

## SSI 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 field programmable gate array/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.

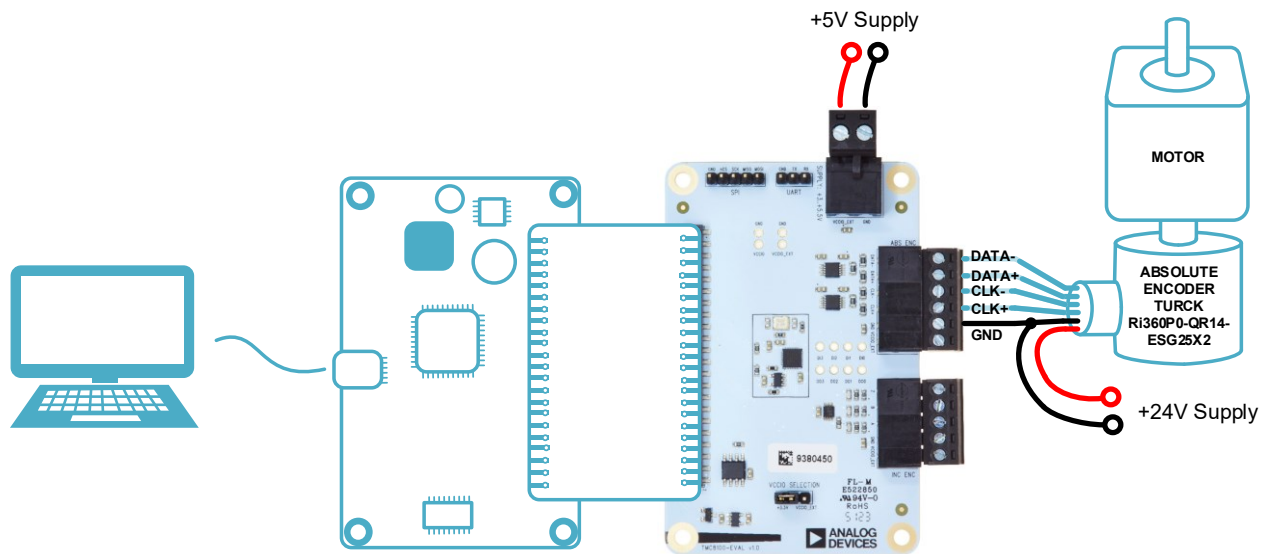
This application note provides details on the TMC8100 software reference implementation supporting absolute position encoder with synchronous serial interface (SSI).

### FEATURES

- SSI Implementation
- Example Uses 1MHz Clock Frequency
- Gray to Binary Code Conversion
- Input Filter
- SSI Clock Frequencies up to 16MHz Possible

### APPLICATIONS

- Motor Position Feedback



*Figure 1. TMC8100-EVAL-KIT connection shows absolute position encoder with SSI*

### 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 SSI protocol for synchronous serial transfer of digital data between encoder and controller. It uses two separate connections – one for the clock generated by the controller and one for the encoder data shifted out from the encoder. The physical layer is based on the RS422 standard and capable of transmitting position values and diagnostic information from encoder to controller.

The encoder used in this example with an SSI connects to the TMC8100-EVAL-KIT through a single cable with six wires being used (*Figure 1*). The six wires are as follows:

- +24V and GND: encoder power supply and ground connection
- DATA+ and DATA-: differential RS-422 signals for transmission of data from encoder to controller
- CLK+ and CLK-: differential RS-422 signals for transmission of clock from controller to encoder

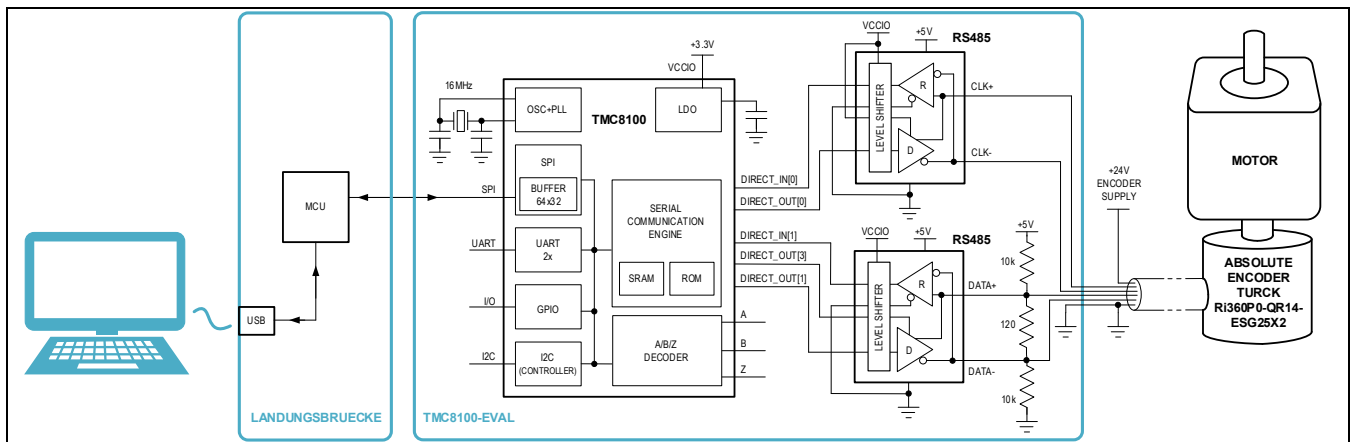
The reference implementation features:

- Generating of clock signal with the required frequency, number of pulses, and polarity
- 1Mbps data rate as supported by the SSI encoder
- Packing and unpacking of data
- Conversion of 16-bit encoder position data from gray code to 16-bit binary code

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 Ri360P0-QR14-ESG25X2 encoder (*Turck*) with SSI.



**Figure 2. Core hardware components and connections**

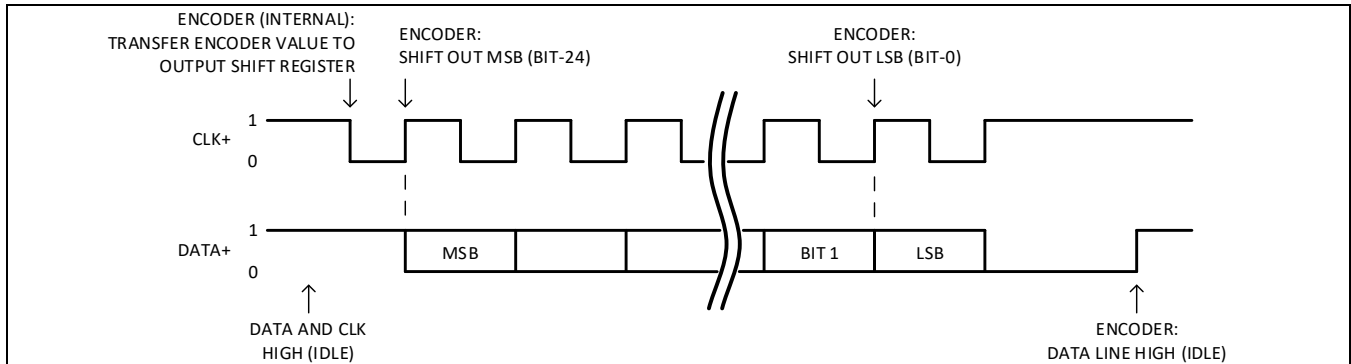
The core hardware components required from the TMC8100-EVAL-KIT are the TMC8100 and the two RS485 transceivers for converting between the TMC8100 and the differential RS485 signals available as DATA+, DATA- and CLK+, CLK- at the connector (*Figure 2*).

The software includes the firmware for the TMC8100 with the implementation of the controller functionality for the SSI 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 together with an encoder is available as Python script.

## SYNCHRONOUS SERIAL INTERFACE PROTOCOL

The SSI uses a clock signal generated by the controller and used as trigger/reference by the encoder to shift out the encoder position data and additional information (e.g., status/error). In idle state both—the clock signal from controller to encoder and the data signal back from encoder to controller—are at high level/logic ‘1.’



**Figure 3. SSI Datagram: Serial transmission of encoder data from most significant bit/MSB to least significant bit/LSB**

With the first falling edge of the clock signal, the encoder data is transferred to the output shift register inside the encoder. With the following rising edges of the clock signal, this data is then shifted out with the MSB first and the LSB last. In this example, 25 bits of data are transferred. Two upper bits (23, 22) contain diagnostic bits while the lower 16-bit contain the encoder position information encoded in gray code ([Table 1](#)). After transfer of data, the data line is kept at low level for a fixed amount of time before it is switched back to high level/idle state. Afterwards, read-out of the next encoder position may start with a falling level of the clock line.

**Table 1. SSI Datagram Bit Description**

DATA BITS	DESCRIPTION
Bit 24 (MSB)	First bit transferred – not used
Bit 23	Positioning element is outside the measuring range
Bit 22	Positioning element is in the measuring range, lower signal quality (e.g. distance too large)
Bit 21 .. Bit 16	Not used
Bit 15 .. Bit 0	Encoder position information (Gray coded)

The same encoder position may be read-out twice with a falling edge of the clock starting the next SSI datagram already while the data line is still low after transmission of the LSB. This might be useful to check encoder data integrity as the datagram does not include any checksum in this example.

While the number of data bits is 25, the number of clock cycles is actually 26—one more than the number of data bits due to the fact that the first data bit is shifted out with the first rising and not the falling edge of the clock and the last data bit with the rising edge preceding the last one.

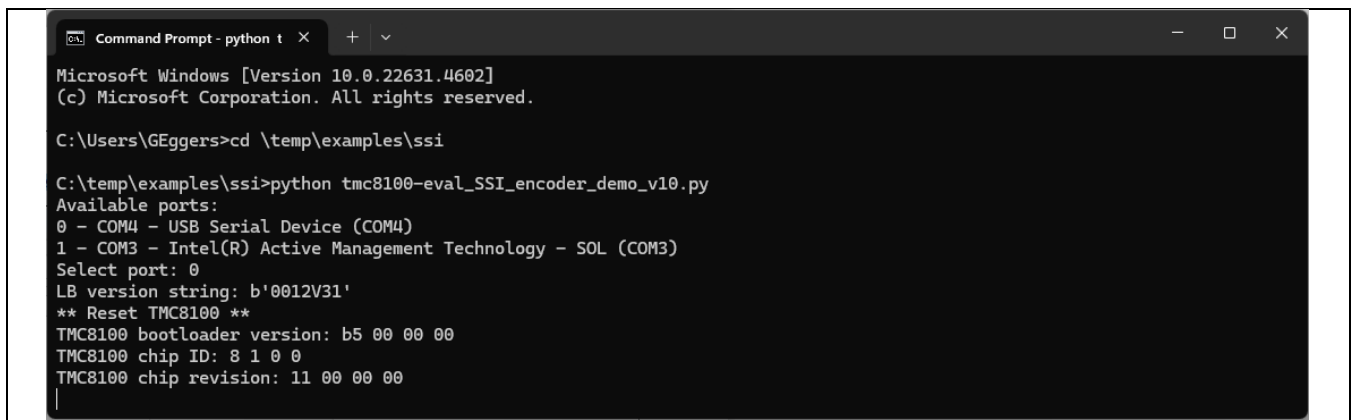
There are a lot of different encoders and encoder manufacturers supporting the SSI protocol with often different number of datagram bits, number of bits used for the encoder position information, data encoding (e.g., binary instead of Gray code), additional information (e.g., status/error bits, multiturn data), and checksum (e.g., parity/CRC). Refer to the documentation provided by the encoder manufacturer on specific protocol implementation details.

## SOFTWARE OVERVIEW

The firmware implementation of the SSI protocol for the TMC8100 is available as source code “tmc8100-eval\_ssi\_encoder\_demo\_v10.asm” and as machine code in intel hex file format “tmc8100-eval\_ssi\_encoder\_demo\_v10.hex”. For first tests/evaluation of the functionality, a Python script “tmc8100-eval\_ssi\_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:



```

Microsoft Windows [Version 10.0.22631.4602]
(c) Microsoft Corporation. All rights reserved.

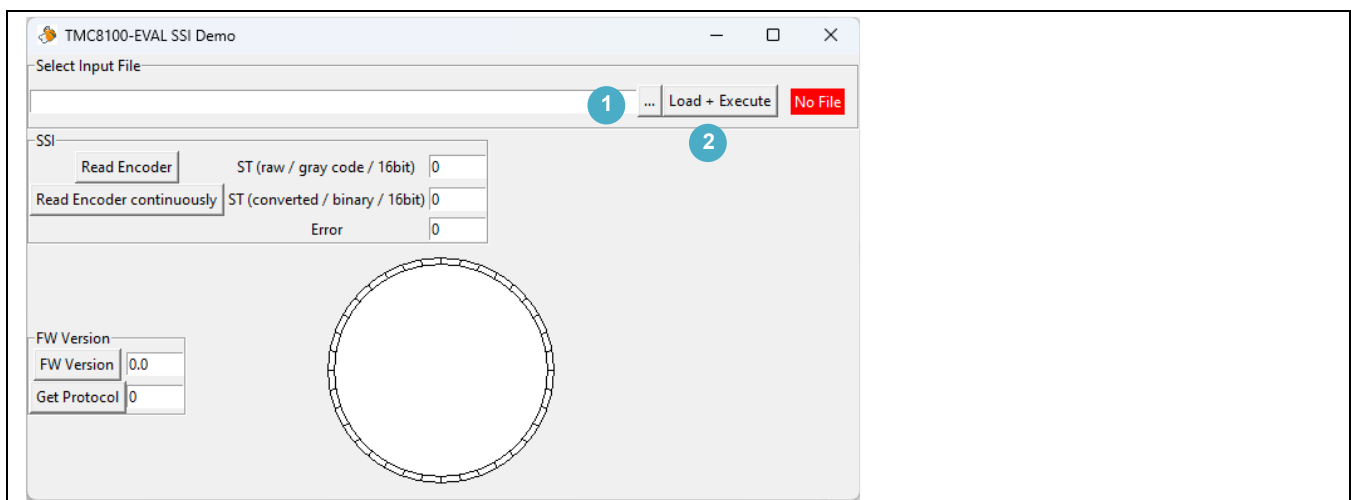
C:\Users\GEggers>cd \temp\examples\ssi

C:\temp\examples\ssi>python tmc8100-eval_SSI_encoder_demo_v10.py
Available ports:
0 - COM4 - USB Serial Device (COM4)
1 - COM3 - Intel(R) Active Management Technology - SOL (COM3)
Select port: 0
LB version string: b'0012V31'
** Reset TMC8100 **
TMC8100 bootloader version: b5 00 00 00
TMC8100 chip ID: 8 1 0 0
TMC8100 chip revision: 11 00 00 00

```

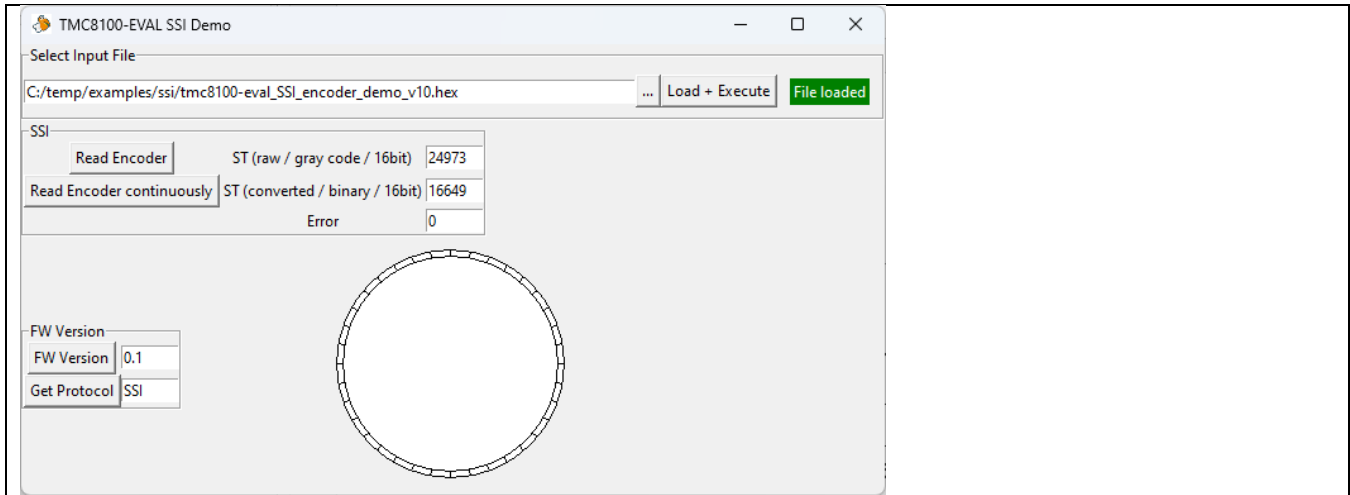
At first, the virtual COM port for the USB connection to the TMC8100-EVAL-KIT must be selected—in this example, it is “COM4.” The lines of output in the terminal window already indicate successful connection to the Landungsbruecke (LB) and detection of the TMC8100 with chip ID and revision number.

Afterwards, the GUI automatically starts in a separate window.



First, the hex file “tmc8100-eval\_ssi\_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) as next step, 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, for downloading a different program a reset or power cycle is required.



The **SSI** frame in the middle of the window offers two command buttons to initiate an encoder read-out. Pressing the **Read Encoder** button instructs the firmware on the TMC8100 to send out the respective number of clock cycles through the SSI and read back the absolute position within one rotation (shown next to the “ST” label). This is shown here in Gray code format [ST (raw / gray code / 16bit)] as received directly from the encoder and as 16-bit value after conversion [ST (converted / binary / 16bit)]. The conversion is done inside the TMC8100 firmware and the original Gray code number is transmitted and shown here just for reference.



Pressing the **Read Encoder continuously** button triggers read-out of the encoder value at a constant rate by the Python program. In addition to the digital numbers with the position values, an analog dial-wheel with a red position mark is also updated with the encoder value as indicator for the current absolute 360° angle position. To stop continuous encoder read-out, the Python program must be terminated/restarted.

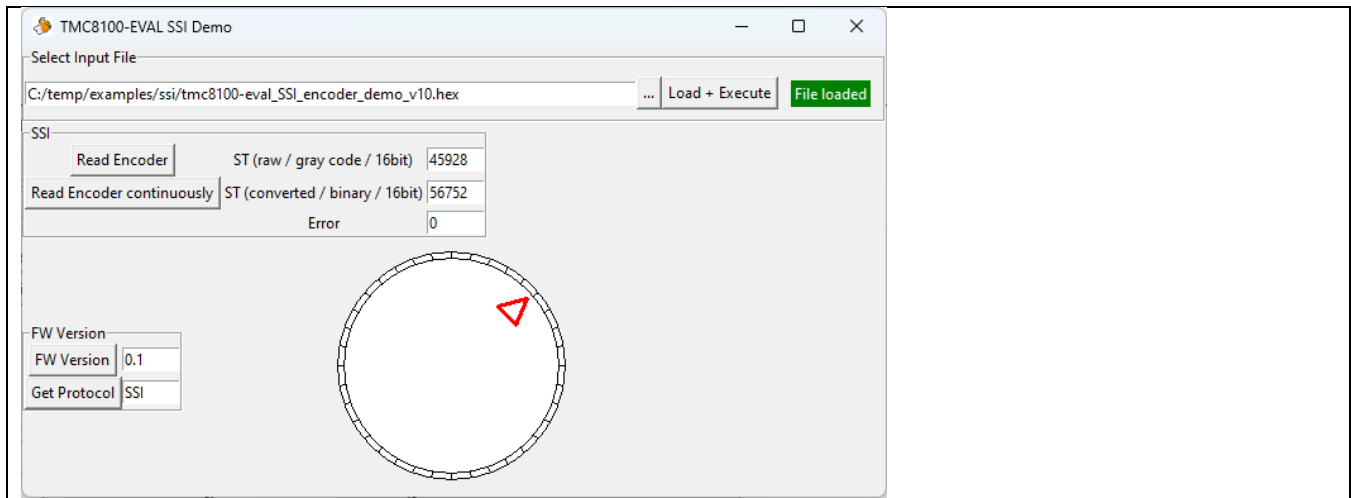


Figure 4. GUI in Python for testing the firmware implementation

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.



Output to the command line window is suppressed during continuous encoder read-out for performance reasons.

## FIRMWARE IMPLEMENTATION

The example source code “tmc8100-eval\_ssi\_encoder\_demo\_v10.asm” may be used as a starting point and modified according to application requirements. An [Assembler](#) is available for translation of the source code.

The flowchart ([Figure 5](#)) gives an overview of the example code.

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 = $53 ; "S"
PROTOCOL_2 = $53 ; "S"
PROTOCOL_1 = $49 ; "I"
PROTOCOL_0 = $20 ; " "
```

```

; system register
SYSTEM_CORE           = $0
SYSTEM_TIMER          = $1
SYSTEM_CRC            = $2
    
```

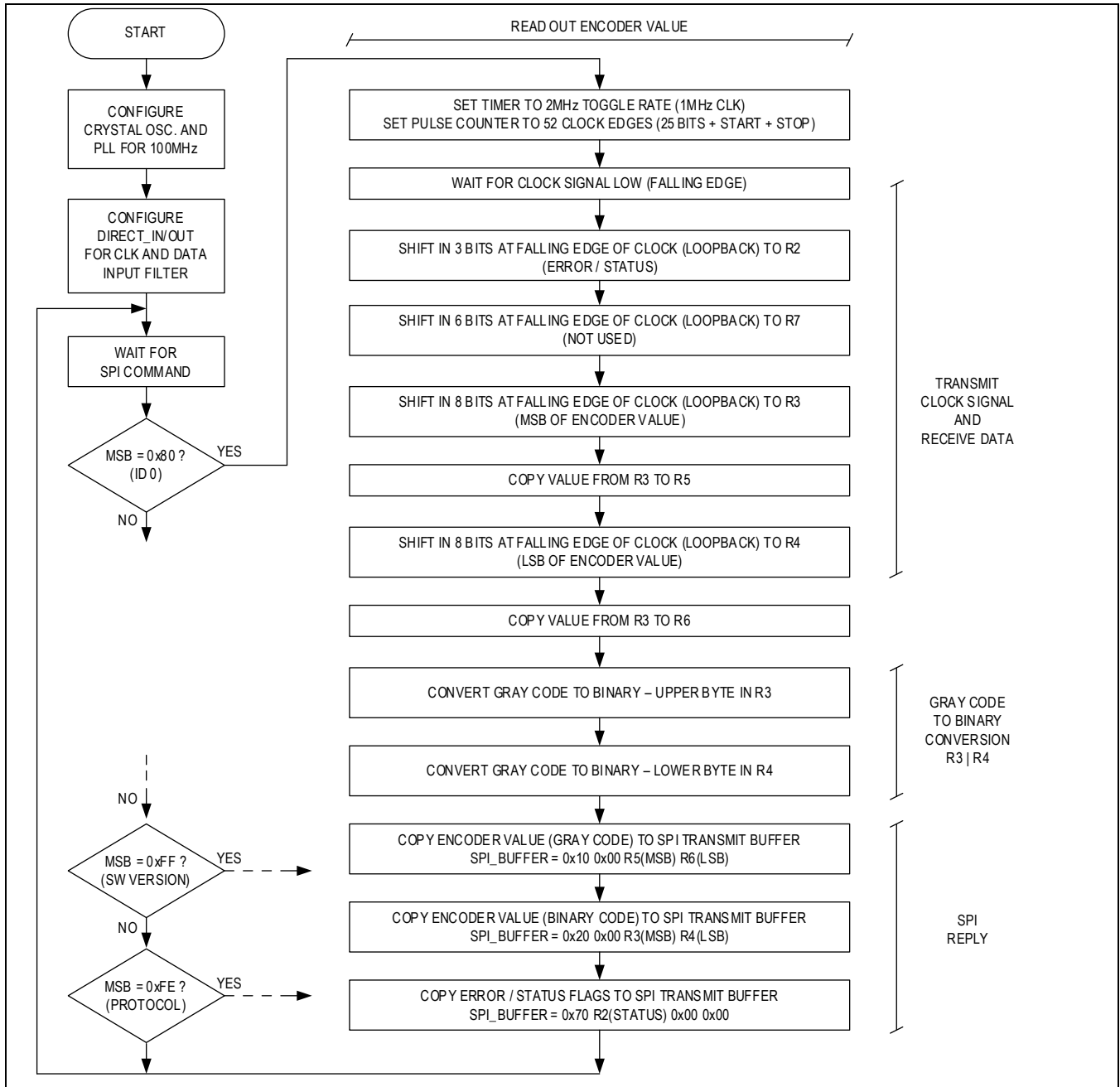


Figure 5. Firmware implementation overview

## SPI Configuration

The standard SPI signals (SPI\_CSN, SPI\_SCLK, SPI\_SDI and SPI\_SDO) available with the TMC8100 do not require any configuration. They have fixed functionality and pin assignment for the package. Nevertheless, an additional signal SPI\_DATA\_AVAILABLE is available that can be configured instead of GPIO(6) and indicates data that is written to the SPI output buffer (output high '1') by the firmware inside the TMC8100. This allows for feedback to the attached microcontroller which may then initiate an SPI datagram/transaction to fetch the reply data for a previously transmitted command from the TMC8100. This functionality is already configured by the bootloader after power-up and is here included for completeness:

```
LD GPIO0_ALT1_FUNCTION, r0
LD GPIO_OUT_ENABLE, r1
SET $4, r0, r0 ; GPIO_ALT1_FUNCTION(5 downto 4) = "01" -> connect spi_data_available to GPIO(6)
CLR $5, r0, r0
ST GPIO0_ALT1_FUNCTION, r0
SET $6, r1, r1 ; GPIO(6) / SPI_DATA_AVAILABLE -> output
ST GPIO_OUT_ENABLE, r1
```

## 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. In this example, the crystal oscillator is used with the 16MHz crystal available on the TMC8100-EVAL-KIT. The PLL output and system frequency is set to 100MHz to simplify clock calculation/divider settings afterwards. For the SSI, a crystal clock is not strictly necessary as the reference clock for communication is provided by the TMC8100 itself. Therefore, using the internal clock of the TMC8100 together with the PLL might be an alternative in such a case.

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 are 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 LDI and ST 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 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](#), both RS485 transceivers on the TMC8100-EVAL-KIT are used for communication with the encoder. The upper one in the block diagram is used for transmitting the clock signal from TMC8100 to the encoder. Transmit enable is switched on permanently for this RS485 transceiver. DIRECT\_OUT(0) is used here for clock output (alternate function of this output selected with command “ST DIRECT\_ALT\_FUNCITON, r0”) and DIRECT\_IN(0) to loop back the clock signal from the RS485 transceiver output to the TMC8100 for driver delay compensation. This is an option and not strictly necessary as the clock loop back can be also done internally.

The lower RS485 transceiver in the block diagram is used for receiving data from the encoder. The transmitter is disabled here. DIRECT\_IN(1) is used for serial data input while DIRECT\_OUT(1) and DIRECT\_OUT(3) are not necessary in this application and set to fixed levels (both low ‘0’).

As the clock line (CLK+) is at high level ‘1’ in idle state, the corresponding output DIRECT\_OUT(0) is inverted. For data and clock loopback input, a filter in hardware is enabled.

```

; configure DIRECT_OUT(0) as inverted
LDI $10, r0
ST DIRECT_POLARITY, r0
; set DIRECT_OUT(1) = 0 -> always low
SFCLR WAIT1SF_NO_WAIT, 0, 1
; set DIRECT_OUT(3) = 0 -> always low
SFCLR WAIT1SF_NO_WAIT, 0, 3
; configure DIRECT_OUT(0) as clock output of the system timer and DIRECT_OUT(3) as output
LDI $01, r0
ST DIRECT_ALT_FUNCTION, r0
; enable input filter for DIRECT_IN
LDI %0000_0001, r0
STS r0, SYSTEM_CORE, SYSTEM_CORE_INPUT_FILTER_W

```

## 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
; read encoder position data
LDI $80, r1
COMP EQ r0, r1
JC read_encoder_data
; get software version
. . . .
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 reading out the encoder data including flags and position value. If this comparison is successful, program execution jumps to address “read\_encoder\_data”. The program code at this address—described in more detail in the following images—then sends out the pre-defined number of clock cycles through DIRECT\_OUT(0) and collects the reply data from the encoder through DIRECT\_IN(1). The received data is assembled into 32-bit SPI datagrams and put into the SPI output buffer. Simultaneously, SPI\_BUFFER\_FULL/GPIO6 changes from low ‘0’ to high ‘1’ to indicate new data available for SPI transaction. 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. The last transaction for read-out may already include the next command.

The Python script available to test the encoder implementation “tmc8100-eval\_ssi\_encoder\_demo.py” with the TMC8100-EVAL-KIT offers a GUI with a **Read Encoder** push button ([Figure 4](#)). Pressing this button instructs the Landungsbruecke/LB to send out an SPI datagram 0x80 0x00 0x00 0x00 (extract from “tmc8100-eval\_ssi\_encoder\_demo.py”).

```
def spi_get_encoder_value():
# get encoder value command
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 ‘0’ to high ‘1’ 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 ST value (raw / gray code value)
value = SpiWriteCommand([0x00, 0x00, 0x00, 0x00])
print(f"ST (raw): {value[0]:02x} {value[1]:02x} {value[2]:02x} {value[3]:02x}")
. . . .
# get ST value (binary value)
value = SpiWriteCommand([0x00, 0x00, 0x00, 0x00])
print(f"ST (binary): {value[0]:02x} {value[1]:02x} {value[2]:02x} {value[3]:02x}")
. . . .
# get error flags
value = SpiWriteCommand([0x00, 0x00, 0x00, 0x00])
```

```
print(f"ERR: {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 three 32-bit datagrams in this example. SPI 32-bit datagrams are given as four 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—first-in first-out (FIFO).

SPI COMMAND (32-BIT)	SPI REPLY (32-BIT)
0x80 0x00 0x00 0x00 Read encoder error flags and encoder absolute position value within one rotation (ST)	<ol style="list-style-type: none"> <li>0x10 0x00 ST (MSB) ST (LSB) (raw position value in gray code from encoder)</li> <li>0x20 0x00 ST (MSB) ST (LSB) (converted binary position value)</li> <li>0x70 Flags 0x00 0x00</li> </ol>
0xff 0x00 0x00 0x00 get firmware version	<ol style="list-style-type: none"> <li>0xff 0x00 VERSION_MAJOR VERSION_MINOR</li> </ol>
0xfe 0x00 0x00 0x00 get encoder protocol	<ol style="list-style-type: none"> <li>0x53 (“S”) 0x53 (“S”) 0x49 (“I”) 0x20 (“ “)</li> </ol>

When the **Read Encoder continuously** button is pressed, the first command in the table 0x80 0x00 0x00 0x00 is sent repeatedly by the Python program.

## Read Encoder Data

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 “read\_encoder\_data” in program code and starts with preparing and transmitting clock pulses through DIRECT\_OUT(0) and attached RS485 transceiver toward the encoder.

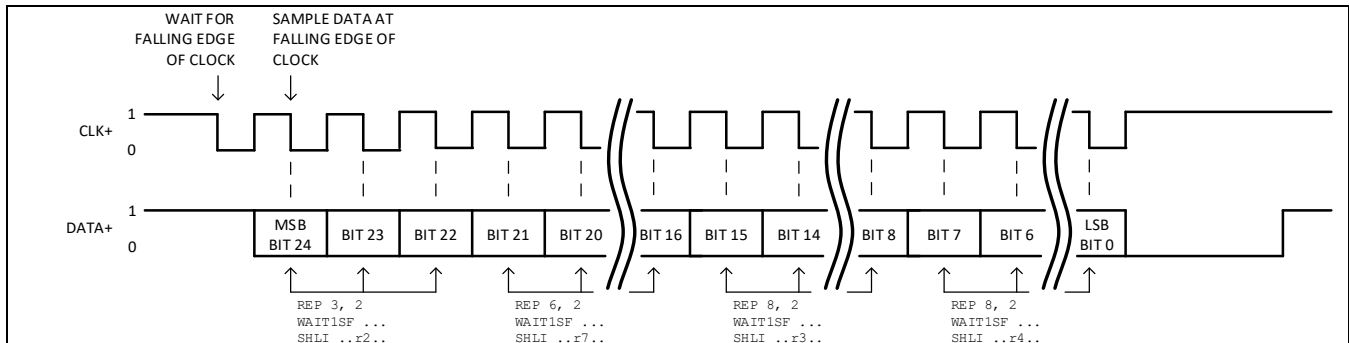
```
; set timer to 100MHz / 50 = 2MHz clock toggle rate -> 1MHz SSI clock
LDI 49, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_COUNTER_LIMIT_W
; number of clock edges: 52 (25 x 2 + falling edge at the beginning and rising edge at the end)
LDI 52, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_PULS_COUNTER_LIMIT_W
LDI 1, r0 ; enable counter
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_CTRL_W
. . .
```

The clock signal is configured for 1MHz which is the maximum for the encoder used in this example. At each overflow of the system counter, the clock output toggles. Therefore, the system counter is set to double the output clock frequency—in this case 2MHz. The number of clock cycles is equal to the number of bits shifted out from the encoder—which is 25 in this case—plus the falling edge at the beginning and a rising edge at the end. This results in 26 clock cycles altogether with 52 rising and falling edges on the clock output. As final step of the counter system, timer initialization clock generation is started.

As the next step, reply data from the encoder is captured on input DIRECT\_IN(1). As reference—also to provide some kind of delay compensation—the clock signal looped back through the RS485 transceiver and input at pin DIRECT\_IN(0) is used. Note: The internal clock may be used directly in case there is no loopback available—e.g., when using an RS422 transmitter.

The firmware waits until the clock signal CLK+ changes from high level ‘1’ to low level ‘0’ indicating start of transmission (WAIT0SF WAIT\_IN0, WAIT\_NO\_ACTION). The serial data from the encoder is shifted out with the next

rising edge of the clock signal and captured at the input DIRECT\_IN(1) by the TMC8100 firmware with the falling edge of the clock signal. The first bit captured is the MSB/Bit-24 of the 25-bit datagram (Figure 6).



**Figure 6. TMC8100 encoder clock output (CLK+)/serial data receive input (DATA+)**

```

; wait for clock signal low / start of transmission
WAIT0SF WAIT_IN0, WAIT_NO_ACTION ; wait for CLK signal low

; shift in error flags on falling edge of clock
REP 3, 2
WAIT1SF WAIT_IN0, WAIT_NO_ACTION ; wait for CLK signal high
SHLI WAIT0SF WAIT_IN0, r2, FLAG_IN1 ; shift DATA in MSB first on falling edge of clock signal

; shift in absolute multi turn counter value (not used)
REP 6, 2
WAIT1SF WAIT_IN0, WAIT_NO_ACTION ; wait for CLK signal high
SHLI WAIT0SF WAIT_IN0, r7, FLAG_IN1 ; shift DATA in MSB first

; shift in single turn counter value - MSB
REP 8, 2
WAIT1SF WAIT_IN0, WAIT_NO_ACTION ; wait for CLK signal high
SHLI WAIT0SF WAIT_IN0, r3, FLAG_IN1 ; shift DATA in MSB first
OR r3, r3, r5 ; copy original value to r5

; shift in single turn counter value - LSB
REP 8, 2
WAIT1SF WAIT_IN0, WAIT_NO_ACTION ; wait for CLK signal high
SHLI WAIT0SF WAIT_IN0, r4, FLAG_IN1 ; shift DATA in MSB first
OR r4, r4, r6 ; copy original value to r6

```

The first three bits received are shifted into register r2 at subsequent falling edges of the clock signal with the two instructions after REP 3, 2 repeated three times in a row. The first of the two instructions always waits until the clock signal is high '1' (WAIT1SF ...) and the second one shifts the serial data into register r2 with the falling edge of the clock (SHLI ..r2..)—from left to right with the MSB first.

The next three instructions starting with REP 6, 2 shift 6 bits into register r7 which are not used in this example. With other SSI encoders, they may contain the multiturn counter information.

Finally, there are two blocks with three instructions each starting with REP 8, 2 that shift the encoder counter value into register r3 (MSB) and register r4 (LSB) at the falling edge of the clock signal. This received data is also copied to registers r5 (MSB) and r6 (LSB) to preserve the original data from the encoder in gray code.



```
LDI $0, r0
ST SPI_BUFFER_2, r0
ST SPI_BUFFER_1, r3
ST SPI_BUFFER_0, r4
```

The second 32-bit SPI datagram contains the converted absolute position information in binary code for the lower 16-bit and a fixed value of 0x20 as MSB.

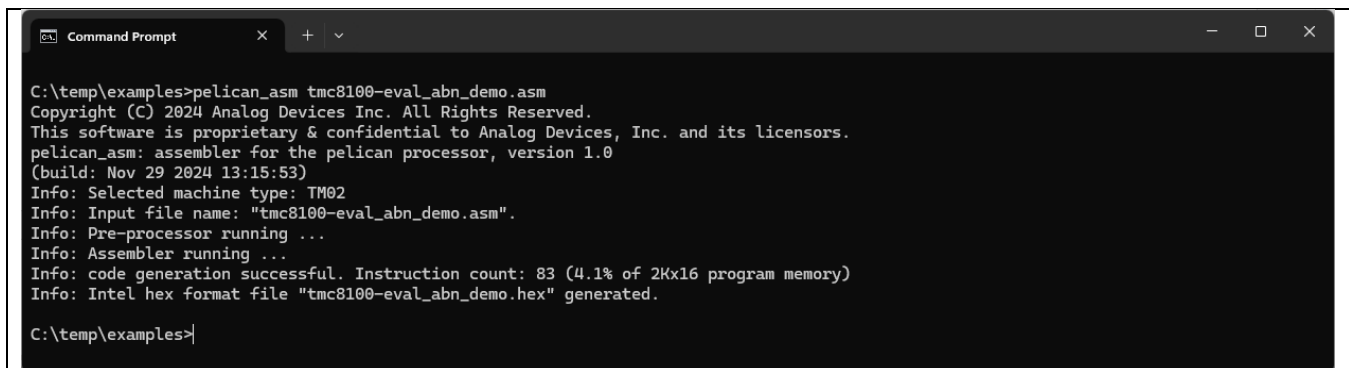
```
LDI %0111_0000, r0 ; flag value
ST SPI_BUFFER_3, r0
ST SPI_BUFFER_2, r2 ; flags
LDI $0, r0
ST SPI_BUFFER_1, r0 ; no CRC
ST SPI_BUFFER_0, r0 ; no CRC
```

The third and last 32-bit SPI datagram contains the flags received from the encoder and a fixed value of 0x70 as MSB.

## 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).



```

C:\temp\examples>pelican_asm tmc8100-eval_abn_demo.asm
Copyright (C) 2024 Analog Devices Inc. All Rights Reserved.
This software is proprietary & confidential to Analog Devices, Inc. and its licensors.
pelican_asm: assembler for the pelican processor, version 1.0
(build: Nov 29 2024 13:15:53)
Info: Selected machine type: TM02
Info: Input file name: "tmc8100-eval_abn_demo.asm".
Info: Pre-processor running ...
Info: Assembler running ...
Info: code generation successful. Instruction count: 83 (4.1% of 2Kx16 program memory)
Info: Intel hex format file "tmc8100-eval_abn_demo.hex" generated.

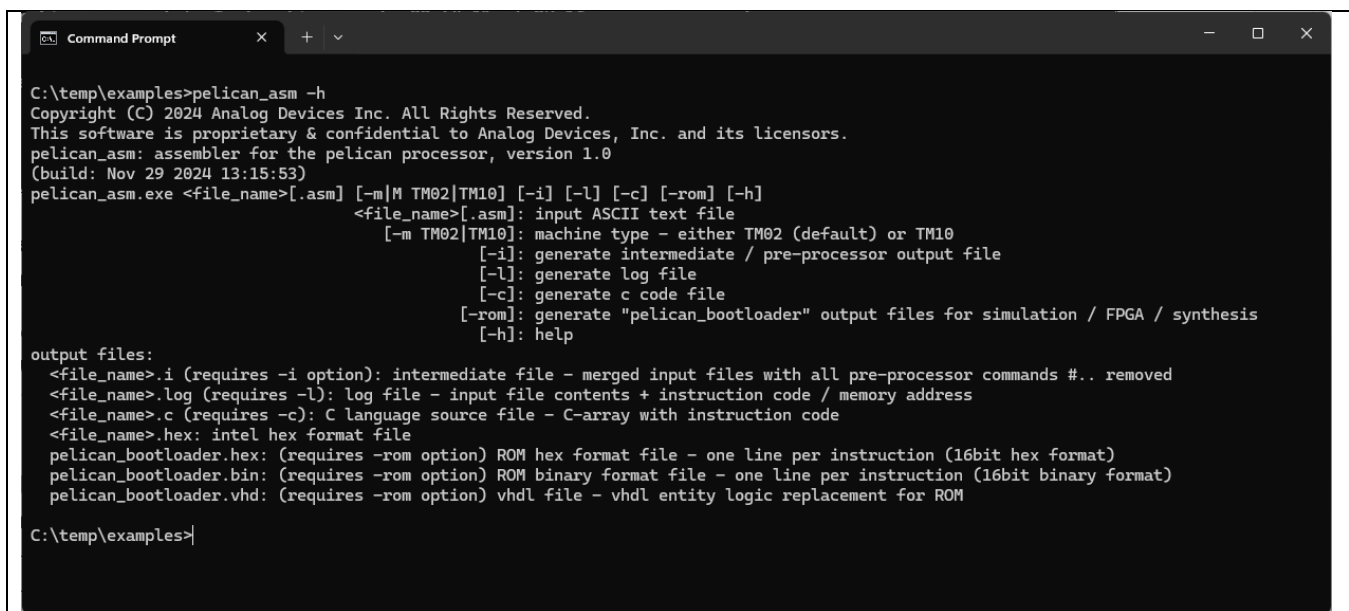
C:\temp\examples>

```

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.



```

C:\temp\examples>pelican_asm -h
Copyright (C) 2024 Analog Devices Inc. All Rights Reserved.
This software is proprietary & confidential to Analog Devices, Inc. and its licensors.
pelican_asm: assembler for the pelican processor, version 1.0
(build: Nov 29 2024 13:15:53)
pelican_asm.exe <file_name>[.asm] [-m|M TM02|TM10] [-i] [-l] [-c] [-rom] [-h]
<file_name>[.asm]: input ASCII text file
[-m TM02|TM10]: machine type - either TM02 (default) or TM10
[-i]: generate intermediate / pre-processor output file
[-l]: generate log file
[-c]: generate c code file
[-rom]: generate "pelican_bootloader" output files for simulation / FPGA / synthesis
[-h]: help

output files:
<file_name>.i (requires -i option): intermediate file - merged input files with all pre-processor commands #.. removed
<file_name>.log (requires -l): log file - input file contents + instruction code / memory address
<file_name>.c (requires -c): C language source file - C-array with instruction code
<file_name>.hex: intel hex format file
pelican_bootloader.hex: (requires -rom option) ROM hex format file - one line per instruction (16bit hex format)
pelican_bootloader.bin: (requires -rom option) ROM binary format file - one line per instruction (16bit binary format)
pelican_bootloader.vhd: (requires -rom option) vhd file - vhd entity logic replacement for ROM

C:\temp\examples>

```

- 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.

ALL INFORMATION CONTAINED HEREIN IS PROVIDED “AS IS” WITHOUT REPRESENTATION OR WARRANTY. NO RESPONSIBILITY IS ASSUMED BY ANALOG DEVICES FOR ITS USE, NOR FOR ANY INFRINGEMENTS OF PATENTS OR OTHER RIGHTS OF THIRD PARTIES THAT MAY RESULT FROM ITS USE. SPECIFICATIONS ARE SUBJECT TO CHANGE WITHOUT NOTICE. NO LICENCE, EITHER EXPRESSED OR IMPLIED, IS GRANTED UNDER ANY ADI PATENT RIGHT, COPYRIGHT, MASK WORK RIGHT, OR ANY OTHER ADI INTELLECTUAL PROPERTY RIGHT RELATING TO ANY COMBINATION, MACHINE, OR PROCESS, IN WHICH ADI PRODUCTS OR SERVICES ARE USED. TRADEMARKS AND REGISTERED TRADEMARKS ARE THE PROPERTY OF THEIR RESPECTIVE OWNERS. ALL ANALOG DEVICES PRODUCTS CONTAINED HEREIN ARE SUBJECT TO RELEASE AND AVAILABILITY.