APPLICATION NOTE

AT6493: SAM C21/CAN BUS/Firmware

SMART ARM-Based Microcontroller

Introduction

CAN bus is a message based protocol designed specifically for automotive applications, but is also used in areas such as aerospace, maritime, railway vehicles, industrial automation, and medical equipment.

This application note will cover the firmware required to initialize and start the SAM C21 CAN controller and send/receive message in both standard and extended formats. Firmware within this document was created using Atmel® Software Framework (ASF), which is an extension to Atmel Studio.

Features

- +5V operation
- Dual independent CAN controllers
- Supports CAN 2.0, which is the latest specification consisting of two parts
  - Part A (CAN 2.0A)
    - Standard format with an 11-bit identifier
    - Data rates up to 256kbit/sec
  - Part B (CAN 2.0B)
    - Extended format with a 29-bit identifier
    - Data rates up to 1Mbit/sec

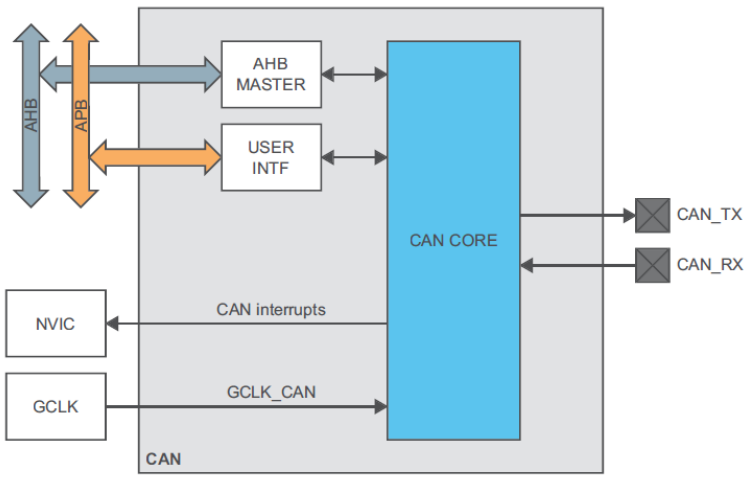


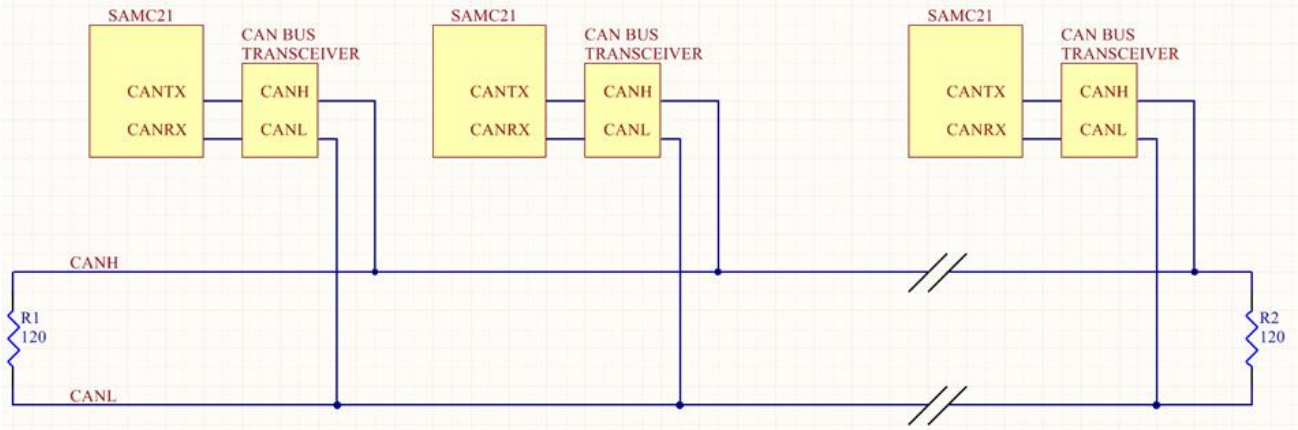
Table of Contents

1 CAN Hardware

1.1 Connecting the SAM C21 to the CAN bus

CAN is a multi-master serial bus used for connected nodes called Electronic Control Units (ECUs) – these are usually referred to as Nodes; all nodes are connected to each other over a two-wire bus. In order to connect the SAM C21 to the CAN bus, an external CAN transceiver is required. The bus should be terminated at both ends with 120Ω resistors. Figure 1-1 shows a typical CAN bus layout.

Figure 1-1. Typical CAN bus Layout



CAN bus transceivers are readily available from a number of semiconductor manufacturers, however, we would prefer that you use the Atmel ATA6560 or ATA6561 CAN bus transceivers.

2 CAN Setup

2.1 CAN bus Bitrate

The nominal and databit rates, among other basic CAN settings, can be found in the header file “conf_can.h”. Assuming the use of a 48MHz generic clock (GLCK) and a desired CAN bus speed of 500kHz the following defines are used to set the nominal bitrate:

- Nominal Bitrate Prescaler
  - #define CONF_CAN_NBTP_NBRP_VALUE 5
- Nominal bit (Re)Synchronization Jump Width
  - #define CONF_CAN_NBTP_NSJW_VALUE 3
- Nominal bit Time segment before sample point
  - #define CONF_CAN_NBTP_NTSEG1_VALUE 10
- Nominal bit Time segment after sample point
  - #define CONF_CAN_NBTP_NTSEG2_VALUE 3

Given the above #defines the time quanta is 48MHz / (5 + 1) = 8MHz, and each bit is (3 + 10 + 3) or 16 time quanta which is 8MHz / 16 = 500kHz.

In a similar manner, the following defines are used to set the databit rate:

- Databit Baud Rate Prescaler
  - #define CONF_CAN_DBTP_DBRP_VALUE 5
- Databit (Re)Synchronization Jump Width
  - #define CONF_CAN_DBTP_DSJW_VALUE 3
- Databit Time segment before sample point
  - #define CONF_CAN_DBTP_DTSEG1_VALUE 10
- Databit Time segment after sample point
  - #define CONF_CAN_DBTP_DTSEG2_VALUE 3

Given the above #defines the time quanta is 48MHz / (5 + 1) = 8MHz, and each bit is (3 + 10 + 3) or 16 time quanta which is 8MHz / 16 = 500kHz.

2.2 Message IDs

2.2.1 Transmit Message ID

The CAN message ID not only provides identification for the type of message being sent or received, it also determines the priority of the message. For example a message with an ID of 10 is of higher priority than a message with an ID of 15. In addition message IDs must be unique on a single CAN bus – no two nodes should send a message with the same ID.

The transmit message ID should be based on the type of data being sent and its priority.

2.2.2 Receive Message ID

The CAN offers the possibility to configure two sets of acceptance filters; one for standard identifiers and one for extended identifiers. These filters can be assigned to an RX Buffer or to RX FIFO 0 or 1. For acceptance filtering each list of filters is executed from element #0 until the first matching element. Acceptance filtering stops at the first matching element. The following filter elements are not evaluated for this message.

The main features of the acceptance filters are:

- Each filter element can be configured as
  - Range filter (from - to)
  - Filter for one or two dedicated IDs
  - Classic bit mask filter
- Each filter element is configurable for acceptance or rejection filtering
- Each filter element can be enabled / disabled individually
- Filters are checked sequentially, execution stops with the first matching filter element

The type of filtering is based on the settings of register bits SF1ID/SF2ID for standard frames and EF1ID/EF2ID for extended frames.

- Range Filtering
  - The filter matches for all received frames with Message IDs in the range defined by SF1ID/SF2ID for standard frames or EF1ID/EF2ID for extended frames
  - There are two possibilities when range filtering is used with extended frames:
    - EFT = “00”:
      - The Message ID of received frames is AND’ed with the Extended ID AND Mask (XIDAM) before the range filter is applied
    - EFT = “11”:
      - The Extended ID AND Mask (XIDAM) is not used for range filtering
- Filter for Specific IDs
  - A filter element can be configured to filter for one or two specific Message IDs. To filter for one specific Message ID, the filter element has to be configured with SF1ID = SF2ID for standard message format and EF1ID = EF2ID for extended format.
- Classic Bit Mask Filter
  - Classic bit mask filtering is intended to filter groups of Message IDs by masking single bits of a received Message ID. With classic bit mask filtering SF1ID/EF1ID is used as Message ID filter, while SF2ID/EF2ID is used as filter mask.
  - A zero bit at the filter mask will mask out the corresponding bit position of the configured ID filter, e.g. the value of the received Message ID at that bit position is not relevant for acceptance filtering. Only those bits of the received Message ID where the corresponding mask bits are one are relevant for acceptance filtering.
  - In case all mask bits are one, a match occurs only when the received Message ID and the Message ID filter are identical. If all mask bits are zero, all Message IDs match.

Settings of acceptance filtering will be done during initialization of the SAM C21’s CAN module as shown later in this application note.

3 CAN Configuration

3.1 CAN TX/RX Pins

In order to connect the CAN controllers TX and RX pins to the I/O pins of the processor, the pin MUX and correct I/O pins need to be set.

There are two discrete CAN controllers on the SAM C21, referred to as CAN0 and CAN1, each can be connected to one of two I/O pairs (refer to the I/O Multiplexing section of the SAM C21 datasheet for more details). The pin MUX and I/O pin settings are handled by the following code:

```
#define CAN_TX_MUX_SETTING      MUX_PA24G_CAN0_TX
#define CAN_TX_PIN              PIN_PA24G_CAN0_TX
#define CAN_RX_MUX_SETTING      MUX_PA25G_CAN0_RX
#define CAN_RX_PIN              PIN_PA25G_CAN0_RX

struct system_pinmux_config pin_config;
system_pinmux_get_config_defaults(&pin_config);
pin_config.mux_position = CAN_TX_MUX_SETTING;
system_pinmux_pin_set_config(CAN_TX_PIN, &pin_config);
pin_config.mux_position = CAN_RX_MUX_SETTING;
system_pinmux_pin_set_config(CAN_RX_PIN, &pin_config);
```

The CAN_TX_PIN and CAN_RX_PIN pin constants can be modified to select either one of the two I/O pin pairs supported by the pin MUX.

3.2 CAN Module Initialization

Once the CAN TX and RX pins have been defined the CAN module can be initialized. Sample initialization code for the CAN module running in normal mode is shown below:

```
#define CAN_MODULE              CAN0

struct can_config config_can;
can_get_config_defaults(&config_can);
can_init(&can_instance, CAN_MODULE, &config_can);

can_switch_operation_mode(&can_instance, CAN_OPERATION_MODE_NORMAL_OPERATION);
system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_CAN0);
```

The code initializes the CAN0 module, similar code can be placed for the CAN1 module for SAM C21 parts supporting its I/O pins.

3.3 Standard RX Filtering (11-bit Message ID)

The following function will enable standard message filtering:

```
#define CAN_RX_STANDARD_FILTER_INDEX       0

static void can_set_standard_filter(uint32_t filter_value)
{
        struct can_standard_message_filter_element sd_filter;

        can_get_standard_message_filter_element_default(&sd_filter);
        sd_filter.S0.bit.SFID1 = filter_value;

        can_set_rx_standand_filter(&can_instance, &sd_filter, CAN_RX_STANDARD_FILTER_INDEX);
        can_enable_interrupt(&can_instance, CAN_RX_FIFO_0_NEW_MESSAGE);
}
```

3.4 Extended RX Filtering (29-bit Message ID)

The following function will enable extended message filtering:

```
#define CAN_RX_EXTENDED_FILTER_INDEX       0

static void can_set_extended_filter(uint32_t filter_value)
{
        struct can_extended_message_filter_element et_filter;

        can_get_extended_message_filter_element_default(&et_filter);
        et_filter.F0.bit.EFID1 = filter_value;

        can_set_rx_extended_filter(&can_instance, &et_filter, CAN_RX_EXTENDED_FILTER_INDEX);
        can_enable_interrupt(&can_instance, CAN_RX_FIFO_1_NEW_MESSAGE);
}
```

3.5 Sending a Standard Message (11-bit Message ID)

The following function is used to send a standard message with 11-bit message identifier:

```
static void can_send_standard_message(uint32_t id_value, uint8_t *data)
{
	uint32_t i;
	struct can_tx_element tx_element;

	can_get_tx_buffer_element_defaults(&tx_element);
	tx_element.T0.reg |= CAN_TX_ELEMENT_T0_ID(id_value << 18);
	for (i = 0; i < 8; i++)
	{
		tx_element.data[i] = *data;
		data++;
	}

	can_set_tx_buffer_element(&can_instance, &tx_element, CAN_TX_BUFFER_INDEX);
	can_tx_transfer_request(&can_instance, 1 << CAN_TX_BUFFER_INDEX);
}
```

3.6 Sending an Extended Message (29-bit Message ID)

The following function is used to send an extended message with 29-bit message identifier:

```
static void can_send_extended_message(uint32_t id_value, uint8_t *data)
{
	uint32_t i;
	struct can_tx_element tx_element;

	can_get_tx_buffer_element_defaults(&tx_element);
	tx_element.T0.reg |= CAN_TX_ELEMENT_T0_ID(id_value) | CAN_TX_ELEMENT_T0_XTD;
	for (i = 0; i < 8; i++)
	{
		tx_element.data[i] = *data;
		data++;
	}

	can_set_tx_buffer_element(&can_instance, &tx_element, CAN_TX_BUFFER_INDEX);
	can_tx_transfer_request(&can_instance, 1 << CAN_TX_BUFFER_INDEX);
}
```

3.7 Interrupt Handler (Received Messages)

Messages received by the CAN controller are placed in one or two FIFOs depending on the message type (these were set using defines in the file “conf_can.h”). The files also contains defines for the TX data size, and the size of the FIFOs and buffers used by the CAN controller. The example below uses data blocks of eight bytes for both standard and extended messages.

In this example, data received in a message with a standard 11-bit identifier are placed in FIFO 0, and in FIFO 1 if the message received contains an extended 29-bit identifier.

```
void CAN0_Handler(void)
{
	uint32_t status;
	status = can_read_interrupt_status(&can_instance);

	if (status & CAN_RX_FIFO_0_NEW_MESSAGE)
	{
		can_clear_interrupt_status(&can_instance, CAN_RX_FIFO_0_NEW_MESSAGE);
		can_get_rx_fifo_0_element(&can_instance, &rx_element_fifo_0, standard_receive_index);
		can_rx_fifo_acknowledge(&can_instance, 0, standard_receive_index);
		standard_receive_index++;
		if (standard_receive_index == CONF_CAN0_RX_FIFO_0_NUM)
		{
			standard_receive_index = 0;
		}
		// Received data is in rx_element_fifo_0_data->data[x], where x is 0..7

	}

	if (status & CAN_RX_FIFO_1_NEW_MESSAGE)
	{
		can_clear_interrupt_status(&can_instance, CAN_RX_FIFO_1_NEW_MESSAGE);
		can_get_rx_fifo_1_element(&can_instance, &rx_element_fifo_1, extended_receive_index);
		can_rx_fifo_acknowledge(&can_instance, 0, extended_receive_index);
		extended_receive_index++;
		if (extended_receive_index == CONF_CAN0_RX_FIFO_1_NUM)
		{
			extended_receive_index = 0;
		}
		// Received data is in rx_element_fifo_1_data->data[x], where x is 0..7
	}
}
```

4 Summary

The firmware used as a basis for this application note is attached.

5 Revision History

Doc Rev.	Date	Comments
6493A	06/2015	Initial document release.

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**