revision: Read only address =0x00, value □ 0x01.

4.4.4.2 Boot ID1 & ID2

Boot ID1 & ID2: Read only addresses = 0x01 & 0x02, value = 0xD1 & 0xD2.

4.5 Boot Loader Configuration

The Boot loader configuration space managed by the boot loader is included in the “*Boot Loader Flash Section*”.

Table 4-6. Boot Loader Configuration Space (Code Number **4**)

Signature Space			Default value
Boot Status Byte	“**BSB**”	Add.: 0x00	0xFF
Software Security Byte	“**SSB**”	Add.: 0x01	0xFF
Extra Byte	“**EB**”	Add.: 0x02	0xFF (1)
Bit-Timing Control 1	“**BTC1**”	Add.: 0x03	0xFF (2)
Bit-Timing Control 2	“**BTC2**”	Add.: 0x04	0xFF (2)
Bit-Timing Control 3	“**BTC3**”	Add.: 0x05	0xFF (2)
Node Number	“**NNB**”	Add.: 0x06	0xFF (3)
CAN Re-locatable ID Segment	“**CRIS**”	Add.: 0x07	0x00
Start Address Low	“**SA_L**”	Add.: 0x08	0x00
Start Address High	“**SA_H**”	Add.: 0x09	0x00
Number of page(s)			-- No paging --

Note: 1. See “Extra Byte - “EB”” on page 14. for validity.
2. See “Bit-Timing Control [1..3] - “BTC[1..3]”” on page 14. for validity.
3. See “(CAN) Node Number - “NNB”” on page 14. for validity.

4.5.1 Reading or Programming

The “*ISP Read*” command accesses byte by byte to this space (no paging).

Access protection is only provided on the Software Security Byte (c.f. Section 4.5.4.2 ”Software Security Byte - “SSB”” on page 13).

4.5.2 Erasing

The “*ISP Erase*” command is **not available** for this space.

4.5.3 Limits

Details on the Boot loader configuration space (code number **4**) for ISP commands are given in Table 4-6 ”Boot Loader Configuration Space (Code Number 4)” on page 12.

4.5.4 Boot Loader Configuration Byte Description

*4.5.4.1 Boot Status Byte - "**BSB**"*

The Boot Status Byte of the boot loader is used in the "*boot process*" (Section 3.3.3 "Boot Process" on page 7) to control the starting of the application or the boot loader. If no Hardware Condition is set, the default value (0xFF) of the Boot Status Byte will force the boot loader to start. Else (Boot Status Byte != 0xFF & no Hardware Condition) the application will start.

*4.5.4.2 Software Security Byte - "**SSB**"*

The boot loader has the Software Security Byte "**SSB**" to protect itself and the application from user access or ISP access. It protects the Flash and EEPROM memory spaces and itself.

The "I*SP Program*" command on Software Security Byte "**SSB**" can only write an higher priority level. There are three levels of security:

Table 4-7. Security levels

Level	Security	"SSB"	Comment
0	**NO_SECURITY**	0xFF	- This is the default level. - Only level 1 or level 2 can be written over level 0.
1	**WR_SECURITY**	0xFE	- In level 1, it is impossible to write in the Flash and EEPROM memory spaces. - The boot loader returns an error message. - Only level 2 can be written over level 0.
2	**RD_WR_SECURITY**	≤ 0xFC	- All read and write accesses to/from the Flash and EEPROM memory spaces are not allowed. - The boot loader returns an error message. - Only an "*ISP Erase*" command on the Flash memory space resets (level 0) the Software Security Byte.

The table below gives the authorized actions regarding the SSB level.

Table 4-8. Allowed actions regarding the Software Security Byte "**SSB**"

ISP Command	NO_SECURITY	WR_SECURITY	RD_WR_SECURITY
Erase **Flash** memory space	Allow	Allow	Allow
Erase **EEPROM** memory space	Allow	-	-
Write **Flash** memory space	Allow	-	-
Write **EEPROM** memory space	Allow	-	-
Read **Flash** memory space	Allow	Allow	-
Read **EEPROM** memory space	Allow	Allow	-
Write byte(s) in **Boot loader configuration** (except for "**SSB**")	Allow	-	-
Read byte(s) in **Boot loader configuration**	Allow	Allow	Allow
Write "**SSB**"	Allow	only a higher level	-
Read **Boot loader information**	Allow	Allow	Allow
Read **Signature**	Allow	Allow	Allow
Blank check (any memory)	Allow	Allow	Allow
Changing of memory space	Allow	Allow	Allow

4.5.4.3 *Extra Byte - "**EB**"*

The Extra Byte is used to switch the CAN Initialization to auto-bitrate or to fixed CAN bit timing.

- "**EB**" = 0xFFH: Use the software auto-bitrate.
- "**EB**" **!=** 0xFFH: Use CANBT[1..3] bytes of Boot loader configuration space to set the CAN bit timing registers of the CAN peripheral (no auto-bit rate).

4.5.4.4 *Bit-Timing Control [1..3] - "**BTC[1..3]**"*

When "**EB**" **!=** 0xFFH, Bit-Timing Control[1..3] bytes ("**BTC1**", "**BTC2**" & "**BTC3**") of Boot loader configuration space are used to set the CAN Bit-Timing Registers of the CAN peripheral - no auto-bit rate.

A way to setup these bytes is described in Section 4.6.4.1 "CANBT[1..3] Registers" on page 16.

4.5.4.5 *(CAN) Node Number - "**NNB**"*

See "CAN Protocol & ISP Commands" on page 17.

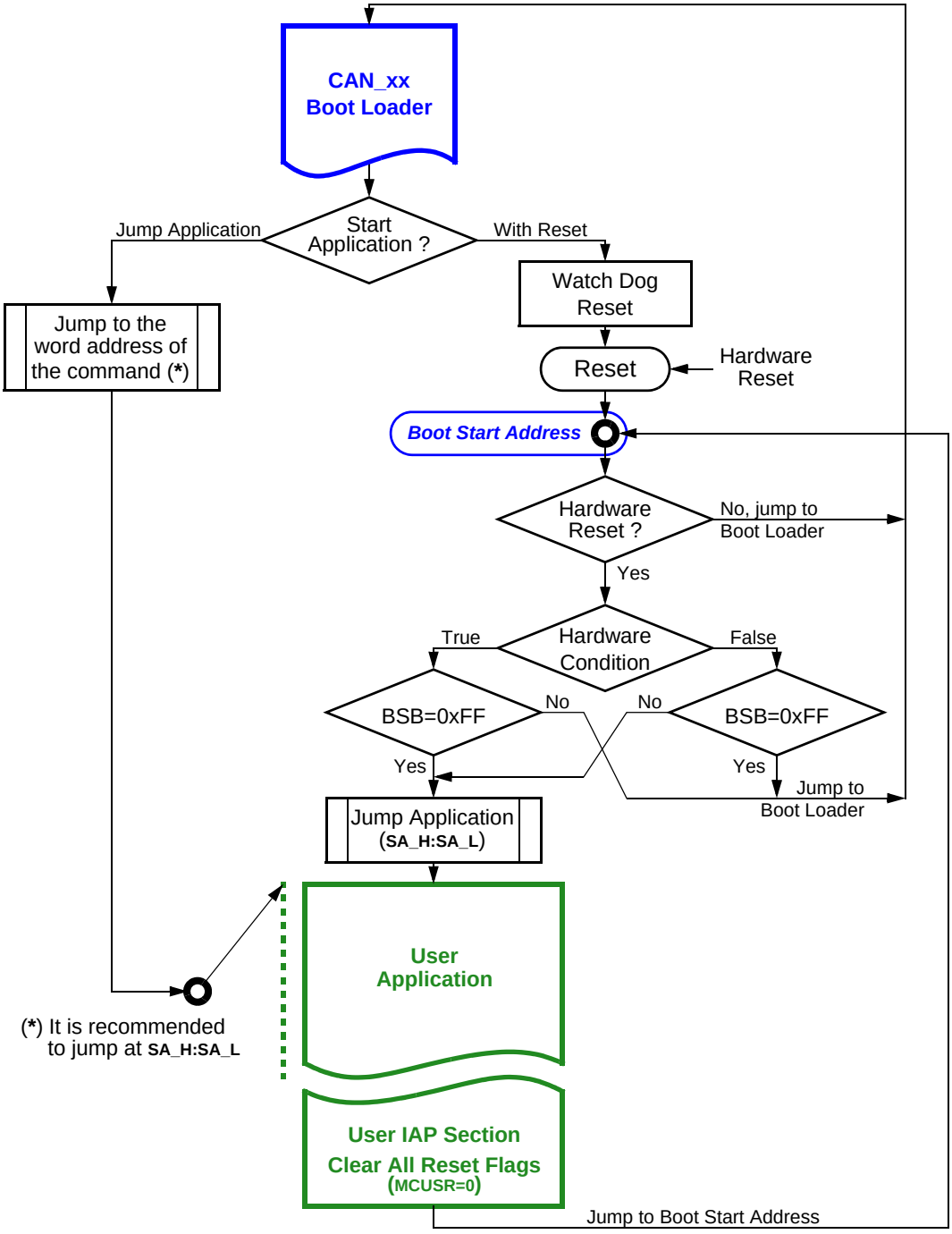
4.5.4.6 *CAN Re-locatable ID Segment - "**CRIS**"*

See "CAN Protocol & ISP Commands" on page 17.

4.5.4.7 *Start (application) Address High & Low- "**SA_H**" & "**SA_L**"*

See "CAN Protocol & ISP Commands" on page 17.

Figure 4-1. Start Application & Reset Diagram



4.6 Device Registers

The device registers space (Code Number **5**) managed by the boot loader is the 64 I/O registers and the 160 Ext. I/O registers of the device. They are accessed by the equivalent assembler instruction:

LDS Rxx, REG_ADD

where **REG_ADD** is in the address range:

- 0x20 (**PINA**) up to 0xFA (**CANMSG**) for AT90CANxx Family,
- 0x23 (**PINB**) up to 0xFA (**CANMSG**) for ATmegaxxM1/C1 Family.

4.6.1 Reading or Programming

The "*ISP Read*" command accesses byte by byte to this space (no paging).

No access protection is provided on this read only space.

4.6.2 Erasing

Not applicable for this read only space.

4.6.3 Limits

This space is not bit addressing and an unimplemented register returns 0xFF.

4.6.4 Device Registers Description

c.f. appropriate data sheet for information.

4.6.4.1 CANBT[1..3] Registers

The CANBT[1..3] Registers are at the addresses 0xE2 to 0xE4.

They can be read before disabling the auto-bit rate (**EB !=** 0xFFH) and re- copied into "**BTC1**", "**BTC2**" & "**BTC3**" of the Boot loader configuration space (see "Bit-Timing Control [1..3] - "BTC[1..3]"" on page 14). . Then, the Boot loader will always start with this Bit-Timing (while **EB !=** 0xFFH !!!). Is very useful in case of IAP.

5. CAN Protocol & ISP Commands

This section describes the higher level protocol over the CAN network communication and the coding of the associated ISP commands.

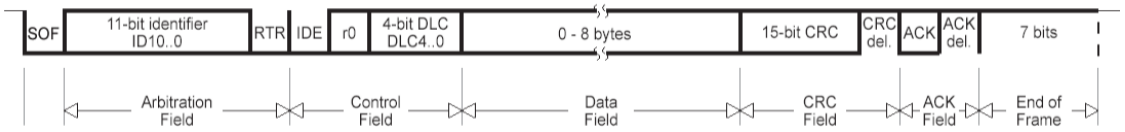
5.1 CAN Frame Description

The CAN protocol only supports the CAN standard frame (c.f. ISO 11898 for high speed and ISO 11519-2 for low speed) also known as CAN 2.0 A with 11-bit identifier.

A message in the CAN standard frame format begins with the "Start Of Frame (SOF)", this is followed by the "Arbitration field" which consist of the identifier and the "Remote Transmission Request (RTR)" bit used to distinguish between the data frame and the data request frame called remote frame. The following "Control field" contains the "IDentifier Extension (IDE)" bit and the "Data Length Code (DLC)" used to indicate the number of following data bytes in the "Data field". In a remote frame, the DLC contains the number of requested data bytes. The "Data field" that follows can hold up to 8 data bytes. The frame integrity is guaranteed by the following "Cyclic Redundant Check (CRC)" sum. The "ACKnowledge (ACK) field" compromises the ACK slot and the ACK delimiter. The bit in the ACK slot is sent as a recessive bit and is overwritten as a dominant bit by the receivers which have at this time received the data correctly.

The ISP CAN protocol only uses CAN standard data frame.

Figure 5-1. CAN Standard Data Frame



To describe the ISP CAN protocol, a symbolic name is used for Identifier, but default values are given within the following presentation.

Table 5-1. Template for ISP CAN command

Identifier 11 bits	**Length** 4 bits	**Data[0]** 1 byte	**...**	**Data[n-1]** 1 byte	**Description**
SYMBOLIC_NAME (“**CRIS**”<<4) + **X**	n (≤8)	Value or meaning			Command description

Because in a point-to-point connection, the transmit CAN message is repeated until a hardware acknowledge is done by the receiver.

The boot loader can acknowledge an incoming CAN frame only if a configuration is found.

This functionality is not guaranteed on a network with several CAN nodes.

5.2 CAN ISP Command Data Stream Protocol

5.2.1 CAN ISP Command Description

Several CAN message identifiers are defined to manage this protocol.

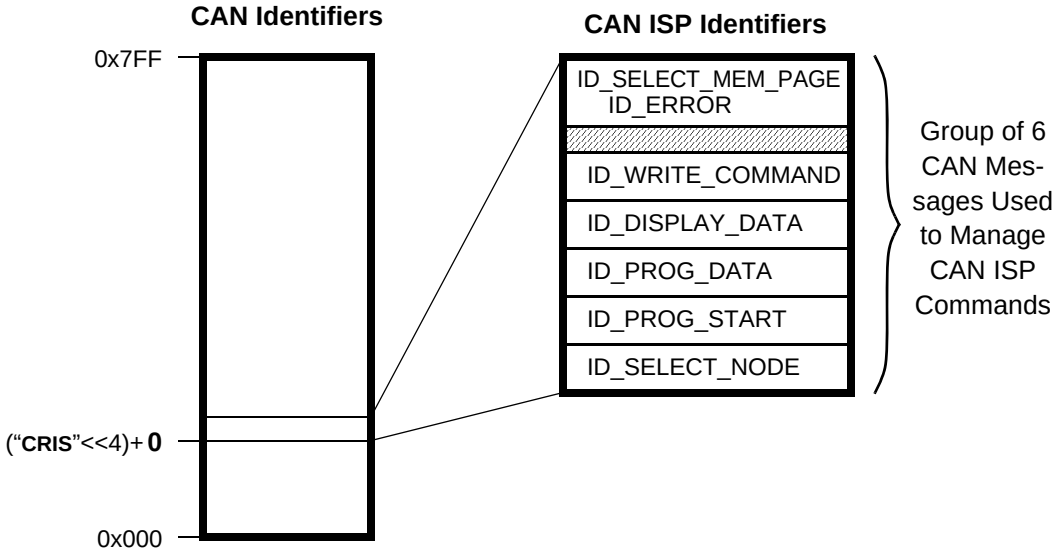
Table 5-2. Defined CAN Message Identifiers for CAN ISP Protocol

Identifier	ISP Command Detail	Value
ID_SELECT_NODE	Open/Close a communication with a node	(“**CRIS**” << 4) + **0**
ID_PROG_START	Start Memory space programming	(“**CRIS**” << 4) + **1**
ID_PROG_DATA	Data for Memory space programming	(“**CRIS**” << 4) + **2**
ID_DISPLAY_DATA	Read data from Memory space	(“**CRIS**” << 4) + **3**
ID_START_APPLI	Start application	(“**CRIS**” << 4) + **4**
ID_SELECT_MEM_PAGE	Selection of Memory space or page	(“**CRIS**” << 4) + **6**
ID_ERROR	Error message from boot loader only	

It is possible to allocate a new value for CAN ISP identifiers by writing the “**CRIS**” byte with the base value for the group of identifier.

The maximum “**CRIS**” value is 0x7F and its the default value is 0x00.

Figure 5-2. Remapping of CAN Message Identifiers for CAN ISP Protocol



Example: “**CRIS**” = 0x28

– “**ID_SELECT_NODE**” = 0x280
– ....
– “**ID_ERROR**” = 0x286

5.2.2 Communication Initialization

The communication with a device (CAN node) must be opened prior to initiate any ISP communication. To open communication with the device, the Host sends a “Connecting” CAN message (“*ID_SELECT_NODE*”) with the node number “**NNB**” passed as parameter. If the node number passed is 0xFF then the CAN boot loader accepts the communication (Figure 5-3). Otherwise the node number passed in parameter must be equal to the local “**NNB**” (Figure 5-4).

Figure 5-3. CAN Boot Loader First Connection

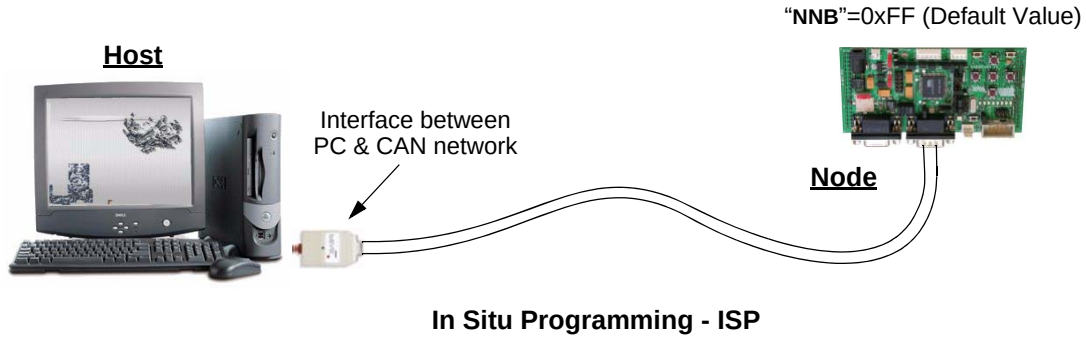
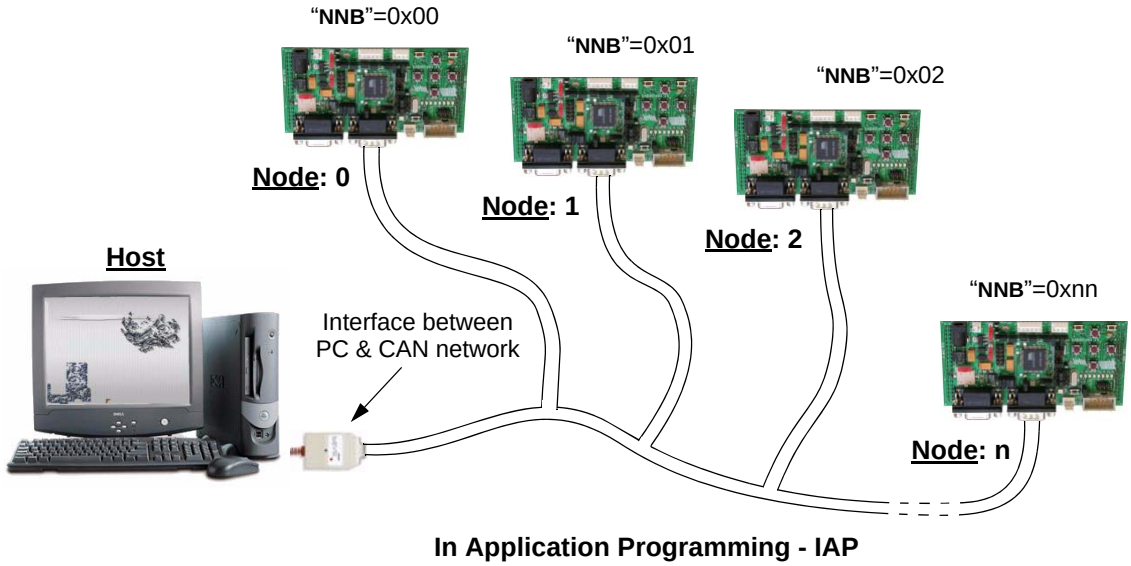


Figure 5-4. CAN Boot Loader Network Connection



Before opening a new communication with another device, the current device communication must be closed with its connecting CAN message (“*ID_SELECT_NODE*”).

5.3 CAN ISP Commands

5.3.1 CAN Node Select

A CAN node must be first **opened** at the beginning and then **closed** at the end of the session.

5.3.1.1 CAN Node Select Requests from Host

Table 5-3. CAN Node Select Requests from Host

Identifier	L	Data[0]	Description
ID_SELECT_NODE (("**CRIS**"<<4)+ **0**)	1	Node Number ("**NNB**")	**Open** or **close** communication with a specific node

5.3.1.2 CAN Node Select Answers from Boot Loader

Table 5-4. CAN Node Select Answers from Boot Loader

Identifier	L	Data[0]	Data[1]	Description
ID_SELECT_NODE (("**CRIS**"<<4)+ **0**)	2	"*Boot Loader Revision*"	0x00	Communication **closed**
			0x01	Communication **opened**

5.3.2 Changing Memory / Page

To change of memory space and/or of page, there is only one command, the switch is made by "*Data[0]*" of the CAN frame.

5.3.2.1 Changing Memory / Page Requests from Host

Table 5-5. Changing Memory / Page Requests from Host

Identifier	L	Data[0]	Data[1]	Data[2]	Description
ID_SELECT_MEM_PAGE (("**CRIS**"<<4)+ **6**)	3	0x00	Memory space	Page	No action
		0x01			Select Memory space
		0x02			Select Page
		0x03			Select Memory space & Page

5.3.2.2 Changing Memory / Page Answers from Boot Loader

Table 5-6. Changing Memory / Page Answers from Boot Loader

Identifier	L	Data[0]	Description
ID_SELECT_MEM_PAGE (("**CRIS**"<<4)+ **6**)	1	0x00	Selection OK (even if "*Data[0]*"=0 in the request frame)

5.3.3 Reading / Blank Checking Memory

These operations can be executed only with a device previously open in communication. This command is available on the memory space and on the page previously defined.

To start the reading or blank checking operation, the Host sends a CAN message ("*ID_DISPLAY_DATA*") with the operation required in Data[0], the start address and end address are passed as parameters.

5.3.3.1 *Reading / Blank Checking Memory Requests from Host*

Table 5-7. Reading / Blank Checking Memory Requests from Host

Identifier	L	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Description
ID_DISPLAY_DATA (("**CRIS**"<<4)+ **3**)	5	0x00	Start Address (MSB, LSB)		End Address (MSB, LSB)		Display data of selected Memory space / Page
		0x80					Blank check on selected Memory space / Page

5.3.3.2 *Reading / Blank Checking Memory Answers from Boot Loader*

Table 5-8. Reading / Blank Checking Memory Answers from Boot Loader

Identifier	L	Data[0]	Data[1]	...	Data[7]	Description
ID_DISPLAY_DATA (("**CRIS**"<<4)+ **3**)	up to 8	Up to 8 Data Bytes				Data Read
	0	-	-	-	-	Blank check OK
	2	First not blank address		-	-	Error on Blank check
ID_ERROR (("**CRIS**"<<4)+ **6**)	1	0x00	-	-	-	Error Software Security Set ("*Display data*" only)

5.3.4 Programming / Erasing Memory

These operations can be executed only with a device previously open in communication. They need two steps:

- The first step is to indicate address range for program or erase command.
- The second step is to transmit the data for programming only.

To start the programming operation, the Host sends a "start programming" CAN message (ID_PROG_START) with the operation required in "*Data[0]*", the start address and the end address are passed as parameters.

5.3.4.1 *Programming / Erasing Memory Requests from Host*

Table 5-9. Unit. Programming / Erasing Memory Requests from Host

Identifier	L	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5..7]	Description
ID_PROG_START (("**CRIS**"<<4)+ **1**)	5	0x00	Start Address (MSB, LSB)		End Address (MSB, LSB)		-	Init. prog. the selected Memory space / Page
	3	0x80	0xFF	0xFF	-	-	-	Erase the selected Memory space / Page
ID_PROG_DATA (("**CRIS**"<<4)+ **2**)	n	data[0..(n-1)] ($n \leq 8$)						Data to program

5.3.4.2 *Programming / Erasing Memory Answers from Boot Loader*

Table 5-10. Programming / Erasing Memory Answers from Boot Loader

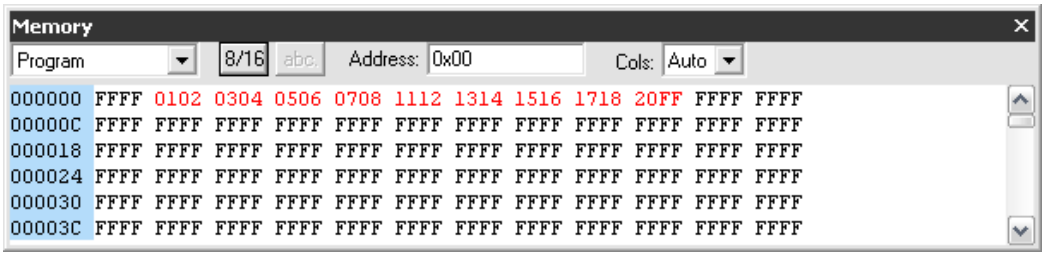
Identifier	L	Data[0]	Description
ID_PROG_START (("**CRIS**"<<4)+ **1**)	0	-	Command OK
ID_PROG_DATA (("**CRIS**"<<4)+ **2**)	1	0x00	Command OK and end of transfer
		0x02	Command OK but new (other) data expected
ID_ERROR (("**CRIS**"<<4)+ **6**)	1	0x00	Error - Software Security Set ("*Init. program*" only)

5.3.4.3 *Programming Memory Examples*

Table 5-11. Programming Memory Examples

Request/ Answer	CAN Message (hexadecimal)			Description
	Identifier	L	Data[..70]	
R (>>)	000	1	FF	CAN Node Select
A (<<)	000	2	03 01	Communication opened
Default Memory space = Flash, default Page = page_0				
R (>>)	001	5	00 00 02 00 12	Prog. Add 0x0002 up to 0x0012
A (<<)	001	0	0	Command OK
R (>>)	002	8	01 02 03 04 05 06 07 08	1st Data transfer
A (<<)	002	1	02	Command OK, new data expected
R (>>)	002	8	11 12 13 14 15 16 17 18	2nd Data transfer
A (<<)	002	1	02	Command OK, new data expected
R (>>)	002	1	20	3rd Data transfer
A (<<)	002	1	00	Command OK, end of transfer

Figure 5-5. Result of the Above Programming Memory Example (1)



Note: 1. AVR Studio® Program Memory display

5.3.5 Starting Application

This operation can be executed only with a device previously open in communication.

5.3.5.1 Starting Application Requests from Host

To start the application, the host sends a start application CAN message with the “way of” selected in “*Data[1]*”. The application can be start by a watchdog reset or by jumping to the defined word address. The jump word address can be differ from **SA_H:SA_L** (Boot Loader Configuration Space).

Table 5-12. Start application Requests from Host

Identifier	L	Data[0]	Data[1]	Data[2]	Data[3]	Description
ID_START_APPLI (("**CRIS**"<<4)+ **4**)	2	0x03	0x00	-	-	Start application with watchdog reset
	4		0x01	Jump W-Add. (MSB, LSB)		Start Application at W-Add. (MSB : LSB) without reset

5.3.5.2 Starting Application Answer from Boot Loader

No answer is returned by the boot loader.

6. API - Application Programming Interface

6.1 API Definition

An application programming interface (API) is a source code interface that a computer system or program library provides in order to support requests for services to be made of it by a computer program.

6.2 API Implementation

The specificity of Atmel® AVR 8-bit microprocessors is that the code providing the writing in flash needs to be located in the boot loader section. If the "Slim" CAN Boot Loader is flashed in a part, the boot loader section is already occupied and unavailable for user.

The solution that allows the user to write in flash is to “open” the “`flash_wr_block()`” routine contained in the "Slim" CAN Boot Loader. Then the user could call it from its own application (and use a well-tried piece of code).

6.3 API - Limitation of Use

The CAN Boot Loader flashed was developed and compiled with IAR™. Only a user program also compiled with IAR can access to (call) the "Slim" CAN Boot Loader API.

6.4 API Details

6.4.1 Function Name

`flash_wr_block()`

Note: This function is located in “*flash_boot_lib.c*” file.

6.4.2 Features

This function allows to write up to 65535 bytes (64K Bytes-1 byte) in the Flash memory. This function manages alignment issue (byte and flash-page alignments).

Note:
1. This function is not able to address the fully 65535 bytes in one time because we cannot find in the device a source buffer up to 64K bytes !
2. For Flash memory size greater than 64K, the page setting must be done before (ex: setting or clearing the RAMPZ register). The default setting is 0.

6.4.3 Warning

Bytes to program must be in a different page that “`flash_wr_block()`” function.

6.4.4 Function Parameters

1. `* src`: Pointer on **`unsigned char`** - Source buffer (in SRAM),
2. `dest`: **`unsigned short`** - Destination, start address value in Flash memory where data must be written,
3. `byte_nb`: **`unsigned short`** - Number of bytes to write.

6.4.5 Function Return

(none)

6.5 Entry Point

Because the user application and the "Slim" CAN Boot Loader are not compiled together, an entry point is used. Regardless the device part number, the entry point is fixed to:

API_ENTRY_POINT = (FLASH_SIZE - FLASH_BOOT_SIZE[1]) + 4 bytes

Note: 1. FLASH_BOOT_SIZE = 4 Kilo Bytes

The following table gives the API entry point address versus used device.

Table 6-1. API Entry Point Versus Device Part Number.

Part Number	API_ENTRY_POINT - Word Address	API_ENTRY_POINT - Byte Address
AT90CAN32	0x03802	0x07004
AT90CAN64	0x07802	0x0F004
AT90CAN128	0x0F802	0x1F004
ATmega16M1	0x01802	0x03004
ATmega32M1	0x03802	0x07004
ATmega32C1	0x03802	0x07004
ATmega64M1	0x07802	0x0F004
ATmega64C1	0x07802	0x0F004

6.6 API Call Example with IAR

The user program can call the API used the following code [1]:

- In the driver (*.c) file - Function

```
void (*flash_write) (unsigned char* src,     \
                     unsigned short dest,    \
                     unsigned short byte_nb) \
          = (void (*)(unsigned char*,        \
                      unsigned short,        \
                      unsigned short))       \
                     (API_ENTRY_POINT)       ;
```

Note: Here, ENTRY_POINT is a byte address

- In the header (*.h) file - Function prototype

```
extern void (*flash_write) (unsigned char* src,    \
                            unsigned short dest,   \
                            unsigned short byte_nb);
```

Note: 1. Only available for IAR C Compiler.

6.7 API Call Example with Other C Compilers

6.7.1 Passing Variables Between IAR C-compiler and Other Compiler or Assembler

When the IAR C-compiler is used for the AVR the Register File is segmented as shown in Figure 6-1.

- Scratch Registers are not preserved across functions calls.
- Local registers are preserved across function calls.
- The Y Register (R28:R29) is used as Data Stack Pointer to SRAM.
- The Scratch Registers are used to passing parameters and return values between functions.

When a function is called the parameters to be passed to the function is placed in the Register File Registers R16-R23. When a function is returning a value this value is placed in the Register File Registers R16-R19, depending on the size of the parameters and the returned value.

Figure 6-2 shows example placement of parameter when calling a function:

Figure 6-1. Segments in the Register File.

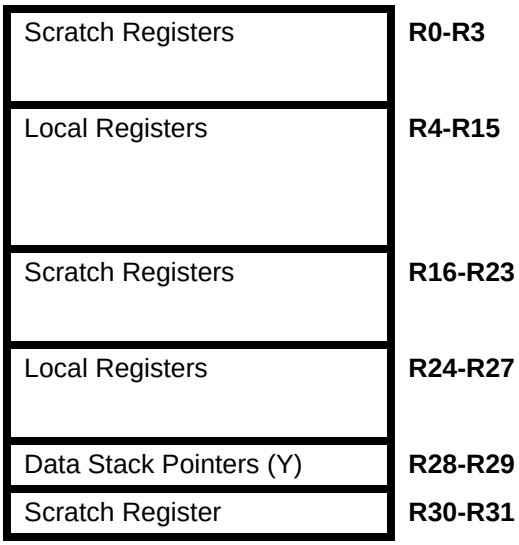


Table 6-2. Placement and Parameters to C-functions.

Function	Parameter 1 Registers	Parameter 2 Registers
func (char, char)	R16	R20
func (char, short)	R16	R20, R21
func (short, long)	R16, R17	R20, R211, R22, R23
func (long, long)	R16, R17, R18, R19	R20, R21, R22, R23

For complete reference of the supported data types and corresponding sizes, see the *AVR® IAR Compiler Guide* from IAR Systems, Data Representation section.

6.7.2 Registers used in "`flash_wr_block()`" API

- Function parameters:

- ***src** : R17:R16 then => R9:R8 => and transferred in Z(R31:30)
- **dest** : R19:R18 then => R25:R24 => R27:R26(address)
- **nb_bytes** : R21:R20 then => R5:R4

- Other resources:
  - Y(R29:R28) is used as “Data Stack Pointer”
  - Z(R31:30) is used with “LPM” & “SPM”
  - Clear R0 using “LPM” and (R0:R1) is used in “fill_temp_buffer”
- Summary of registers used:

  R0, R1, R4, R5, R6, R7, R8, R9, R10, R11, R16, R17, R18, R19, R20, R21, R24, R25, R26, R27, R28, R29, R30, R31

- “*flash_boot_lib.lst*” file extract:

```
138          //---------------------------------------------------------------------------
    139          #pragma location ="API_FLASH"

   \                                 In segment API_FLASH, align 2, keep-with-next
    140          void flash_wr_block(U8* src, U16 dest, U16 byte_nb)
   \                        flash_wr_block:
    141          {
   \   00000000   92BA               ST      -Y, R11
   \   00000002   92AA               ST      -Y, R10
   \   00000004   929A               ST      -Y, R9
   \   00000006   928A               ST      -Y, R8
   \   00000008   927A               ST      -Y, R7
   \   0000000A   926A               ST      -Y, R6
   \   0000000C   925A               ST      -Y, R5
   \   0000000E   924A               ST      -Y, R4
   \   00000010   93BA               ST      -Y, R27
   \   00000012   93AA               ST      -Y, R26
   \   00000014   939A               ST      -Y, R25
   \   00000016   938A               ST      -Y, R24
   \   00000018                      REQUIRE ?Register_R4_is_cg_reg
   \   00000018                      REQUIRE ?Register_R5_is_cg_reg
   \   00000018                      REQUIRE ?Register_R6_is_cg_reg
   \   00000018                      REQUIRE ?Register_R7_is_cg_reg
   \   00000018                      REQUIRE ?Register_R8_is_cg_reg
   \   00000018                      REQUIRE ?Register_R9_is_cg_reg
   \   00000018                      REQUIRE ?Register_R10_is_cg_reg
   \   00000018                      REQUIRE ?Register_R11_is_cg_reg
   \   00000018   0148               MOVW    R9:R8, R17:R16
   \   0000001A   01C9               MOVW    R25:R24, R19:R18
   \   0000001C   012A               MOVW    R5:R4, R21:R20
    142          U8      save_i_flag;
    143          U16     u16_temp, nb_word;
    144          U16     address;
    145          U16     save_page_addr;
    146
    147              //--- Special for API's ------------------------
    148              //- First of all, disabling the Global Interrupt
    149              save_i_flag = SREG;
   \   0000001E   B6AF               IN      R10, 0x3F
    150              Disable_interrupt();
   \   00000020   94F8               CLI
```

Note: For more details, please refer to the following *.lst files (if generated ! - (compiler option)):
- “*flash_boot_lib.lst*” file,
- “*flash_boot_drv.lst*” file.

7. Appendix A: #define's

7.1 Processor Definition

```
//---------------- BOOT LOADER DEFINITION -------------------------------------
#define     BOOT_LOADER_SIZE            0x1000  // Size in bytes: 4KB
#define     MAX_FLASH_SIZE_TO_ERASE     ( FLASH_SIZE - ((U32)(BOOT_LOADER_SIZE)) )

//---------------- PROCESSOR DEFINITION ---------------------------------------

#define     XRAM_END            XRAMEND     // Defined in "ioxxx.h"
#define     RAM_END             RAMEND      // Defined in "ioxxx.h"
#define     E2_END              E2END       // Defined in "ioxxx.h"
#define     FLASH_END           FLASHEND    // Defined in bytes in "ioxxx.h"
#define     FLASH_SIZE          ((U32)(FLASH_END)) + 1 // Size in bytes

// Switches for specific definitions
#if   defined(__AT90CAN128__)               // __HAS_ELPM__ defined by IAR
#   define  MANUF_ID            0x1E        // ATMEL
#   define  FAMILY_CODE         0x81        // AT90CANxxx family
#   define  PRODUCT_NAME        0x97        // 128 Kbytes of Flash
#   define  PRODUCT_REV         0x00        // Rev 0
#   define  FLASH_PAGE_SIZE     256         // Size in bytes
#   define  _BOOT_CONF_TYPE_    __farflash  // Bootloader mapped above 64K
#   define  _RAMPZ_IS_USED_                 // RAMPZ register used if Flash memory
used is upper than 64K bytes


#elif defined(__AT90CAN64__)                // __HAS_ELPM__ not-defined by IAR
#   define  MANUF_ID            0x1E        // ATMEL
#   define  FAMILY_CODE         0x81        // AT90CANxxx family
#   define  PRODUCT_NAME        0x96        // 64 Kbytes of Flash
#   define  PRODUCT_REV         0x00        // Rev 0
#   define  FLASH_PAGE_SIZE     256         // Size in bytes
#   define  _BOOT_CONF_TYPE_    __flash     // Bootloader mapped below 64K


#elif defined(__AT90CAN32__)                // __HAS_ELPM__ not-defined by IAR
#   define  MANUF_ID            0x1E        // ATMEL
#   define  FAMILY_CODE         0x81        // AT90CANxxx family
#   define  PRODUCT_NAME        0x95        // 32 Kbytes of Flash
#   define  PRODUCT_REV         0x00        // Rev 0
#   define  FLASH_PAGE_SIZE     256         // Size in bytes
#   define  _BOOT_CONF_TYPE_    __flash     // Bootloader mapped below 64K

#elif defined(__ATmega16M1__)               // __HAS_ELPM__ not-defined by IAR
#   define  MANUF_ID            0x1E        // ATMEL
#   define  FAMILY_CODE         0x84        // ATmegaxxM1C1 family
#   define  PRODUCT_NAME        0x94        // 16 Kbytes of Flash
#   define  PRODUCT_REV         0x00        // Rev 0
#   define  FLASH_PAGE_SIZE     128         // Size in bytes
#   define  _BOOT_CONF_TYPE_    __flash     // Bootloader mapped below 64K

#elif defined(__ATmega32M1__)               // __HAS_ELPM__ not-defined by IAR
#   define  MANUF_ID            0x1E        // ATMEL
#   define  FAMILY_CODE         0x84        // ATmegaxxM1C1 family
#   define  PRODUCT_NAME        0x95        // 32 Kbytes of Flash
#   define  PRODUCT_REV         0x00        // Rev 0
#   define  FLASH_PAGE_SIZE     128         // Size in bytes
#   define  _BOOT_CONF_TYPE_    __flash     // Bootloader mapped below 64K

#elif defined(__ATmega32C1__)                // __HAS_ELPM__ not-defined by IAR
#   define  MANUF_ID            0x1E        // ATMEL
#   define  FAMILY_CODE         0x86        // ATmegaxxM1C1 family
#   define  PRODUCT_NAME        0x95        // 32 Kbytes of Flash
#   define  PRODUCT_REV         0x00        // Rev 0
#   define  FLASH_PAGE_SIZE     128         // Size in bytes
#   define  _BOOT_CONF_TYPE_    __flash     // Bootloader mapped below 64K

#elif defined(__ATmega64M1__)               // __HAS_ELPM__ not-defined by IAR
#   define  MANUF_ID            0x1E        // ATMEL
```

```
#   define  FAMILY_CODE         0x84        // AATmegaxxM1C1 family
#   define  PRODUCT_NAME        0x96        // 64 Kbytes of Flash
#   define  PRODUCT_REV         0x00        // Rev 0
#   define  FLASH_PAGE_SIZE     256         // Size in bytes
#   define  _BOOT_CONF_TYPE_    __flash     // Bootloader mapped below 64K

#elif defined(__ATmega64C1__)               // __HAS_ELPM__ not-defined by IAR
#   define  MANUF_ID            0x1E        // ATMEL
#   define  FAMILY_CODE         0x86        // ATmegaxxM1C1 family
#   define  PRODUCT_NAME        0x96        // 64 Kbytes of Flash
#   define  PRODUCT_REV         0x00        // Rev 0
#   define  FLASH_PAGE_SIZE     256         // Size in bytes
#   define  _BOOT_CONF_TYPE_    __flash     // Bootloader mapped below 64K

#else
#   error Wrong device selection in IAR Embedded Workbench IDE
#endif

#define     FOSC                8000        // 8 MHz External crystal
```

7.2 CAN Link Definition

```
//---------------- CAN DEFINITION -------------
#define CAN_BAUDRATE    CAN_AUTOBAUD

#if    defined(__AT90CAN128__) || \
       defined(__AT90CAN64__)  || \
       defined(__AT90CAN32__)
#   define CAN_PORT_IN     PIND
#   define CAN_PORT_DIR    DDRD
#   define CAN_PORT_OUT    PORTD
#   define CAN_INPUT_PIN   6
#   define CAN_OUTPUT_PIN  5
#   define NB_MOB          15
#elif  defined(__ATmega16M1__) || \
       defined(__ATmega32M1__) || \
       defined(__ATmega32C1__) || \
       defined(__ATmega64M1__) || \
       defined(__ATmega64C1__)
#   define CAN_PORT_IN     PINC
#   define CAN_PORT_DIR    DDRC
#   define CAN_PORT_OUT    PORTC
#   define CAN_INPUT_PIN   3
#   define CAN_OUTPUT_PIN  2
#   define NB_MOB          6
#else
#   error Wrong device selection in IAR Embedded Workbench IDE
#endif
```

7.3 Boot Loader Definition

```
//-------------- BOOTLOADER CONFIGURATION -------------
#define BOOT_LOADER_SIZE            0x1000  // Size in bytes: 4KB
#define MAX_FLASH_SIZE_TO_ERASE   ( FLASH_SIZE - ((U32)(BOOT_LOADER_SIZE)) )

#define BOOT_VERSION    0x03
#define BOOT_ID1        0xD1
#define BOOT_ID2        0xD2

#define BSB_DEFAULT     0xFF
#define SSB_DEFAULT     0xFF
#define EB_DEFAULT      0xFF
#define BTC1_DEFAULT    0xFF
#define BTC2_DEFAULT    0xFF
#define BTC3_DEFAULT    0xFF
#define NNB_DEFAULT     0xFF
#define CRIS_DEFAULT    0x00
#define SA_H_DEFAULT    0x00
#define SA_L_DEFAULT    0x00

#define BSB  ((U16) &boot_conf[0])
```

```
#define SSB  ((U16) &boot_conf[1])
#define EB   ((U16) &boot_conf[2])
#define BTC1 ((U16) &boot_conf[3])
#define BTC2 ((U16) &boot_conf[4])
#define BTC3 ((U16) &boot_conf[5])
#define NNB  ((U16) &boot_conf[6])
#define CRIS ((U16) &boot_conf[7])
#define SA_H ((U16) &boot_conf[8])
#define SA_L ((U16) &boot_conf[9])

#define BOOT_CONF_SIZE         10
#define SSB_INDEX            0x01
#define SSB_NO_SECURITY      0xFF
#define SSB_WR_PROTECTION    0xFE
#define SSB_RD_WR_PROTECTION 0xFC
```

7.4 Memory Definition

```
//-------- MEMORY DEFINITION -----------------
#define MEM_FLASH               0
#define MEM_EEPROM              1
#define MEM_SIGNATURE           2
#define MEM_BOOT_INF            3   // Boot Loader information
#define MEM_BOOT_CONF           4   // Boot Loader configuration
#define MEM_HW_REG              5
#define MEM_DEF_MAX             MEM_HW_REG
#define MEM_DEFAULT             MEM_FLASH

#define PAGE_DEFAULT       0x00
#define ADD_DEFAULT        0x0000
#define N_DEFAULT          0x0001
```

7.5 CAN Protocol Definition

```
//------ IAP data --------------------------
#define MAX_BASE_ISP_IAP_ID         0x7F0
#define MIN_BASE_ISP_IAP_ID         0x000

//------ Protocol commands -----------------
#define CAN_ID_SELECT_NODE          0x00

#define CAN_ID_PROG_START           0x01
#   define  CAN_INIT_PROG          0x00
#   define  CAN_FULL_ERASE_1       0x80
#   define  CAN_FULL_ERASE_2       0xFF
#   define  CAN_FULL_ERASE_3       0xFF

#define CAN_ID_PROG_DATA            0x02

#define CAN_ID_DISPLAY_DATA         0x03
#   define  CAN_READ_DATA          0x00
#   define  CAN_BLANK_CHECK        0x80

#define CAN_ID_START_APPLI          0x04
#   define  CAN_START_APPLI        0x03
#   define  CAN_RESET_APPLI        0x00
#   define  CAN_JUMP_APPLI         0x01

#define CAN_ID_SELECT_MEM_PAGE      0x06
#   define  CAN_NO_ACTION          0x00
#   define  CAN_SEL_MEM            0x01
#   define  CAN_SEL_PAGE           0x02
#   define  CAN_SEL_MEM_N_PAGE     0x03

#define CAN_ID_ERROR                0x06

#define COMMAND_OK                  0x00
#define OK_END_OF_DATA              0x00
#define OK_NEW_DATA                 0x02

#define LOCAL_BUFFER_SIZE           0x100
```

8. Appendix B: CAN Protocol Summary

Table 8-1. CAN Protocol Summary - Requests from Host

ISP Command Request Identifier	L	Data [0]	Data [1]	Data [2]	Data [3]	Data [4]	Data [5]	Data [6]	Data [7]	Description
ID_SELECT_NODE (("**CRIS**"<<4)+ **0**)	1	Node	-	-	-	-	-	-	-	Open or close communication
ID_PROG_START (("**CRIS**"<<4)+ **1**)	5	0x00	Start Address		End Address		-	-	-	Initialization of programming
	3	0x80	0xFF	0xFF	-	-	-	-	-	Erasing
ID_PROG_DATA (("**CRIS**"<<4)+ **2**)	n	data[0..(n-1)] (n≤8)								Data to program
ID_DISPLAY_DATA (("**CRIS**"<<4)+ **3**)	5	0x00	Start Address		End Address		-	-	-	Display (read) data
		0x80					-	-	-	Blank check
ID_START_APPLI (("**CRIS**"<<4)+ **4**)	2	0x03	0x00	-	-	-	-	-	-	Start Application with reset
	4		0x01	0x0000		-	-	-	-	Start Application jump add. 0
ID_SELECT_MEM_PAGE (("**CRIS**"<<4)+ **6**)	3	0x00	Memory space	Page	-	-	-	-	-	No action
		0x01			-	-	-	-	-	Select Memory space
		0x02			-	-	-	-	-	Select Page
		0x03			-	-	-	-	-	Select Memory space & Page

Table 8-2. CAN Protocol Summary - Answers from Boot Loader

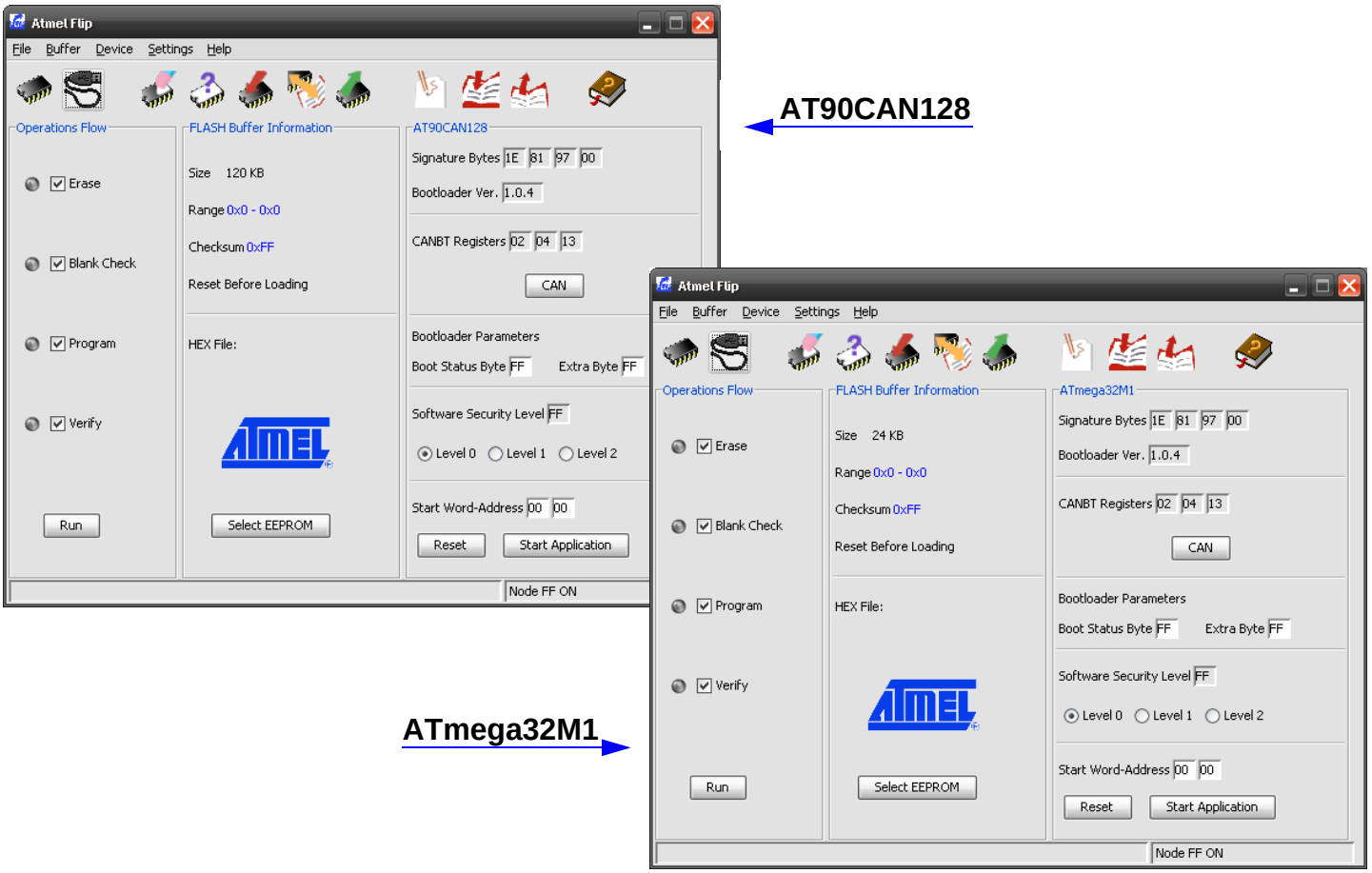
ISP Command Answer Identifier	L	Data [0]	Data [1]	Data [2]	Data [3]	Data [4]	Data [5]	Data [6]	Data [7]	Description
ID_SELECT_NODE (("**CRIS**"<<4)+ **0**)	2	Boot loader revision	0x00	-	-	-	-	-	-	Communication closed
			0x01	-	-	-	-	-	-	Communication opened
ID_PROG_START (("**CRIS**"<<4)+ **1**)	0	-	-	-	-	-	-	-	-	Command OK
ID_PROG_DATA (("**CRIS**"<<4)+ **2**)	1	0x00	-	-	-	-	-	-	-	Cmd. OK & end of transfer
		0x02	-	-	-	-	-	-	-	Cmd. OK & new data expected
ID_DISPLAY_DATA (("**CRIS**"<<4)+ **3**)	n	data[0..(n-1)] (n≤8)								Data Read
	0	-	-	-	-	-	-	-	-	Blank check OK
	2	1st Failed Address		-	-	-	-	-	-	Error on Blank check
ID_SELECT_MEM_PAGE or ID_ERROR (("**CRIS**"<<4)+ **6**)	1	0x00	-	-	-	-	-	-	-	Selection OK or Error Software Security Set

9. Appendix C: Flip Interface

FLIP is a flexible PC-application which lets you program and configure Atmel's microcontroller devices in-system. This new major version of FLIP offers the following capabilities :

- Perform In-System Programming through RS232, USB or CAN interfaces.
- May be used through its intuitive Graphical User Interface or launched from a DOS® window (see the batchisp manual), from an embedded software IDE like KEIL's uVision2, or even from your own application (see the ISP Functions Library manual).
- Runs under Windows® 9x / Me / NT® / 2000 / XP®
- Supports Intel® MCS-86 Hexadecimal Object, Code 88 file format for data file loading and saving.
- Buffer editing capabilities : fill, search, copy, reset, modify, goto address.
- Target device memory control : erase, blank check, program, verify, read, security level and special bytes reading and setting.
- Parts serialization capability (from batchisp only).
- ISP hardware conditions may be set by software.
- The demo mode emulates ISP operations without any target hardware.

Figure 9-1. Flip Window Examples



Flip link: *http://www.atmel.com/dyn/resources/prod_documents/Flip_Installer_3_3_4.exe*

10. Appendix C: Compiling Notes vs Targeted Devices

References : - *IAR C/C++ Compiler for AVR 5.20.1 (5.20.1.50092)*
- *Platform selection in "..\$PROJ_DIR$\config.h" file*

10.1 Targeted Selection

Project -> Options -> General Options -> Target -> Processor Configuration

- *either*: --cpu=can128, **AT90CAN128**
- *either*: --cpu=can64, **AT90CAN64**
- *either*: --cpu=can32, **AT90CAN32**
- *either*: --cpu=32m1, **ATmega32M1**
- *or*: --cpu=32c1, **ATmega32C1**
  (*not yet implemented - 13 April 2009*)
- ... : --cpu=64m1, **ATmega64M1**
- ... : --cpu=64c1, **ATmega64C1**
- ... : --cpu=16m1, **ATmega16M1**

10.2 Optimizations

Project -> Options -> C/C++ Compiler -> Optimizations

- Speed: **High**(Maximum optimization)
- Number of cross-call passes: **Ulimited**
- Always do cross call optimization: **ON**

10.3 Selection of the Corresponding Linker Command File

Project -> Options -> Linker -> Config -> Linker Command File

- ☑ Override default
- either: *..\$PROJ_DIR$\can128_iar_can_bootloader_link.xcl*
- either: *..\$PROJ_DIR$\can64_iar_can_bootloader_link.xcl*
- either: *..\$PROJ_DIR$\can32_iar_can_bootloader_link.xcl*
- either: *..\$PROJ_DIR$\m64m1c1_iar_can_bootloader_link.xcl*
- either: *..\$PROJ_DIR$\m32m1c1_iar_can_bootloader_link.xcl*
- or: *..\$PROJ_DIR$\m16m1c1_iar_can_bootloader_link.xcl*

10.4 Compiling

1. *Project -> Clean*
2. *Project -> Rebuilt All*

Production of: "*IAR_can_boot_loader.a90*" (equ. "**.hex*") and "*IAR_can_boot_loader.dbg*"
located in: *..\$PROJ_DIR$\output_iar\debug\exe*

10.5 Pre Compiled Hexa Files

Some pre compiled hexa files (depending of the platform used) has been produced in:
..\$PROJ_DIR$\output_iar\debug\exe\pre_compiled_hex_file

- for **AT90CAN128** : *IAR_can_boot_loader_dvk90can1_at90can128.hex*
  or *IAR_can_boot_loader_stk600_at90can128.hex*
- for **AT90CAN64** : *IAR_can_boot_loader_stk600_at90can64.hex*
- for **AT90CAN32** : *IAR_can_boot_loader_stk600_at90can32.hex*
- for **ATmega32M1/C1** : *IAR_can_boot_loader_stk600_atmega32m1c1.hex*
  or *IAR_can_boot_loader_mc310_atmega32m1c1.hex*

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 Requests
www.atmel.com/literature