Tamagawa Absolute Encoder Protocol Support for TMC8100

by Goeran Eggers

DESCRIPTION

The TMC8100 includes a programmable microcontroller optimized for serial synchronous and asynchronous absolute encoder protocols up to 16Mbit/s. It can replace dedicated encoder protocol interface ICs and FPGA implementations while supporting in-system updates for different encoder capabilities or for switching to other encoder protocols. The TMC8100 is a compact, cost-effective, and flexible communication solution for adding absolute encoder support to industrial drives.

This application note provides details on the TMC8100 software reference implementation supporting Tamagawa absolute encoder.

FEATURES

- Tamagawa T-Format Implementation
- Supports 2.5Mbps Absolute Encoder
- Supports Data Readout and On-the-Fly Cyclic Redundancy Check/CRC Checksum Calculation

APPLICATIONS

- Servo Drive Position Feedback
- Servo Drive Motor Control

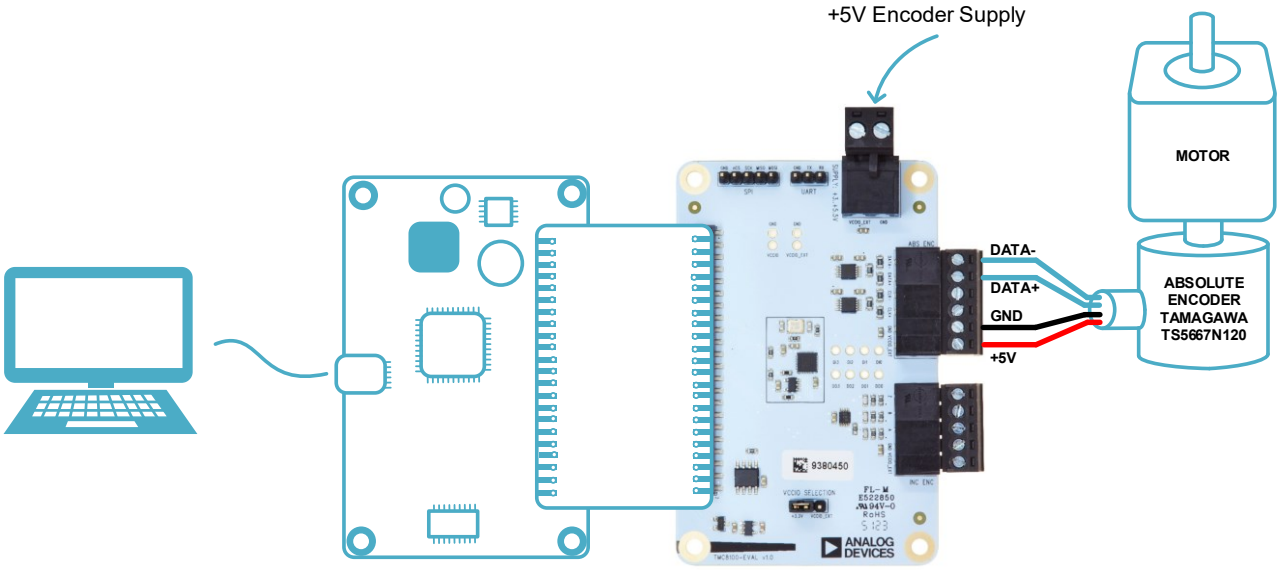


Figure 1. TMC8100-EVAL-KIT connection with Tamagawa T-Format absolute encoder

SYSTEM DESCRIPTION

Servo motor drives, e.g., in industrial applications typically require accurate, reliable, and low-latency position feedback. For a long time, optical encoders with incremental A/B/N output have been the industry standard. Nevertheless, absolute position encoders are gaining traction often with additional functionality and different interface protocols (mostly vendor-specific). An example is the T-Format protocol from Tamagawa for serial transfer of digital data between encoder and controller. The physical layer is based on the RS485 standard and capable of transmitting position values, diagnostic information, and allows reading and writing of the internal memory of the encoder.

The encoder with T-Format protocol connects to the TMC8100-EVAL-KIT through a single-shielded four-wire cable (*Figure 1*). The four wires are as follows:

- +5V and GND: encoder power supply and ground connection
- DATA+ and DATA-: differential RS485 signals for data communication (bidirectional)

The reference implementation features:

- 2.5Mbps data rate as supported by the T-Format protocol
- Send command (ID0, ID1, ID2, ID3, ID6, ID7, ID8, IDC, IDD) and receive reply
- Packing and unpacking of data
- On-the-fly calculation of CRC checksum and adding to transmit data
- On-the-fly calculation of CRC checksum and compare with received CRC

The reference implementation is available as source code. Users may use this as a starting point and apply changes as required by the application.

SYSTEM OVERVIEW

The provided software has been designed to operate in conjunction with the TMC8100-EVAL-KIT and tested with the TS5667N120 encoder from *Tamagawa*.

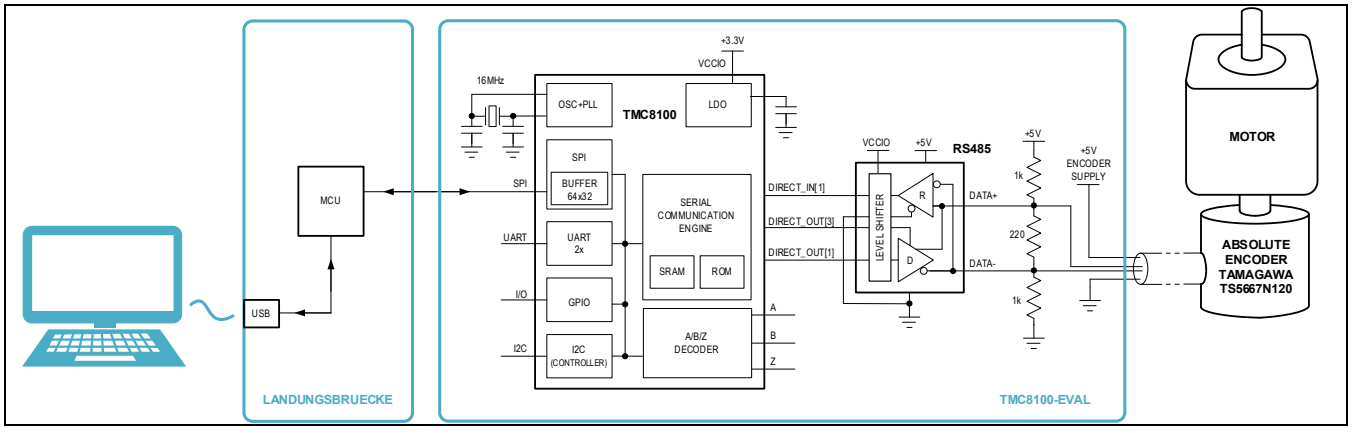


Figure 2. Core hardware components and connections

The core hardware components used from the TMC8100-EVAL-KIT are the TMC8100 and an RS485 transceiver for converting between the TMC8100 and the differential RS485 signals available as DATA+ and DATA- at the connector (*Figure 2*).

The software includes the firmware for the TMC8100 with the implementation of the controller functionality for the protocol support. This firmware must be downloaded to the TMC8100 after power-up. An additional GUI for selecting and downloading the firmware and demonstrating and testing the functionality of the firmware afterwards is available as Python script.

TAMAGAWA T-FORMAT PROTOCOL

Tamagawa is a manufacturer of encoder technology. The encoders are available with incremental or absolute position output. This reference implementation focuses on encoder with digital output providing absolute position information using RS485 line drivers. The protocol format used for communication is known as T-Format.

Communication with an encoder using the T-Format usually comprises a request sent from the controller to the encoder and a reply from the encoder back to the controller before the next request may be sent.

The communication protocol can be roughly divided into three types of requests: data readout, reset, and built-in electrically erasable programmable read-only memory (EEPROM) access. The different requests are identified by unique ID codes.

Table 1. ID Codes Supported by the Reference Implementation

TRANSACTION TYPE	ID CODE	DESCRIPTION
Data Readout	0	Absolute data in one revolution (ABS)
	1	Multiturn data (ABM)
	2	Encoder ID (ENID)
	3	ABS + ABM + ENID + Encoder error status (ALMC)
EEPROM Write	6	Write EEPROM [Address (ADF) and Data (EDF)]
EEPROM Read	D	Read EEPROM [Address (ADF)]
Reset	7	Reset ABS value
	8	Reset ABM value
	C	Reset Errors

For data readout, the TMC8100 sends a control field (CF) datagram to the encoder (*Table 2*) and the encoder replies with a copy of this CF, a status field (SF), several data fields depending on the request and a CRC checksum (CRC) (*Table 3*). For EEPROM Write, the TMC8100 sends a CF followed by EEPROM Address Field (ADF), EEPROM Data Field (EDF), and CRC. The EEPROM Read comprises Control Field (CF), EEPROM Address Field (ADF), and CRC. In both cases, the Encoder replies with EEPROM ADF, EEPROM Data Field/EDF, and CRC checksum. For a reset, the TMC8100 sends a CF and the reply always contains the absolute encoder position within one rotation (ABS).

Table 2. Encoder Request Format

ID CODE	TRANSACTION TYPE	FIELDS TRANSMITTED TMC8100			
0	Data Readout	CF			
1	Data Readout	CF			
2	Data Readout	CF			
3	Data Readout	CF			
6	EEPROM Write	CF	ADF	EDF	CRC
D	EEPROM Read	CF	ADF	CRC	
7	Reset	CF			
8	Reset	CF			
C	Reset	CF			

CF: Control Field
ADF: EEPROM Address Field
EDF: EEPROM Data Field
CRC: CRC Field

Table 3. Encoder Reply Format

ID CODE	TRANSACTION TYPE	FIELDS TRANSMITTED (ENCODER)										
0	Data Readout	CF	SF	ABS0	ABS1	ABS2	CRC					
1	Data Readout	CF	SF	ABM0	ABM1	ABM2	CRC					
2	Data Readout	CF	SF	ENID	CRC							
3	Data Readout	CF	SF	ABS0	ABS1	ABS2	ENID	ABM0	ABM1	ABM2	ALMC	CRC
6	EEPROM Write	CF	ADF	EDF	CRC							
D	EEPORM Read	CF	ADF	EDF	CRC							
7	Reset	CF	SF	ABS0	ABS1	ABS2	CRC					
8	Reset	CF	SF	ABS0	ABS1	ABS2	CRC					
C	Reset	CF	SF	ABS0	ABS1	ABS2	CRC					

ABS: Absolute Position within One Revolution
ABM: Multiturn Position
ENID: Encoder ID
ALMC: Encoder Error

As an example, the request and reply for readout of the absolute encoder position within one revolution (ID Code 0) is shown in *Figure 3*.

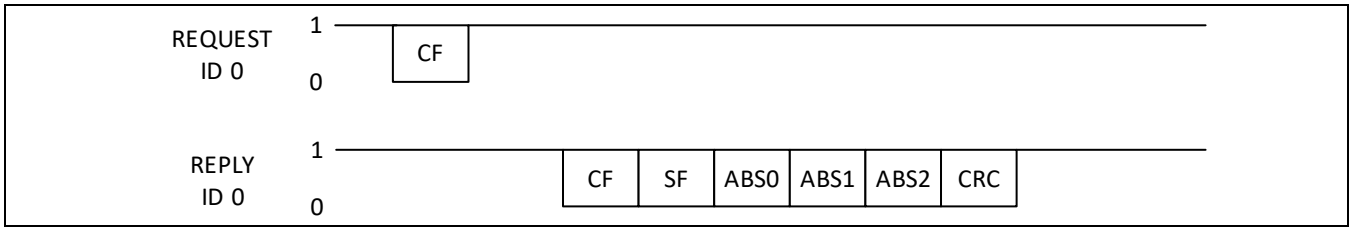


Figure 3. Frame format for data readout

Communication is asynchronous. Each field comprises 10 bits—one start bit, 8 bits of data with LSB first, and one stop bit.

The CF contains one start bit, three sync bits, the ID code (in this example, ID Code 0), and the stop bit (*Figure 4*).

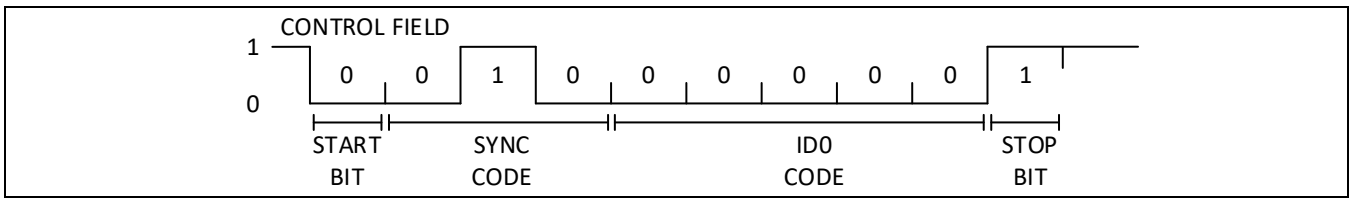


Figure 4. Control field/CF format

The absolute encoder position is split into three subsequent bytes ABS0 .. ABS2 with the least significant byte transmitted first (*Figure 5*).

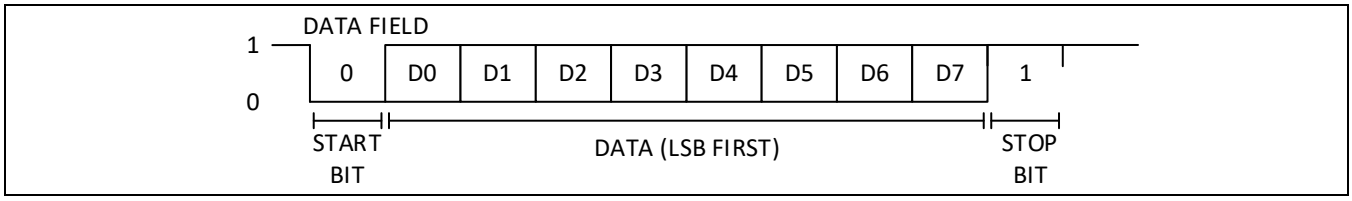


Figure 5. Data field (ABS0, ABS1, and ABS2) format

The CRC code transmitted after the last data field covers and protects the 8-bits in the CF, Status Field (SF), and data fields (ABS0, ABS1, and ABS2) without start bits and stop bits (*Figure 6*).

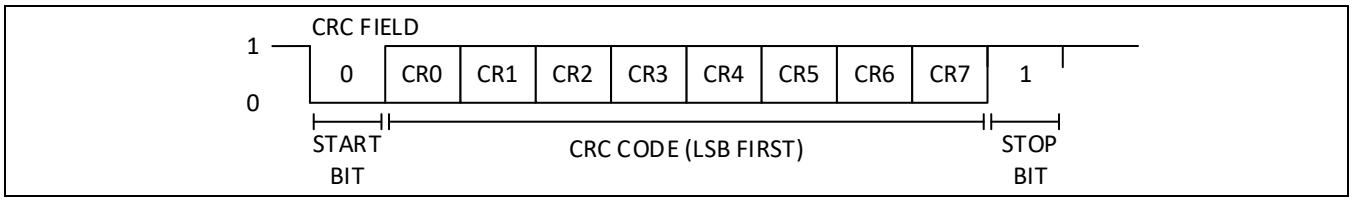


Figure 6. CRC code format

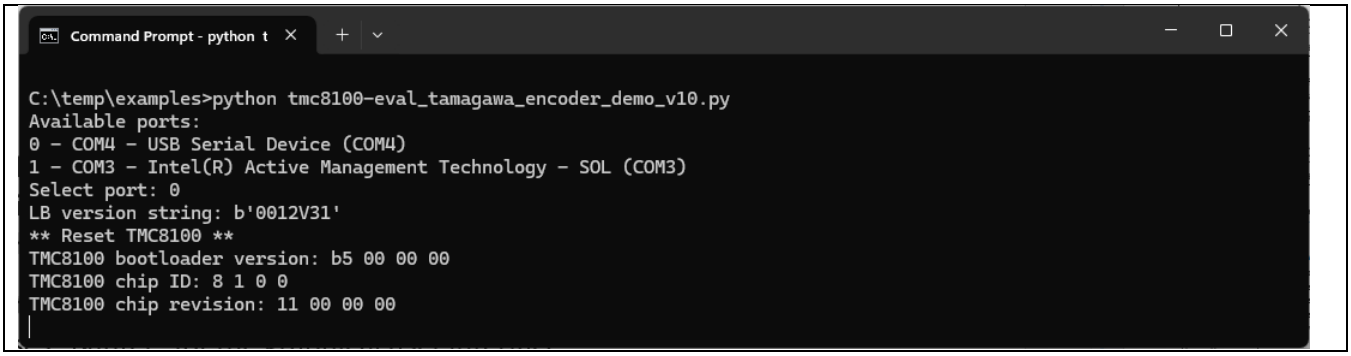
This section is intended to provide a brief overview of the T-Format protocol. For more details, refer to the specifications available from *Tamagawa*.

SOFTWARE OVERVIEW

The firmware implementation for the Tamagawa protocol for the TMC8100 is available as source code “tmc8100-eval_tamagawa_encoder_demo_v10.asm” and as machine code in intel hex file format “tmc8100-eval_tamagawa_encoder_demo_v10.hex”. For first tests/evaluation of the functionality a Python script “tmc8100-eval_tamagawa_encoder_demo_v10.py” is available. This is expected to be executed on a PC connected to the TMC8100-EVAL-KIT through USB with the encoder attached as shown in *Figure 1*.

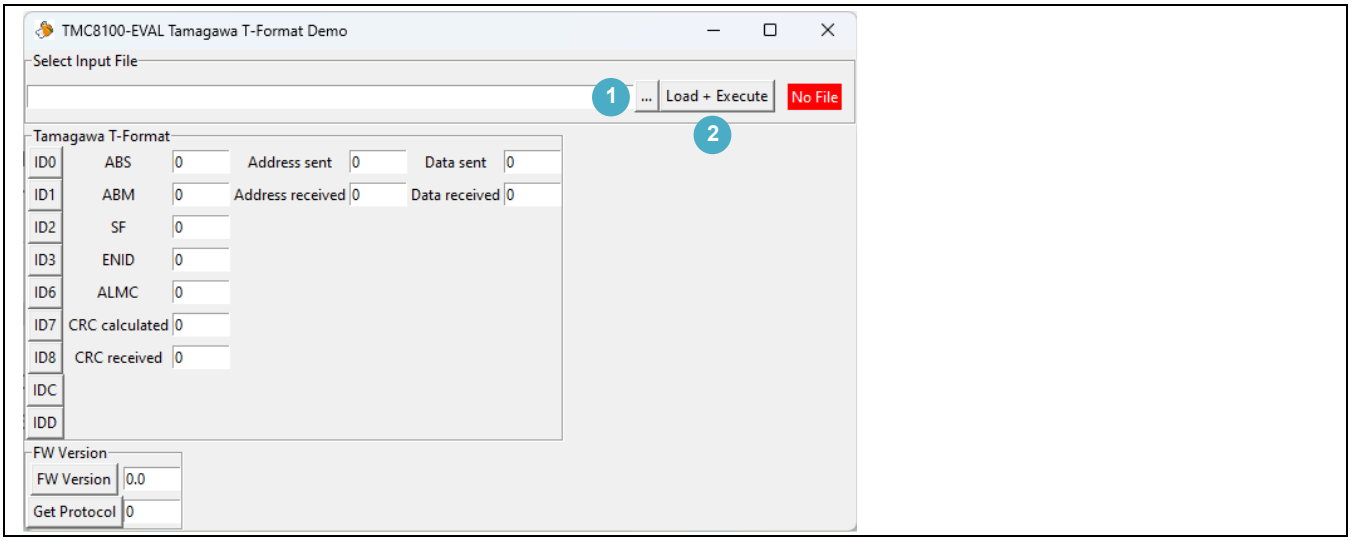
The Python script program requires a Python interpreter installed on the PC and makes use of the “intelhex” and “pySerial” Python libraries—among others. It uses “tkinter” for the graphical user interface.

After connecting the encoder to the TMC8100-EVAL-KIT, USB to the PC, and applying +5V to the TMC8100-EVAL-KIT, the Python script may be executed from the command line:

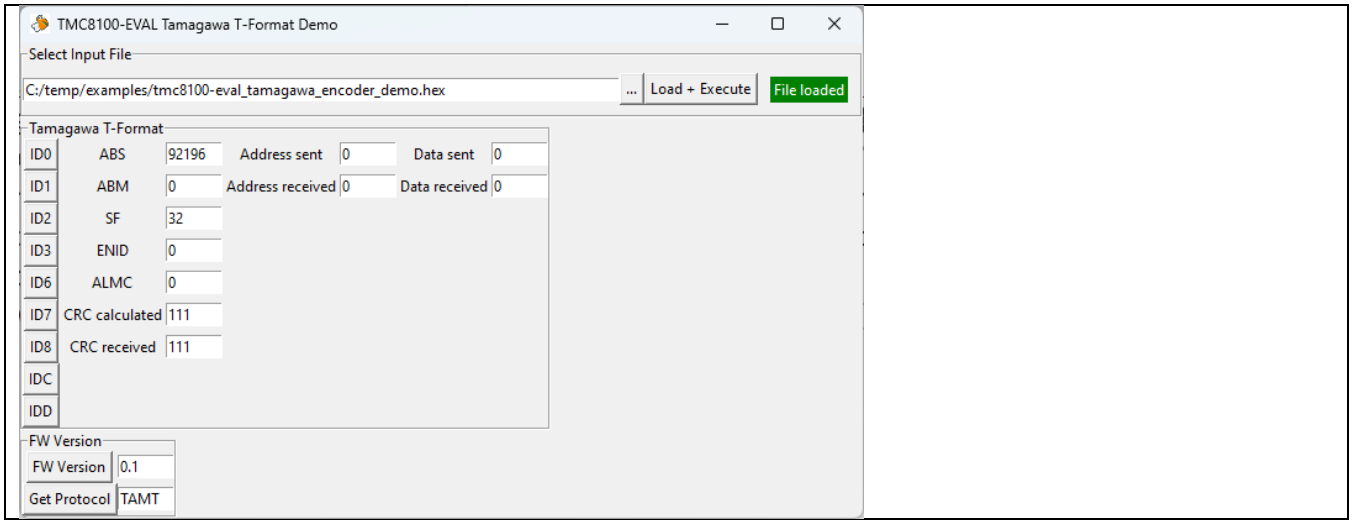


At first, the virtual COM port for the USB connection to the TMC8100-EVAL-KIT must be selected—in this example, it is “COM4”. The lines of output in the terminal window already indicate successful connection to the Landungsbruecke (LB) and detection of the TMC8100 with chip ID and revision number.

Afterwards, the GUI automatically starts in a separate window.

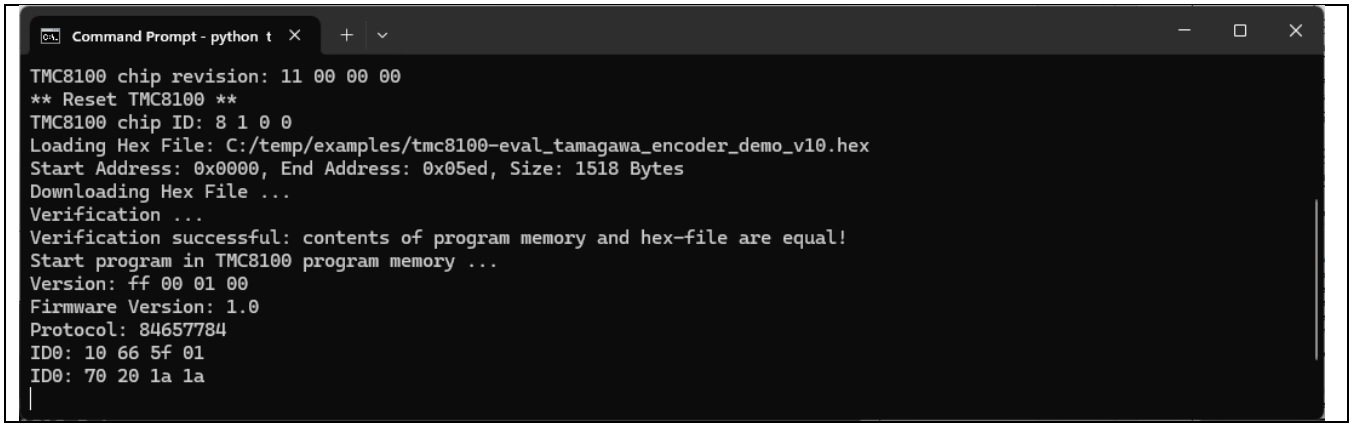


First, the hex file “tmc8100-eval_tamagawa_encoder_demo_v10.hex” with the example code for the TMC8100 must be selected with the “…” button in the Select Input File frame (1). With pressing the Load + Execute button (2) afterwards, the content of the file is written through USB and Landungsbruecke/LB into the SRAM program memory of the TMC8100 with the help of the bootloader, and program execution is started. Note that this program comes with its own communication protocol for encoder access. To put the TMC8100 into bootloader mode again, for example, to download the same or different program a reset or power cycle is required.



The Tamagawa T-Format frame in the middle of the window features the different commands (ID0 .. IDD) as separate buttons on the left. Pressing the ID0 button sends the respective command and read back the absolute position within one rotation (shown next to the "ABS" label) and the status flags (shown next to the "SF" label) from the encoder. The CRC checksum received from the encoder and the checksum calculated in parallel by the TMC8100 while receiving the serial data are shown next to the "CRC calculated" and "CRC received" labels.

In parallel to the GUI with the extracted/relevant data, the raw communication data with some additional information is shown in the command line window—which might be helpful when modifying/extending the TMC8100 program example.



FIRMWARE IMPLEMENTATION

The example source code "tmc8100-eval_tamagawa_encoder_demo_v10.asm" may be used as starting point and modified according to application requirements. An assembler (*Assembler*) is available for translation of the source code.

The flowchart (*Figure 7*) gives an overview of the example code with focus on ID code 0 command transmission and reply reception.

The firmware source code starts with definition of some values (e.g., software version and protocol supported) and the register addresses of peripheral units inside the TMC8100 for better readability.

```
SOFTWARE_VERSION_MAJOR = $01
SOFTWARE_VERSION_MINOR = $00

PROTOCOL_3 = $54 ; "T"
PROTOCOL_2 = $41 ; "A"
PROTOCOL_1 = $4D ; "M"
PROTOCOL_0 = $54 ; "T"
. . . .
; spi register address
SPI_BUFFER_0 = $30
```

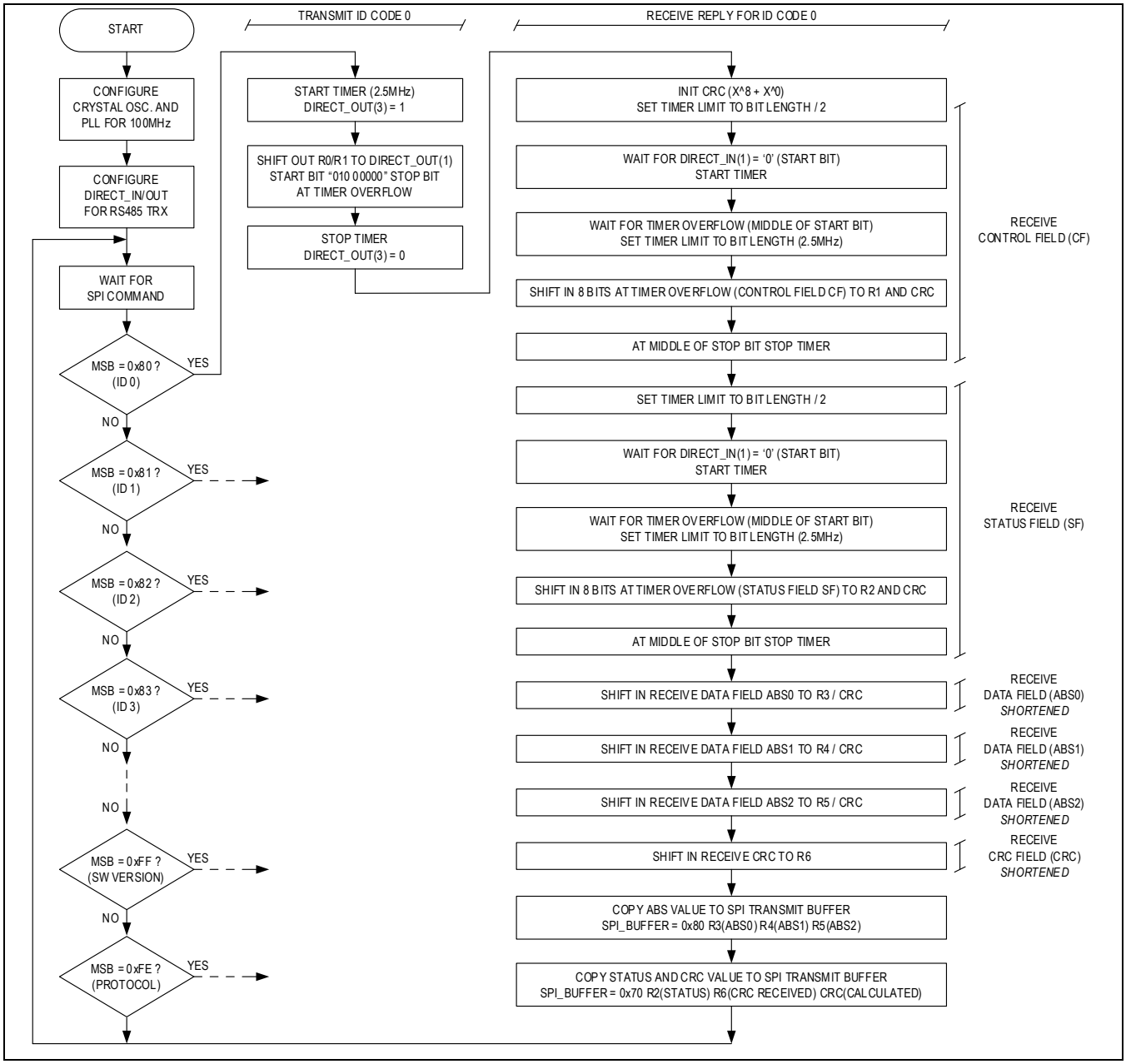


Figure 7. Firmware implementation overview

Clock Selection and Initialization

The TMC8100 always starts running on the internal oscillator and the bootloader configures the PLL for a system clock frequency of 75MHz after power-on/reset. For the Tamagawa protocol, a more precise clock source is required. In this example, the crystal oscillator is used with the 16MHz crystal available on the TMC8100-EVAL-KIT. Selecting an external clock is another option, of course. The PLL output and system frequency is set to 100MHz to simplify clock calculation/divider settings afterwards.

As the first step, pins GPIO0 and GPIO1 are configured for an external crystal in combination with the internal crystal oscillator.

```
LDI $03, r0 ; enable input for GPIO0/GPIO1
ST GPIO_IN, r0

LDI $03, r0 ; disable pull-up for GPIO0/GPIO1
ST GPIO_PU, r0
```

As the next step, the PLL feedback divider is configured for 100MHz PLL output frequency (PLL_FB_100) and the clock circuitry for the crystal oscillator (XTAL), and the PLL input divider is set to get 1MHz clock frequency at the input of the PLL. As the clock block is addressed indirectly for each register, write access four commands are necessary—a pair of load (LDI) and store (ST) instructions to set the register address first and afterwards a pair of load and store instructions to set the new register value.

```
LDI PLL_FB_CFG, r0 ; set pll feedback divider
ST CLK_ADDR, r0
LDI PLL_FB_100, r0
ST CLK_DOUT, r0 ; will trigger write access to clk register

LDI CLK_CTRL_SOURCE, r0
ST CLK_ADDR, r0
LDI $26, r0 ; use XTAL
ST CLK_DOUT, r0 ; will trigger write access to clk register

LDI CLK_CTRL_OPT, r0 ; enable clk fsm
ST CLK_ADDR, r0
LDI $40, r0
ST CLK_DOUT, r0 ; will trigger write access to clk register

LDI CLK_CTRL_PLL_CFG, r0
ST CLK_ADDR, r0
LDI %1011_1101, r0 ; RDIV = 15 (assuming 16MHz external / XTAL clock)
                   ; and select PLL output, start FSM (commit = 1)
ST CLK_DOUT, r0 ; will trigger write access to clk register
```

The last write access also triggers the internal state machine of the clock block to apply all changes. As this includes start-up of the crystal oscillator and PLL lock, it is necessary to check the status register of the clock block and wait until the new 100Mhz system clock is available. Therefore, the address of the configuration register (CLK_CTRL_PLL_CFG) of the clock block is selected as the next step and a program loop reads out this register until Bit 7 (TEST1 $7, r0) has been cleared before moving on with further program execution.

```
    LDI CLK_CTRL_PLL_CFG, r0
    ST CLK_ADDR, r0
    NOP
    NOP
WAIT_FOR_PLL:
    LD CLK_DIN, r0
    NOP
    TEST1 $7, r0
    JC WAIT_FOR_PLL
```

DIRECT_IN/DIRECT_OUT Pin Configuration

As shown in *Figure 2*, just one RS485 transceiver on the TMC8100-EVAL-KIT is used for communication with the encoder. This RS485 transceiver is controlled using DIRECT_OUT(1) for serial transmit data output, DIRECT_OUT(3) for transmitter enable, and DIRECT_IN(1) for serial data input (no configuration necessary for this input signal).

```
    ; set TXD / DIRECT_OUT(1) = 1
    SFSET WAIT1SF NO_WAIT, 0, 1
    ; set TXD_EN / DIRECT_OUT(3) = 0
    SFCLR WAIT1SF NO_WAIT, 0, 3
    ; configure TXD_EN / DIRECT_OUT(3) as output
    LDI $00, r0
    ST  DIRECT_ALT_FUNCTION, r0
```

SPI Command Loop

After configuration of the TMC8100, the program waits for commands received through the SPI in an endless loop. All SPI transactions are expected to be 32-bit datagrams. All commands of this example code fit into one datagram. For simplification, just the upper 8-bit (MSB, received first through SPI) are tested for command selection and execution. The command loop starts with reading out the status register (SPI_STATUS) of the SPI peripheral block and waiting until Bit 0 of the status register changes to one (WAIT1 $0, r0)—indicating that an SPI datagram has been received and is available in the SPI input buffer (SPI_BUFFER).

```
CMD_LOOP:
    ; wait for SPI command
    LDI SPI_STATUS, r0
    WAIT1 $0, r0
    LD SPI_BUFFER_3, r0
    ; ID0 - read encoder absolute data in one revolution
    LDI $80, r1
    COMP EQ r0, r1
    JC tamagawa_id0
    ; ID1 - read encoder multi-turn data
    . . . .
JA CMD_LOOP
```

The upper 8-bit of the contents of the 32-bit SPI input buffer are compared against $80—the SPI command defined in this example code for generating an ID0 request to the encoder—reading out the absolute position within one revolution (ABS). If this comparison is successful, the program execution jumps to address “tamagawa_id0”. The program code at this address—described in more detail in the following sections—then sends out an ID0 command datagram through DIRECT_OUT(1) and collects the reply from the encoder through DIRECT_IN(1). The received data is assembled in SPI 32-bit datagrams and put into the SPI output buffer. Simultaneously, SPI_BUFFER_FULL / GPIO changes from low to high to indicate new data available for SPI transactions. With the next SPI transactions,

this data can be read out. It is expected that a new command is not sent before all data from the previous command has been read out. As one SPI transaction always transfers data in both directions and there is usually more than one transaction necessary to read out all reply data, it is recommended to use a "dummy" command, e.g., 0x00 0x00 0x00 0x00 which is not interpreted by the command loop for all transactions but the last. The last transaction for read-out may already include the next command.

The Python script available to test the encoder implementation "tmc8100-eval_tamagawa_encoder_demo.py" with the TMC8100-EVAL-KIT offers a GUI with an **ID0** push button. Pressing this button instructs the Landungsbruecke/LB to send out an SPI datagram 0x80 0x00 0x00 0x00 (extract from "tmc8100-eval_tamagawa_encoder_demo.py"):

```
def spi_tamagawa_id0():
    # get ABS value
    value = SpiWriteCommand([0x80, 0x00, 0x00, 0x00])
    WaitForGPIO6()
    . . . .
```

Afterwards, the program waits for the reply of the encoder. The pin SPI_DATA_AVAILABLE / GPIO6 switches from low to high as soon as reply data is available in the SPI output buffer of the TMC8100. For reading out this data, the Python program uses the SPI "dummy" commands 0x00 0x00 0x00 0x00 not interpreted by the TMC8100 firmware.

```
    # get ABS value
    value = SpiWriteCommand([0x00, 0x00, 0x00, 0x00])
    print(f"ID0: {value[0]:02x} {value[1]:02x} {value[2]:02x} {value[3]:02x}")
    . . . .
    # get CRC + SF value
    value = SpiWriteCommand([0x00, 0x00, 0x00, 0x00])
    print(f"ID0: {value[0]:02x} {value[1]:02x} {value[2]:02x} {value[3]:02x}")
    . . . .
```

The following table provides an overview of all SPI commands supported by the example program and reply data available in return. SPI commands always fit into one 32-bit datagram while the reply may take up to five 32-bit datagrams. SPI 32-bit datagrams are given as 4 consecutive hex numbers—one hex number per byte with the MSB first for better readability. For the reply, the 32-bit datagrams are shown in the order they are put into the SPI buffer/can be read out.

SPI COMMAND (32-BIT)	SPI REPLY (32-BIT)
0x80 0x00 0x00 0x00 ID code 0 request - read encoder absolute data within one rotation	1. 0x10 ABS0 ABS1 ABS2 2. 0x70 SF CRC received CRC calculated
0x81 00 00 00 ID code 1 request – read encoder multiturn data	1. 0x20 ABM0 ABM1 ABM2 2. 0x70 SF CRC received CRC calculated
0x82 00 00 00 ID code 2 request – read encoder ID	1. 0x40 0x00 0x00 ENID 2. 0x70 SF CRC received CRC calculated
0x83 00 00 00 ID code 3 request – read encoder absolute, multiturn data, ID and error flags	1. 0x10 ABS0 ABS1 ABS2 2. 0x40 0x00 0x00 ENID 3. 0x20 ABM0 ABM1 ABM2 4. 0x50 0x00 0x00 ALMC 5. 0x70 SF CRC received CRC calculated
0x86 00 EDF ADF	1. 0x50 0x00 EDF ADF 2. 0x70 SF CRC received CRC calculated

ID code 6 request – write 8-bit data to 7-bit address	
0x87 00 00 00 ID code 7 request – reset error flags	1. 0x10 ABS0 ABS1 ABS2 2. 0x70 SF CRC received CRC calculated
0x88 00 00 00 ID code 8 request – reset encoder absolute position counter	1. 0x10 ABS0 ABS1 ABS2 2. 0x70 SF CRC received CRC calculated
0x8c 00 00 00 ID code C request – reset encoder multiturn counter	1. 0x10 ABS0 ABS1 ABS2 2. 0x70 SF CRC received CRC calculated
0x8d 00 00 ADF ID code D request - read 8-bit data from 7-bit address	1. 0x50 0x00 EDF ADF 2. 0x70 SF CRC received CRC calculated
0xff 00 00 00 get firmware version	1. 0xff 0x00 VERSION_MAJOR VERSION_MINOR
0xfe 00 00 00 get encoder protocol	1. 0x54 (“T”) 0x41 (“A”) 0x4d (“M”) 0x54 (“T”)

ID Code 0 Request

As soon as a new SPI command has been received while the TMC8100 software is waiting for SPI commands and the upper byte/the byte received first from the 32-bit datagram is equal to $80, the program execution jumps to address “tamagawa_id0” in program code and starts with preparing and transmitting ID code 0 request through DIRECT_OUT(1) and attached RS485 transceiver toward the encoder.

```
; set timer to 100MHz / 40 = 2.5MHz
LDI 39, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_W
LDI 2, r0 ; enable timer
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_CTRL_W
; switch on driver
SFSET WAIT1SF NO_WAIT, 0, 3
. . . .
```

For the encoder protocol, the required transmission data rate is 2.5Mbit/s. As the internal system clock has been set to 100MHz, the system timer used for shifting out the bits is set to 39 to get to a division factor of 100/2.5 = 40. Afterwards, the timer is enabled and the external RS485 driver is switched on by setting DIRECT_OUT(3) output.

```
; start bit + sync code
LDI %0000_0100, r0
; id0 + parity
LDI %0000_0000, r1
REP 4, 1
SHRO WAIT1SF WAIT_OVERFLOW_TIMER, r0, FLAG_OUT1 ; shift out start bit + sync code
REP 5, 1
SHRO WAIT1SF WAIT_OVERFLOW_TIMER, r1, FLAG_OUT1 ; shift out ID0 + parity
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_NO_ACTION    ; wait time for last bit before stop bit
; stop bit
SFSET WAIT1SF NO_WAIT, 0, 1
```

```
. . . .
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_STOP_TIMER
; end of command - switch off driver
SFCLR WAIT1SF NO_WAIT, 0, 3
```

After waiting some time for the output of the RS485 transceiver to stabilize the start bit (“0”) and 3-bit sync code (“010”) it is shifted out from system register r0 on DIRECT_OUT(1) using the SHRO instruction. This instruction is executed four times using the REP 4, 1. Each time, it waits until the system timer wraps around (parameter WAIT_OVERLFLOW_TIMER) and then shifts out the LSB from r0 register. Afterwards, the remaining 5 bits of the datagram with the ID code 0 and a parity bit are shifted out in a similar manner—this time using system register r1 and repeating the SHRO instruction five times by using REP 5, 1. The DIRECT_OUT(1) serial output then switches to “1” for the stop bit and finally DIRECT_OUT(3) switched low to turn off the RS485 transceiver and prepare for receiving data from the encoder.

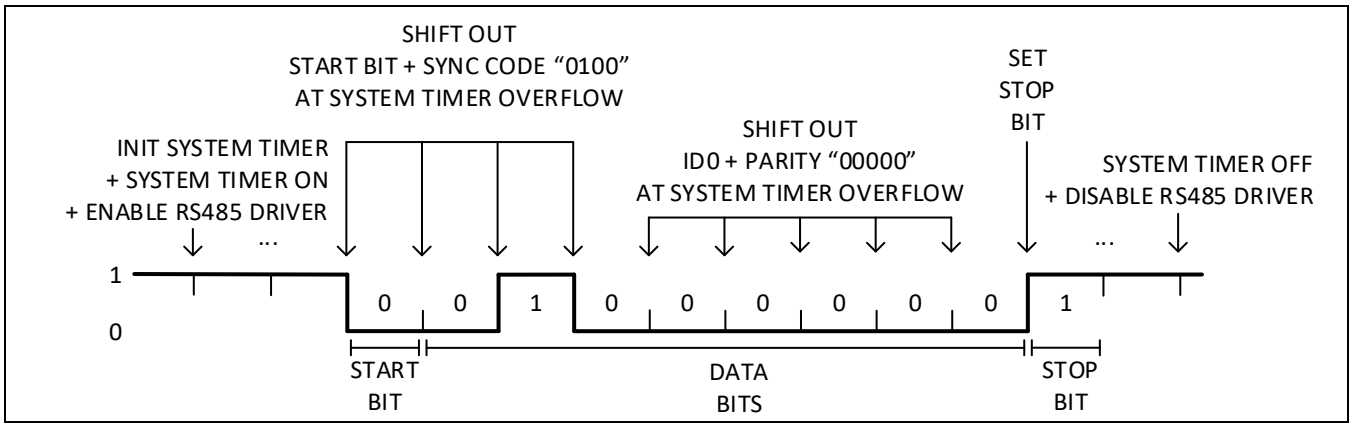


Figure 8. ID0 transmit sequence

ID Code 0 Reply

To prepare for the reply, the CRC calculation block is initialized to enable on-the-fly CRC calculation while the serial data is shifted in.

```
; initialize CRC for 8bit CRC
LDI $0, r0
STS r0, SYSTEM_CRC, SYSTEM_CRC_CTRL_W ; reset CRC block
LDI %0000_0001, r0 ; CRC polynomial: x^8 + x^0
STS r0, SYSTEM_CRC, SYSTEM_CRC_POLYNOM_W ; LSB
LDI %0000_0001, r0 ; CRC polynomial: x^8 + x^0
STS r0, SYSTEM_CRC, SYSTEM_CRC_POLYNOM_W
LDI %0000_0000, r0
STS r0, SYSTEM_CRC, SYSTEM_CRC_POLYNOM_W
STS r0, SYSTEM_CRC, SYSTEM_CRC_POLYNOM_W ; MSB
```

First, the linear feedback shift register (LFSR) used for CRC calculation is reset and the polynom used for CRC for this protocol is loaded into the system CRC register (SYSTEM_CRC_POLYNOM_W). This register is 32-bit in size filled with four byte wide write accesses after CRC block reset to the same register with the LSB first. Bit 0 and bit 8 have to be set to configure the generator polynomial (x^8 + x^0) used for the encoder protocol.

The program then prepares for the first datagram from the encoder (control field—CF).

```
; reset timer and set timer limit to middle of bit
LDI 18, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_W
; wait for falling edge (start bit)
WAIT0SF WAIT_IN1, WAIT_START_TIMER
; wait for middle of start bit
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_NO_ACTION
; set limit to 100MHz / 39 = 2.5MHz and no timer reset
LDI 39, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_NO_RESET_W
REP 8, 1
; wait for timer overflow and shift in sync code 010 and cc0..cc4
SHRI WAIT1SF WAIT_OVERFLOW_TIMER, FLAG_IN1_CRC, r1 ; CF -> r1
; wait for timer overflow, stop bit
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_STOP_TIMER
```

The system timer is configured and the program waits (WAIT0SF) for the falling edge of the start bit of the reply message from the encoder on DIRECT_IN(1) and immediately starts the timer (WAIT_START_TIMER). The timer has been programmed to overflow at approximately half the time than before (limit set to 18 instead of 39) to get to the middle of the start bit. As soon as this position has been reached, the limit is set to 39 again to generate a timer overflow event each time the middle of an incoming data bit has been reached (*Figure 9*). The following 8 bits of incoming data are then shifted into system register r1 with the LSB first and, simultaneously, into the CRC unit for CRC calculation using the repeat command REP 8, 1 followed by shift right in command SHRI. As soon as the middle of the final stop bit has been reached, the timer is stopped. The received control field data is now available in register r1 and the CRC checksum calculated for the first 8 bits received.

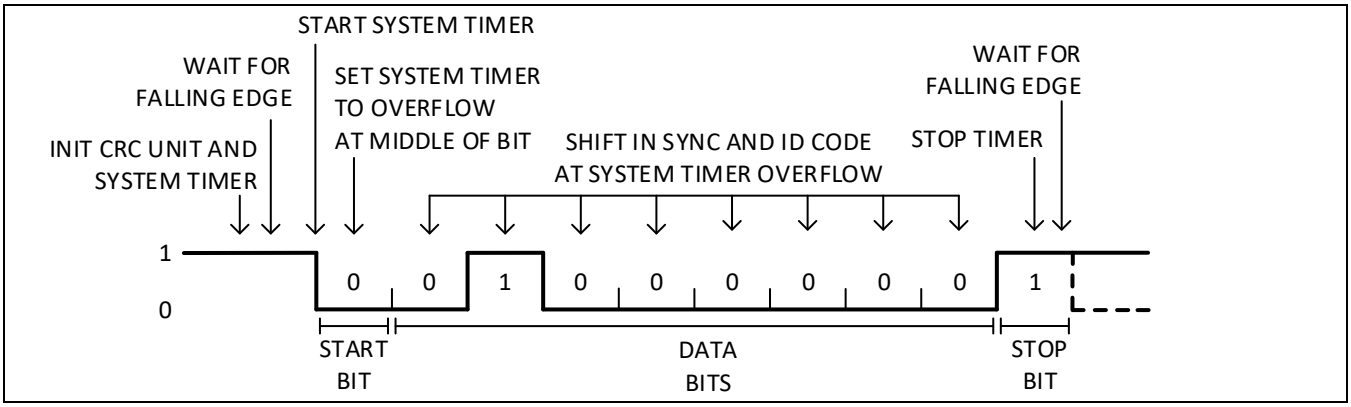


Figure 9. ID0 reply control field receive sequence

The second datagram with the second byte expected from the encoder contains the status field (SF). Again, the timer overflow is set to the middle of the start bit and the timer is started as soon as a falling edge is detected on DIRECT_IN(1).

```
; reset timer and set timer limit to middle of bit
LDI 18, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_W
; wait for falling edge (start bit)
WAIT0SF WAIT_IN1, WAIT_START_TIMER
; wait for middle of start bit
```

```
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_NO_ACTION
; set limit to 100MHz / 39 = 2.5MHz and no timer reset
LDI 39, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_NO_RESET_W
REP 8, 1
; wait for timer overflow and shift in dd0..dd3, ea0..ea1, ca0..ca1
SHRI WAIT1SF WAIT_OVERFLOW_TIMER, FLAG_IN1_CRC, r2 ; SF -> r2
; wait for timer overflow, stop bit
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_STOP_TIMER
```

The 8 data bits are shifted into register r2 and, simultaneously, into the CRC unit using the REP instruction in combination with the SHRI instruction. The timer is stopped when the middle of the stop bit has been reached. At the end, the received status field is available in register r2 and the CRC checksum now calculated over the 8 bits of the control field and the 8 bits of the status field.

Next, the 24-bit absolute encoder position value within one rotation is expected divided into three datagrams with one byte each and the LSB first (ABS0, ABS1, ABS2). The structure of the program code is very similar to the previous one for the status field datagram.

```
; reset timer and set timer limit to middle of bit
LDI 18, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_W
; wait for falling edge (start bit)
WAIT0SF WAIT_IN1, WAIT_START_TIMER
; wait for middle of start bit
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_NO_ACTION
; set limit to 100MHz / 39 = 2.5MHz and no timer reset
LDI 39, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_NO_RESET_W
REP 8, 1
; wait for timer overflow and shift in d0..d7
SHRI WAIT1SF WAIT_OVERFLOW_TIMER, FLAG_IN1_CRC, r3 ; ABS0 -> r3
; wait for timer overflow, stop bit
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_STOP_TIMER
```

The encoder position is stored in system register r3 (ABS0), r4 (ABS1), and finally r5 (ABS2). The CRC checksum calculation is continued to also cover these three bytes.

As the final datagram from the encoder for ID code 0, the CRC checksum is received. The program code for this datagram is again very similar to the previous one.

```
; reset timer and set timer limit to middle of bit
LDI 18, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_W
; wait for falling edge (start bit)
WAIT0SF WAIT_IN1, WAIT_START_TIMER
; wait for middle of start bit
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_NO_ACTION
; set limit to 100MHz / 39 = 2.5MHz and no timer reset
LDI 39, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_LIMIT_NO_RESET_W
```

```
REP 8, 1
; wait for timer overflow and shift in rc0..rc7
SHRI WAIT1SF WAIT_OVERFLOW_TIMER, FLAG_IN1, r6 ; CRC -> r6
; wait for timer overflow, stop bit
WAIT1SF WAIT_OVERFLOW_TIMER, WAIT_STOP_TIMER
```

Note that this time the bits shifted into the register with the SHRI command are not shifted into the CRC unit in parallel (FLAG_IN1 instead of FLAG_IN1_CRC as last parameter of the shift command SHRI). Afterwards, the received CRC value is available in system register r6.

All data received is now available in system registers r1 to r6. As next step, these values are copied to the SPI buffer for read-out by the attached microcontroller/motion controller.

```
; copy reply to SPI transmit buffer
LDI %0001_0000, r0 ; its the ABS value
ST SPI_BUFFER_3, r0 ; 0x10 -> ABS value
ST SPI_BUFFER_2, r3 ; ABS0
ST SPI_BUFFER_1, r4 ; ABS1
ST SPI_BUFFER_0, r5 ; ABS2
```

The first 32-bit SPI datagram is filled up with the absolute position information in the lower 24-bit and a fixed value of 0x10 as MSB. When the SPI datagram is read out, this fixed MSB value can be used to clearly identify the contents of this datagram.

```
; copy crc value to SPI transmit buffer
LDI %0111_0000, r0 ; crc values
ST SPI_BUFFER_3, r0 ; 7 -> CRC + STATUS Field
ST SPI_BUFFER_2, r2 ;
LDS SYSTEM_CRC, SYSTEM_CRC_RESULT0_R, r0
ST SPI_BUFFER_1, r6 ; CRC received
REV r0, r0
ST SPI_BUFFER_0, r0 ; CRC calculated
JA CMD_LOOP
```

The second 32-bit SPI datagram contains the calculated CRC value from the CRC system block as LSB. The calculated CRC is copied to system register r0 using the instruction LDS SYSTEM_CRC, SYSTEM_CRC_RESULT0_R, r0. The bits of the calculated CRC value are in reverse order compared to the received CRC value. The instruction REV r0, r0 is used to reverse the register bits. After that the calculated CRC checksum in register r0 should be identical to the received CRC for any undisturbed data transmission.

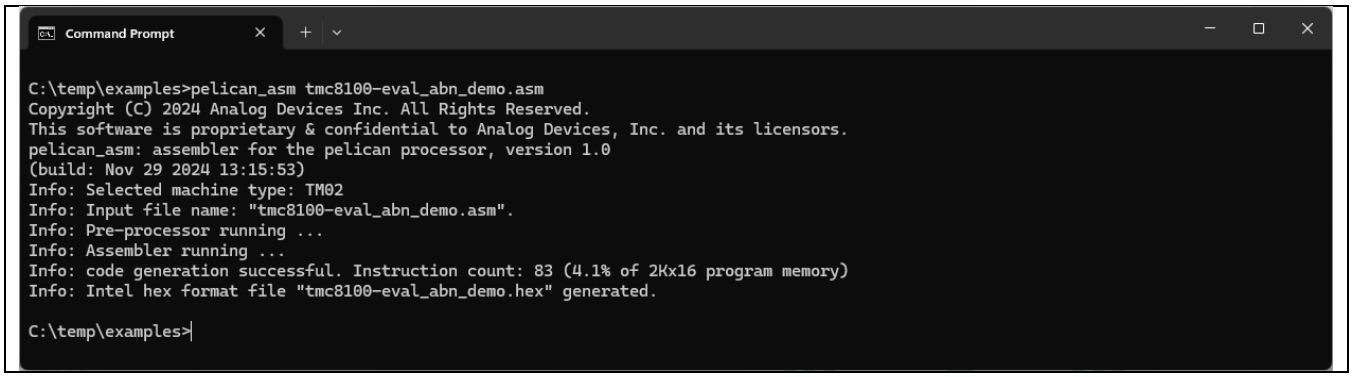
For practical applications, the calculated and received CRC checksums would be compared. In case they do not match the whole datagram/encoder, the information may be dismissed/ignored, e.g., an error flag set.

In this example code, the LSB of the 32-bit SPI datagram is filled with the calculated CRC and the next higher byte with the received CRC code. The received status field bits (SF) are copied to the next higher byte of the datagram. The MSB of the datagram is finally set to the fixed value 0x70 to identify the contents of the datagram before it is copied to the SPI output buffer.

APPENDIX

Assembler

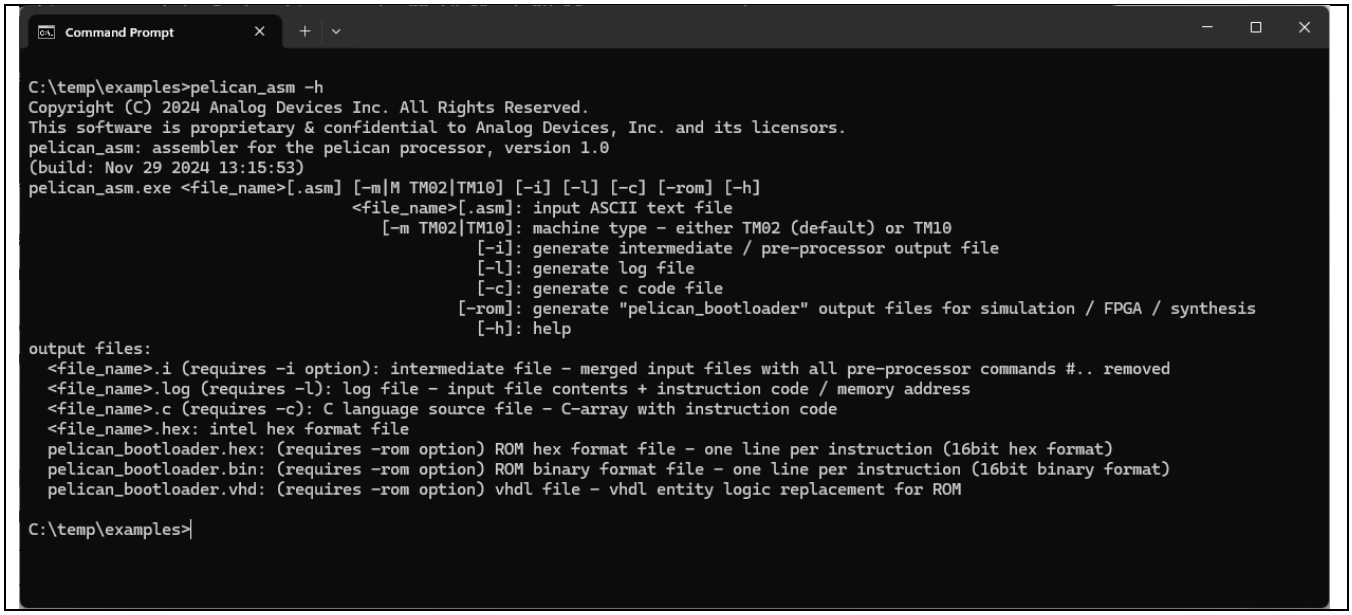
For program development and modification of the examples/reference programs for the TMC8100, an assembler is available. This PC-based command line tool expects as parameter the file name and optional path of the assembler source code file. In case the file with the provided file name is not found, the ending *.asm is added automatically to the provided file name. From this input file, the assembler generates the output file(s).



In this example, the name of the input file with the assembly source code is "tmc8100-eval_abn_demo.asm". The number of machine instructions (16-bit) generated from this file is 83 which is 4.1% of the available program memory (SRAM). The size of the program memory in TMC8100 is 2K x 16-bit—a maximum of 2048 instructions.

There is one generated default output file which has the name of the input file with *.hex as extension instead of *.asm. This output file is a text file with the program code in standard intel hex file format, e.g., for loading the program code into the program memory of the TMC8100 on the TMC8100-EVAL-KIT using the TMCL-IDE or one of the Python scripts available with the example code.

The assembler also supports a number of flags for generating additional output files in different formats and additional information.



- Option “-i”: file “<filename>.i” is generated as output from the pre-processor. Contains the contents of the assembly source file with all pre-processor commands (# ...) being processed (e.g., “#include” file contents merged into the source file) and comments (e.g., /* .. */ | // | ;) removed.
- Option “-l”: file “<filename>.log” is generated with the assembly source code (without comments and pre-processor commands) with additional instruction encoding and the program memory address of the instructions
- Option “-c”: file “<filename.c>” is generated with C-code to support program development for the controller used to bootstrap the TMC8100. This file contains an array of integer numbers with the generated machine code.
- Option “-rom”: 3 additional files is generated—"pelican_bootloader.hex" and "pelican_bootloader.bin" with machine instructions listed as hex/binary numbers—one number per line and "pelican_bootloader.vhd"—with machine instructions as logic code as part of a vhdl entity declaration.
- Option “-m”—machine flag. For support of different implementations/instruction set (TM02 covers the TMC8100).
- Option “-h”—print help test as shown in the screenshot

Syntax/Commands Supported

The assembler currently supports all TMC8100 instructions as described in the TMC8100 data sheet.

Pre-Processor Commands

Comments

The pre-processor removes all comments from the input file(s) for further processing. Currently, the following options are supported:

- **/*** <comment> ***/** - block comment - can include more than one line (C-language style)
- **//** <comment> - comment until end of line (C/C++ - language style)
- **;** <comment> - comment until end of line

#include

#include "<filename>"—contents of file given with name and optional path in <filename> is inserted at the position of the #include pre-processor directive for further processing. The line with the #include directive is removed.

Note: Only quotation marks are allowed around <filename> and there should be no other commands/assignments within this line as they are removed/ignored from further processing.

#define

#define <label> [<replacement text>]

The pre-processor replaces any <label> found in the source file (+ files included with the #include statement) after this command with <replacement text>. As replacement text, the character sequence after <label> separated with at least one space from <label> until end of line (or start of comment) is taken. Only comments and text in quotation marks are excluded from automatic replacement. Label(s) may be redefined afterwards in the source file using another #define with the same <label>.

It is possible to define a <label> without replacement text. In this case, the replacement text is empty. This might be useful, e.g., in combination with the conditional #ifdef / #ifndef pre-processor commands.

#ifdef, #ifndef, #else, #endif

#ifdef <label> <code block 1> #else <code block 2> #endif

In case <label> has been defined earlier, the contents of <code block 1> are interpreted and assembler output generated and <code block 2> is ignored. The intermediate file ("<filename>.i") just shows <code block 1> and not <code block 2>. The code blocks may contain several lines of assembler instructions, etc. In case <label> has not been defined, it is the other way round.

#ifndef <label> <code block 1> #else <code block 2> #endif

In case <label> has been defined earlier, the contents of <code block 2> are interpreted and assembler output generated and <code block 1> is ignored. The intermediate file (<filename>.i) just shows <code block 2> and not <code block 1>. The code blocks may contain several lines of assembler instructions, etc. In case <label> has not been defined, it is the other way round.

Note: It is sufficient to mention the <label> name before using #define <label>—it is not necessary to provide a replacement text/value.

The #else part is optional. #ifdef or #ifndef blocks may be nested.

Assembler Commands

Numbers

To simplify specification of numbers as binary, decimal, and hexadecimal, different formats are supported. These are as follows:

0x123.. or $123.. are interpreted as hexadecimal numbers (character expected: 0-9, a-f or A-F)

%1010.. is interpreted as binary number (characters expected: 0 and 1). The character '_' may be inserted for better readability - e.g., %1010_0011 for an 8-bit number

123.. is interpreted as decimal number (characters expected: 0-9)

Identifiers

Identifiers may be used for better readability instead of numbers. Identifiers must start with a letter (a-z / A-Z) or '_'. Afterwards, numbers are also allowed (0-9). Note that character names are not case sensitive.

Identifiers may be assigned a value using the equal sign – for example,

SPI_BUFFER_0 = $30

Some names should not be used as identifiers:

- assembler instructions - This includes all names listed in the TMC8100 data sheet and also the instruction names preceded with a 'C' - indicating conditional execution of the command.
- Identifier r0..r7 are pre-defined as register 0..7 for specifying all general-purpose registers available
- labels and identifiers should not have the same name

Labels

Labels are used as placeholders for program memory addresses. Labels do not have to be assigned a value—they are initialized with the current program memory address while the assembler translates the assembly source into machine code.

Example for an endless loop:

```
WAIT:
  JA WAIT
```

Note the ':' character after the label name. The assembler automatically initializes the label WAIT with the program memory address of the next instruction—which in this case is the JA WAIT command. Jump backs (as shown above) where the label is initialized before it is actually used as part of an instruction and jump forwards—where the label is initialized after it is referenced with an instruction—are both supported.