TMC8100

Universal Encoder Bus Controller

General Description

The TMC8100 is a dedicated serial protocol converter IC, especially for absolute encoder bus protocols. It operates as a bus controller for these protocols and as a peripheral with either a serial peripheral interface (SPI) or universal asynchronous receiver-transmitter (UART) interface connection to the attached microcontroller/motion controller delivering the extracted and adjusted encoder position information.

It integrates a programmable high performance serial communication engine for synchronous and asynchronous data up-to 16Mb/s. In addition to a clock generator, several counter/timer units, a programmable CRC generator and direct I/Os for connecting bus transceivers, standard SPI, 2x UART, and I^2C interfaces are available.

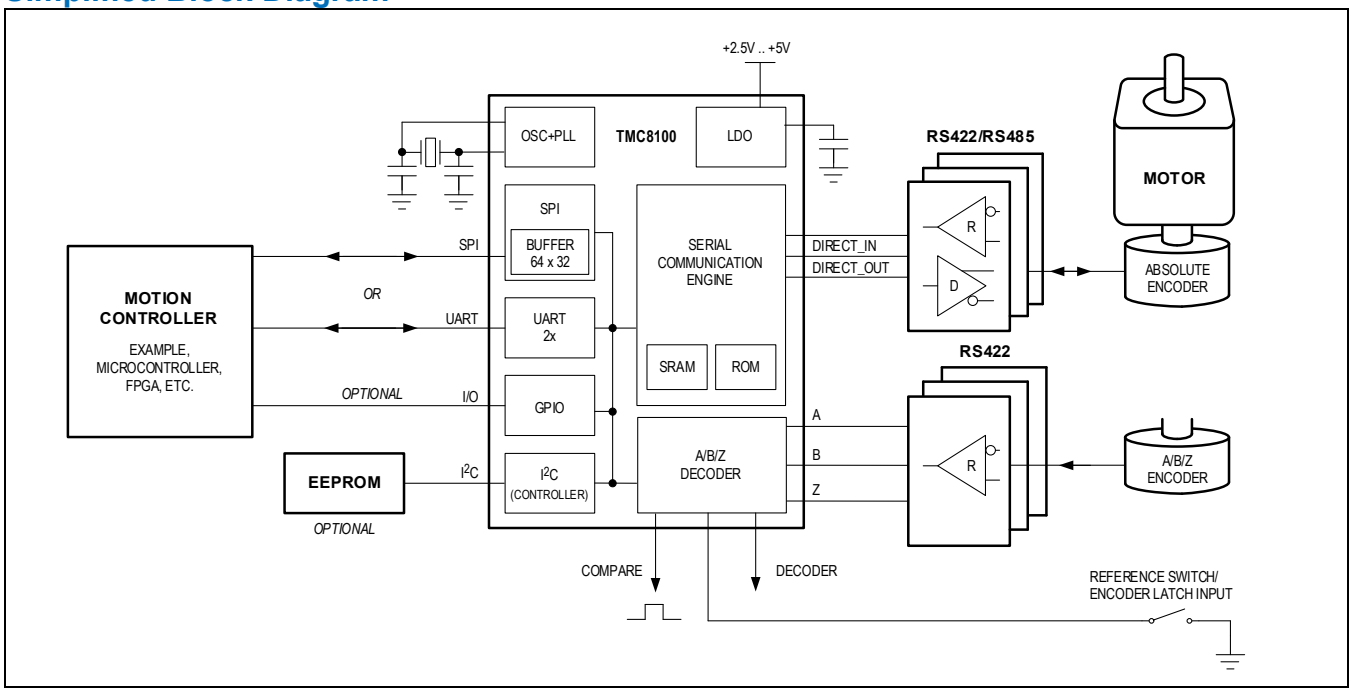
Benefits and Features

- Synchronous serial bus protocols supported, example, SSI, SPI, BiSS C, EnDat 2.x
- Asynchronous serial bus protocols supported, example, Nikon A-format®
- Support for incremental A/B/Z encoder interface
- High speed 25MHz SPI system interface for configuration, control, and position
- High speed 2x UART 16Mbit/s system interface for configuration, control, and position
- Crystal oscillator or external clock with PLL
- Up to 128MHz internal system clock
- 2.5V to 5V single supply
- -40°C to +125°C operating temperature range
- TQFN24, 4mm x 4mm

Applications

- Industrial Manufacturing
- Robots/CoBots
- Automated Guided Vehicle (AGV)

Simplified Block Diagram



Ordering Information ***appears at end of data sheet.***

19-101867B; Rev 0; 4/24

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Absolute Maximum Ratings

V_{CCIO} ..... -0.3V to +6V

RESETN to GND ..... -0.3V to V_{CCIO} to + 0.3V

SPI to GND ..... -0.3V to V_{CCIO} to + 0.3V

GPIO to GND ..... -0.3V to V_{CCIO} to + 0.3V

DIRECT_IN/OUT to GND ..... -0.3V to V_{CCIO} to + 0.3V

Continuous Power Dissipation (Multilayer Board) (T_A = +70°C, derate 25.60 mW/°C above +70°C.) 2051.30mW

Continuous Power Dissipation (Single Layer Board) (T_A = +70°C, derate 16.9 mW/°C above +70°C.) 1355.90mW

Operating Temperature Range ..... -40°C to +125°C

Junction Temperature ..... +150°C

Soldering Temperature (reflow) ..... +260°C

Storage Temperature Range ..... -55°C to +150°C

Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated in the operational sections of the specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Package Information

TQFN24 4x4	
Package Code	T2444+3C+1
Outline Number	*21-0139*
Land Pattern Number	*90-0022*
Thermal Resistance, Single Layer Board:	
Junction-to-Ambient (θ_{JA})	68°C/W
Junction-to-Case Thermal Resistance (θ_{JC})	11°C/W
Thermal Resistance, Four Layer Board:	
Junction-to-Ambient (θ_{JA})	60°C/W
Junction-to-Case Thermal Resistance (θ_{JC})	11°C/W

For the latest package outline information and land patterns (footprints), go to *www.maximintegrated.com/packages*. Note that a "+", "#", or "-" in the package code indicates RoHS status only. Package drawings may show a different suffix character, but the drawing pertains to the package regardless of RoHS status.

Package thermal resistances were obtained using the method described in JEDEC specification JESD51-7, using a four-layer board. For detailed information on package thermal considerations, refer to *www.maximintegrated.com/ thermal-tutorial*.

Electrical Characteristics

(V_{CCIO} = +2.25V to +5.5V, TA = -40ºC to +125ºC, unless otherwise noted., Typical values are at V_{CCIO} = +3.3V, and TA = +25ºC, unless otherwise noted. *Note 1*)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
DC ELECTRICAL CHARACTERISTICS/Operating Voltage Range						
V_{CCIO} Supply Voltage Range			2.25		5.5	V
V_{CCIO} UVLO Threshold	VCCIO_UV	Rising	1.6	1.78	2	V
		Falling	1.4	1.57	1.735	
DC ELECTRICAL CHARACTERISTICS/Current Consumption						
Total V_{CCIO} Quiescent Current Consumption	IQVCCIO	V_{CCIO} = +3.3V, RESETN low		100		uA
Total V_{CCIO} Current Consumption	IVCCIO	V_{CCIO} = +3.3V, EXT_CLK = 1MHz, PLL Output = 128MHz		22		mA
DC ELECTRICAL CHARACTERISTICS/Data In Mode						
Resistive Pull-up (RESETN, SPI, GPIO, DIRECT_IN)	RPU	Internal	60	100	140	kΩ
Resistive Pull-down (SPI, GPIO, DIRECT_IN)	RPD	Internal	60	100	140	kΩ
Rising Threshold (RESETN, SPI, GPIO, DIRECT_IN)	DIH				70	%V_{CCIO}
Falling Threshold (RESETN, SPI, GPIO, DIRECT_IN)	DIL		30			%V_{CCIO}
Hysteresis (RESETN, SPI, GPIO, DIRECT_IN)	DI_HYS			14		%V_{CCIO}
Logic Input Leakage Current (SPI, GPIO, DIRECT_IN)	I_{LEAK}	PU/PD disabled	-1		+1	µA
DC ELECTRICAL CHARACTERISTICS/Data Out Mode						
Output Low Voltage (SPI, GPIO, DIRECT_OUT)	DOL	I = 5mA *Note 3*			0.4	V
Output High Voltage (SPI, GPIO, DIRECT_OUT)	DOH	I = -5mA *Note 3*	V_{CCIO} - 0.4			V
DC ELECTRICAL CHARACTERISTICS/Linear Regulator						
1V8 LDO Output Voltage	V1V8	C_{LOAD} = 2.2µF, min. V_{CCIO} = 2.25V		1.90		V
1V8 LDO Current Limit	I1V8_SH	1V8 shorted to GND	75	126	275	mA
AC ELECTRICAL CHARACTERISTICS/Data In/Out Mode						
Propagation Delay Mismatch (DIRECT_IN/OUT)	tDIMM				7	ns
Maximum Frequency (DIRECT_OUT)	fMAX_DOUT	V_{CCIO} = +3V			40	MHz
AC ELECTRICAL CHARACTERISTICS/Clock						

(V_{CCIO} = +2.25V to +5.5V, TA = -40ºC to +125ºC, unless otherwise noted., Typical values are at V_{CCIO} = +3.3V, and TA = +25ºC, unless otherwise noted. *Note 1*)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
Internal Oscillator Frequency	ICLK			15		MHz
External Oscillator Frequency Range	ECLK		1		32	MHz
Internal PLL Output Frequency	PLL_CLK			75, 100, 128		MHz
AC ELECTRICAL CHARACTERISTICS/Quartz Oscillator						
Oscillator Frequency	f_{XTAL}			8, 16, 24, 25, 32		MHz
Recommended Load Capacitance of the Crystal	CL				9	pF
Crystal Driving current *Note 2*	I_{XTAL}	V_{CCIO} = +3V .. +5.5V, with CL = 9 pF, f_{XTAL} = 32MHz		1.2		mA
Oscillator Transconductance *Note 2*	g_m	V_{CCIO} = +3V .. +5.5V, with CL = 9 pF		1.8		mA/V
Start-up Time	t_{SU}	V_{CCIO} = +3V .. +5.5V, with CL = 9 pF		2.5		ms
AC ELECTRICAL CHARACTERISTICS/SPI *Figure 1*						
SPI Clock Frequency	fSCLK	V_{CCIO} = +3V .. +5.5V			25	MHz
		V_{CCIO} = +2.25V .. +3V			15	
SCLK Clock Period	tCH+CL	V_{CCIO} = +3V .. +5.5V	40			ns
		V_{CCIO} = +2.25 .. +3V	66			
SCLK Pulse Width High	tCH	V_{CCIO} = +2.25 .. +5.5V	12			ns
SCLK Pulse Width Low	tCL	V_{CCIO} = +2.25 .. +5.5V	12			ns
CSN Fall to SDO Delay	tCSDO	V_{CCIO} = +3V .. +5.5V			25	ns
		V_{CCIO} = +2.25V .. +3V			45	
CSN High Pulse Duration *Note 2*	tCSNPW	V_{CCIO} = +2.25V .. +5.5V	4x PLL_CLK			
CSN Hold Time	tCSH	V_{CCIO} = +2.25V .. +5.5V	6			ns
SDI Setup Time	tDS	V_{CCIO} = +2.25V .. +5.5V	4			ns
SDI Hold Time	tDH	V_{CCIO} = +2.25V .. +5.5V	4			ns
SDO Output Data Propagation Delay	tDO	V_{CCIO} = +3V .. +5.5V, CL = 30pF	6		34	ns
		V_{CCIO} = +2.25V .. +3V, CL = 30pF	6		60	
ESD AND EMC TOLERANCE						
ESD Protection (All Pins)		Human Body Model		±2		kV

Note 1: All devices are 100% production tested at TA = +25°C. Specifications over temperature are guaranteed by design and characterization.

Note 2: Guaranteed by design

Note 3: All currents into the device are positive. All currents out of the device are negative.

Timing Diagrams

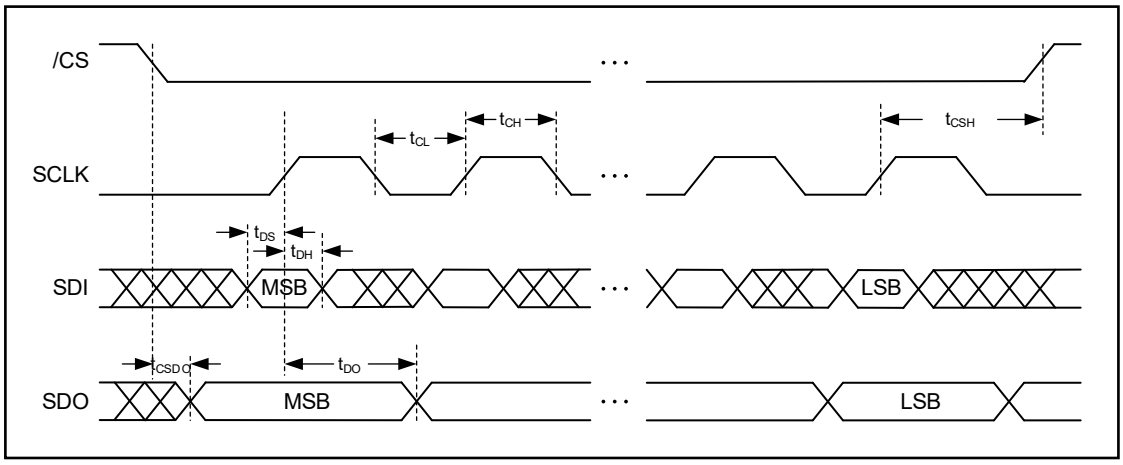


Figure 1. SPI Timing Diagram

Pin Configurations

TQFN

Package is TQFN24 4mm x 4mm with 0.5mm pitch. (T2444+3C).

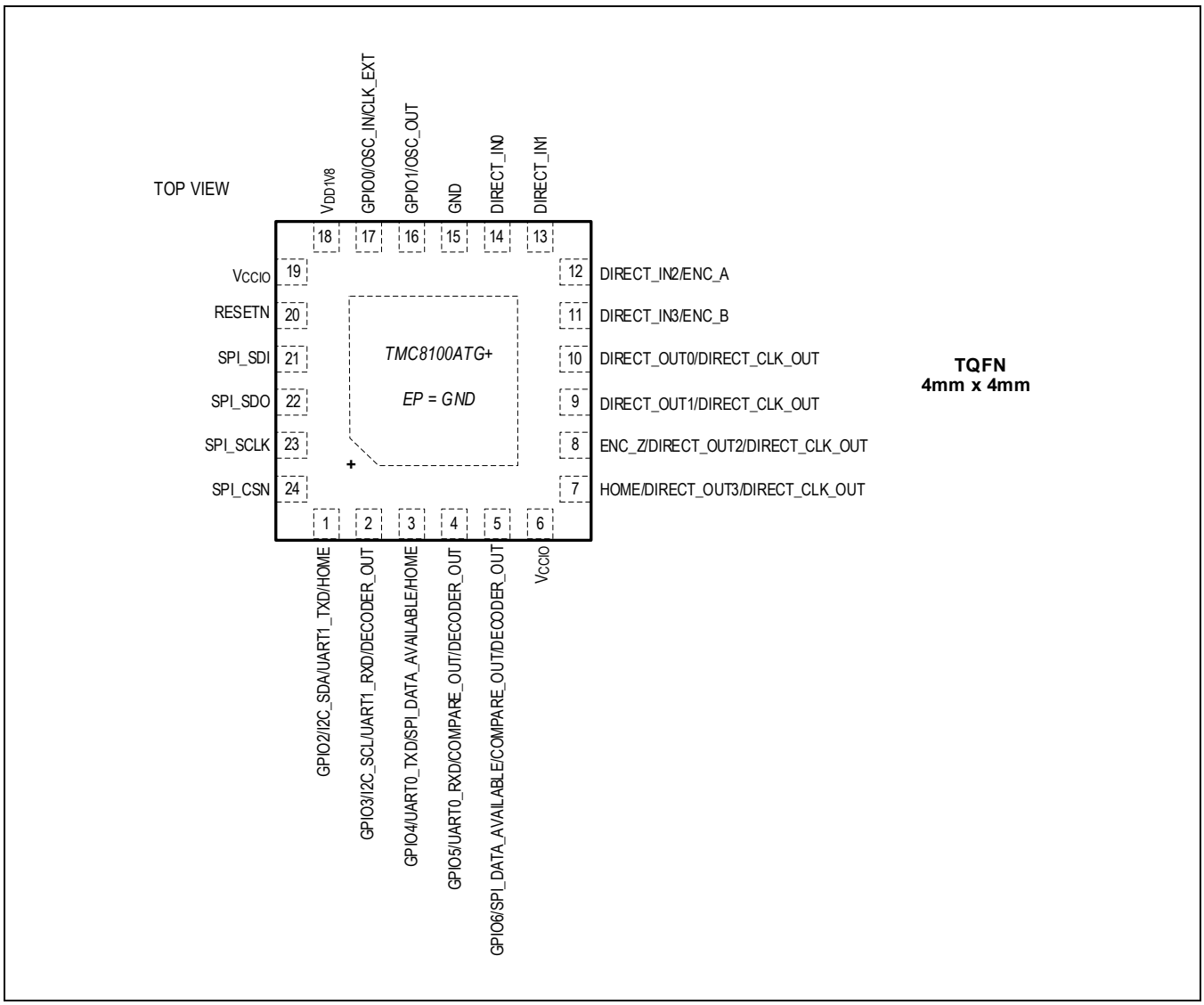


Figure 2. TMC8100 Pin Assignment

Pin Descriptions

PIN	NAME	FUNCTION	REF SUPPLY	Type
PMIO				
19	V_{CCIO}	Supply input for internal LDO and I/O pins. Connect pins 19 and 6 externally.	VCCIO	Power
18	V_{DD1V8}	1V8 output of internal linear regulator. Connect 2.2µF low ESR ceramic capacitor to GND externally.	VCCIO	Power
6	V_{CCIO}	Supply Input for I/O pins. Connect pins 19 and 6 externally.	VCCIO	Power
15	GND	Connect to GND.		
EP		Exposed pad – connect to GND.		Power
SYSTEM				
20	RESETN	Active-low, external reset input. The device remains in reset while this pin is in its active state. Internal pull-up resistor.	VCCIO	DIpud
DIRECTIO				
14	DIRECT_IN0	Serial engine direct input 0 (DIRECT_IN0) with programmable internal pull-up or pull-down resistor - power-up: pull-up resistor enabled	VCCIO	DIpud
13	DIRECT_IN1	Serial engine direct input 1 (DIRECT_IN1) with programmable internal pull-up or pull-down resistor - power-up: pull-up resistor enabled.	VCCIO	DIpud
12	DIRECT_IN2/ENC_A	Serial engine direct input 2 (DIRECT_IN2) and A/B/Z encoder interface channel A input (ENC_A) with programmable internal pull-up or pull-down resistor - power-up: pull-up resistor enabled.	VCCIO	DIpud
11	DIRECT_IN3/ENC_B	Serial engine direct input 3 (DIRECT_IN3) and A/B/Z encoder interface channel B input (ENC_B) with programmable internal pull-up or pull-down resistor - power-up: pull-up resistor enabled.	VCCIO	DIpud
10	DIRECT_OUT0/DIRECT_CLK_OUT	Serial engine direct output 0 (DIRECT_OUT0) or serial engine direct clock output (DIRECT_CLK_OUT) - power-up: DIRECT_OUT0 output selected.	VCCIO	DO
9	DIRECT_OUT1	Protocol engine direct output 1 (DIRECT_OUT1) or protocol engine direct clock output (DIRECT_CLK_OUT) - power-up: DIRECT_OUT1 output selected.	VCCIO	DO
8	ENC_Z/DIRECT_OUT2	A/B/Z encoder interface channel Z input (ENC_Z) with programmable internal pull-up or pull-down resistor or protocol engine direct output 2 (DIRECT_OUT2) or protocol engine direct clock output (DIRECT_CLK_OUT) - power-up: ENC_Z input with pull-up resistor enabled.	VCCIO	DIOpud
7	HOME/DIRECT_OUT3	A/B/Z encoder interface home switch input (HOME) with programmable internal pull-up or pull-down resistor or protocol engine direct output 3 (DIRECT_OUT3) or protocol engine direct clock output (DIRECT_CLK_OUT) - power-up: HOME input with pull-up resistor enabled.	VCCIO	DIOpud
GPIO				
21	SPI_SDI	SPI serial data input (SPI_SDI) with programmable internal pull-up or pull-down resistor - power-up: pull-up resistor enabled.	VCCIO	DIpud
22	SPI_SDO	SPI serial data output (SPI_SDO) with tristate and programmable internal pull-up or pull-down resistor - power-up: pull-up resistor enabled.	VCCIO	DOpud
23	SPI_SCLK	SPI clock input (SPI_SCLK) with programmable internal pull-up or pull-down resistor - power-up: pull-up resistor enabled.	VCCIO	DIpud
24	SPI_CSN	SPI chip select input (SPI_CSN) with programmable internal pull-up or pull-down resistor - power-up: pull-up resistor enabled.	VCCIO	DIOpud
17	GPIO0/OSC_IN/CLK_EXT	General purpose digital input or output 0 (GPIO0) with programmable internal pull-up or pull-down resistor or external clock signal in (CLK_EXT) or crystal oscillator input (OSC_IN) - power-up: GPIO0 configured as input with pull-up resistor enabled.	VCCIO	ADIOpud

16	GPIO1/OSC_OUT	General purpose digital input or output 1 (GPIO1) with programmable internal pull-up or pull-down resistor or crystal oscillator output (OSC_OUT) - power-up: GPIO1 configured as input with pull-up resistor enabled.	VCCIO	ADIOpud
1	GPIO2/I2C_SDA/UART1_TXD/HOME	General purpose digital input or output 2 (GPIO2) with programmable internal pull-up or pull-down resistor or I^2C serial data input/output (I2C_SDA) or UART1 transmit data output (UART1_TXD) or A/B/Z encoder interface home switch input (HOME) - power-up: GPIO2 configured as input with pull-up resistor enabled.	VCCIO	DIOpud
2	GPIO3/I2C_SCL/UART1_RXD/DECODER_OUT	General purpose digital input or output 3 (GPIO3) with programmable internal pull-up or pull-down resistor or I^2C clock output (I2C_SCL) or UART1 receive data input (UART1_RXD) or A/B/Z encoder interface decoder output signal (DECODER_OUT) - power-up: GPIO3 configured as input with pull-up resistor enabled.	VCCIO	DIOpud
3	GPIO4/UART0_TXD/SPI_DATA_AVAILABLE/HOME	General purpose digital input or output (GPIO4) with programmable internal pull-up or pull-down resistor or UART0 transmit data output (UART0_TXD) or SPI transmit data available signal output (SPI_DATA_AVAILABLE) or A/B/Z encoder interface home switch input (HOME) - power-up: GPIO4 configured as input with pull-up resistor enabled.	VCCIO	DIOpud
4	GPIO5/UART0_RXD/COMPARE_OUT/DECODER_OUT	General purpose digital input or output (GPIO5) with programmable internal pull-up or pull-down resistor or UART0 receive data input (UART0_RXD) or A/B/Z encoder interface position compare output (COMPARE_OUT) or A/B/Z encoder interface decoder output signal (DECODER_OUT) - power-up: GPIO5 configured as input with pull-up resistor enabled.	VCCIO	DIOpud
5	GPIO6/SPI_DATA_AVAILABLE/COMPARE_OUT/DECODER_OUT	General purpose digital input or output (GPIO6) with programmable internal pull-up or pull-down resistor or SPI transmit data available (SPI_DATA_AVAILABLE) or A/B/Z encoder interface position compare output (COMPARE_OUT) or A/B/Z encoder interface decoder output signal (DECODER_OUT) - power-up: GPIO6 configured as input with pull-up resistor enabled.	VCCIO	DIOpud

Functional Diagrams

Functional Block Diagram

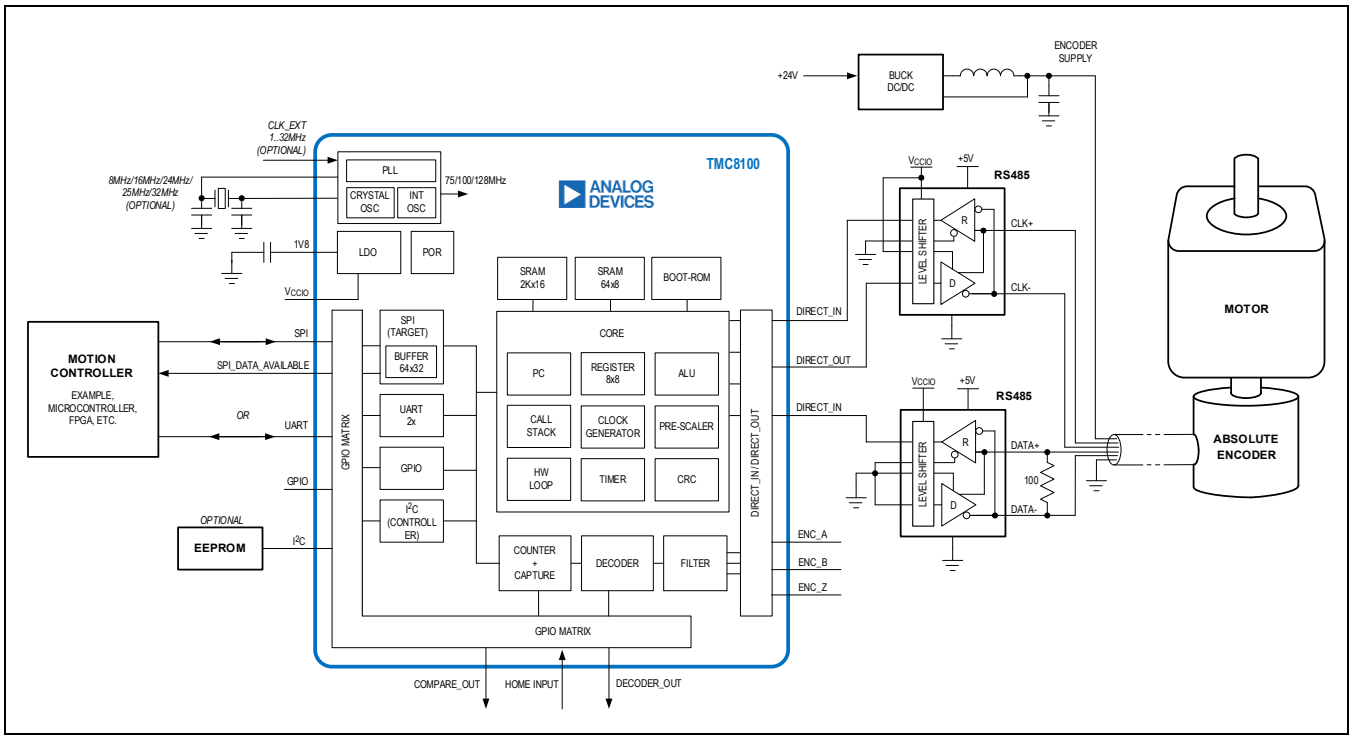


Figure 3. Block Diagram

Detailed Description

The TMC8100 is a programmable serial bus protocol converter IC targeting different absolute encoder bus protocols. It operates as a bus controller for these encoders and as a peripheral with either SPI or UART interface for the attached microcontroller or motion controller delivering the extracted and adjusted encoder position information.

The TMC8100 also supports encoders with standard incremental A/B/Z outputs. The 32-bit encoder position counter includes a capture/compare unit for generating synchronization signals and capturing the encoder counter value on external latch signals.

The TMC8100 offloads a general-purpose microcontroller or motion controller from this encoder data signal conversion task. In contrast to fully hardware-based solutions, it offers a high degree of flexibility for current protocol implementations, customization, and future protocol extensions.

For initial setup after power-up, a program supporting the specific bus protocol has to be loaded into the TMC8100 through SPI or UART with the help of the integrated bootloader program. There is also the option to add an external I^2C EEPROM for initial bootstrap supporting standalone operation.

System Architecture

The TMC8100 contains a programmable serial communication engine. The architecture and command set are optimized to convert synchronous and asynchronous serial data into parallel and vice versa. All instructions are 16-bit wide and execute within one clock cycle. The general-purpose register set contains eight registers with 8-bit each.

The processor core directly supports four digital inputs (DIRECT_IN) and four digital outputs (DIRECT_OUT) for serial data input and output. It offers separate program memory and data memory bus interfaces (Harvard architecture).

The program memory bus is connected to an on-chip static random access memory (SRAM) and a bootloader read-only memory (ROM). Several serial communication peripheral interfaces (SPI, UART, I^2C), the incremental A/B/Z encoder interface, and a small data SRAM (64x8) are connected to the data memory interface.

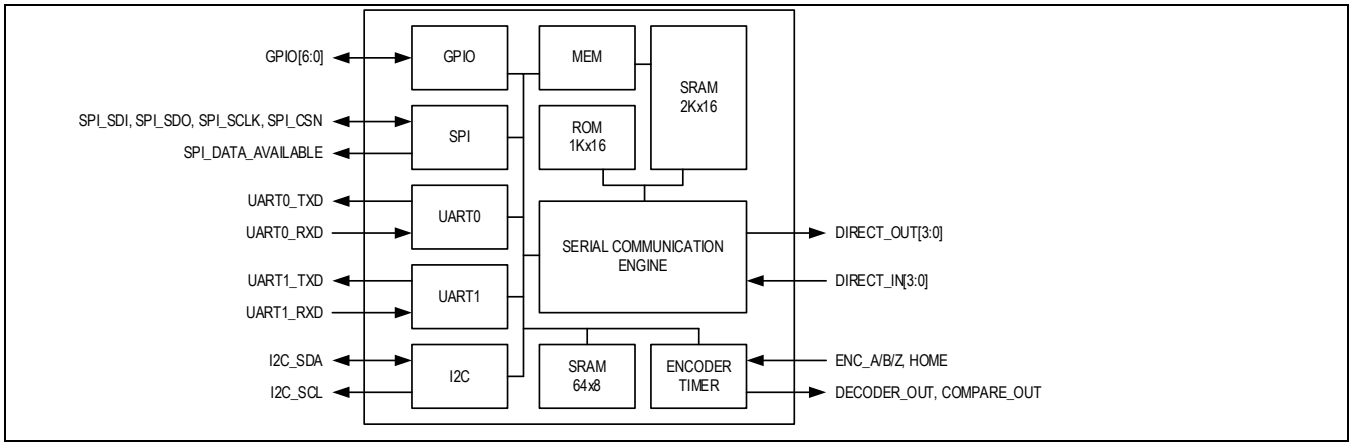


Figure 4. Block Diagram

Program Memory Bus

An embedded 2Kx16 SRAM memory is available for storing program code along with an embedded 1Kx16 ROM with bootloader to support the initial bootstrap of the application program after power-up. Both are connected to the program memory bus. The program memory bus is 16-bit wide, supporting one instruction word fetch per clock cycle. Program memory access is pipelined with two pipeline stages. Therefore, any program branch usually takes two clock cycles until the next instruction from the branch target is ready for decode and execution.

The program memory space is organized in banks with the bootloader ROM placed in the bank that is active immediately after power-up. After loading the application program into the program SRAM, the active bank is switched to the SRAM under bootloader control to start program execution from the SRAM.

Address	Memory	Description
0x0000 to 0x03FF, bank 0	1Kx16 ROM with bootloader	
0x0000 to 0x07FF, bank 1	2Kx16 program memory (SRAM)	

ROM Bootloader

After power-up, the bootloader in the ROM is executed first. The bootloader program takes care of the basic initialization of the system and sets the PLL output frequency to 75MHz with the internal oscillator as clock source. As soon as the interfaces are initialized, pin GPIO6 is configured as output and pulled low as the system is ready now for communication. Three different bootstrap modes are supported: remote through SPI or UART0 from an external microcontroller or standalone with an EEPROM (at least 4KB, example, 24LC32/24LC64) connected to the I^2C interface. During bootstrap, an application program supporting the desired encoder functionality/communication protocol must be loaded into the internal program memory (SRAM). As soon as this is successful, memory banks are switched and program execution starts from the SRAM at address $0000.

The bootstrap mode is detected automatically. In case of SPI and UART0, the receive interface is listening for incoming commands and the commands are executed accordingly. For standalone mode, a read attempt through I^2C for an external EEPROM is generated. If this is successful (ACK received), a sequential read to the first two bytes (first instruction word) starting at address $0000 (two address bytes are transmitted) follow. In case there is an acknowledge (EEPROM available) and both bytes are not $ff (EEPROM empty/erased check), 4KB of the EEPROM contents is copied to the internal SRAM automatically.

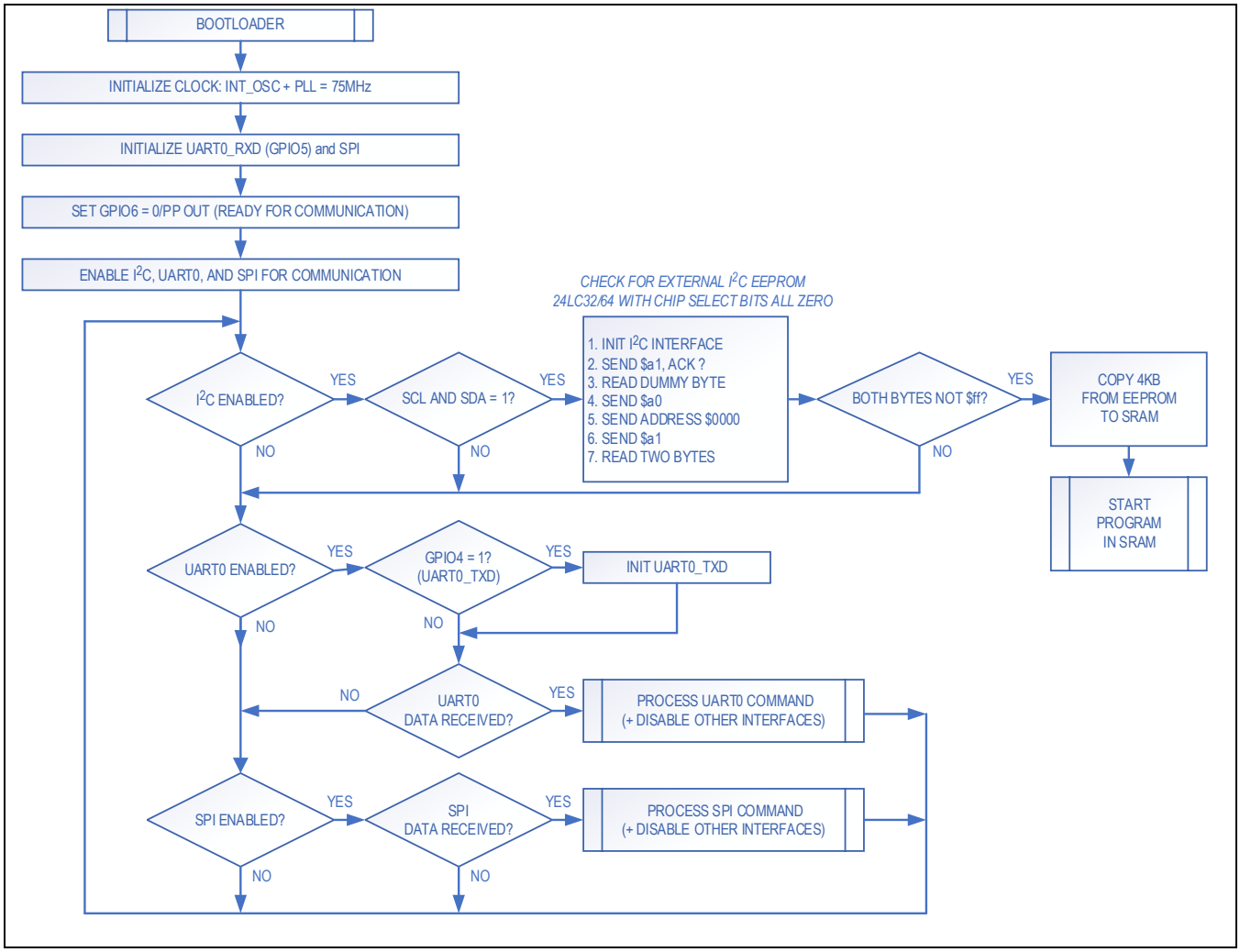


Figure 5. ROM Bootloader

UART0 Bootstrap Protocol

For serial communication through UART0, autobaud is enabled and transmission uses 1 start bit, 8 data bits, and one stop bit (8n1). The bootstrap protocol supports reading and writing to the internal program memory (SRAM) and reading and programming an external EEPROM connected through I^2C. A valid command received through UART0_RXD (GPIO5) from TMC8100 is usually followed by a reply sent out through UART0_TXD (GPIO4).

Table 1. UART0 Bootloader Commands

COMMAND	REPLY	DESCRIPTION
0x55, 0x00	0xb5	Get bootloader version.
0x55, 0x01, <ADDR_L>, <ADDR_H>	No reply	Write program memory address with lower 8-bit of address first and then the upper bits (address of 16-bit instruction word).
0x55, 0x02, <DATA_L>, <DATA_H>	No reply	Write program memory data with lower 8-bit of the instruction word first and then the upper 8-bit (instruction words are always 16-bit). The address counter for read/write access to the program memory is incremented afterwards, automatically.
0x55, 0x03	<DATA_L>, <DATA_H>	Read program memory data. Reply provides lower 8-bit of the instruction word first and then the upper 8-bit.
0x55, 0x04, <BAUD_L>, <BAUD_H>	0x09	Enable I^2C interface and set I^2C baud rate. The command includes the lower and upper byte of the I^2C baud-rate divider.
0x55, 0x05, <ADDR_L>, <ADDR_H>, <DATA_L>, <DATA_H>	0x09	Write EEPROM - lower 8-bit of the address and upper bits must be provided same as lower 8-bit of the instruction word and then the other 8-bit. As the write access to the external EEPROM takes some time, there is a reply after the write access is finished.
0x55, 0x06, <ADDR_L>, <ADDR_H>	<DATA_L>, <DATA_H>	Read EEPROM – lower 8-bit and upper bits of the address must be provided. Reply contains the 8-bit data at the specified EEPROM address and the 8-bit at the subsequent address (one 16-bit instruction word).
0x55, 0x07	No reply	Start program – stop bootloader program execution and start program execution from SRAM/address $0000.
0x55, 0x08	$38, $31, $30, $30	Get Chip ID – “8100” as ASCII characters.
0x55, 0x09	$11	Get Chip Revision.

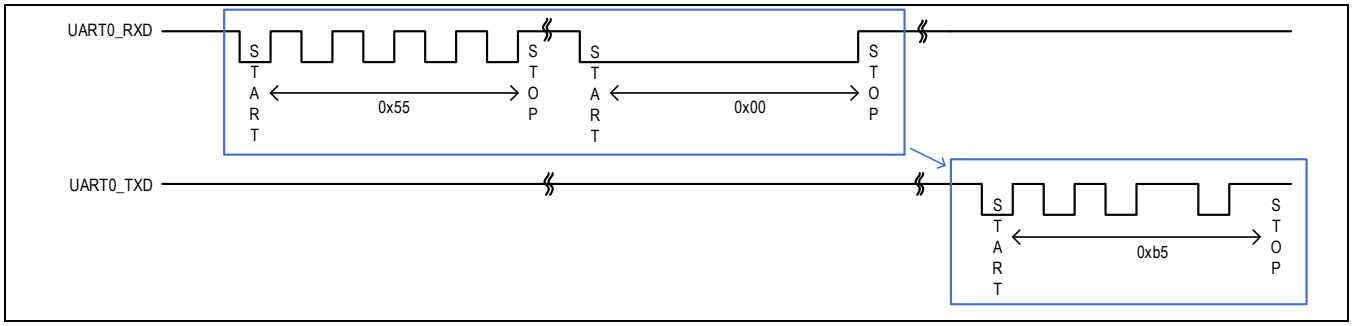


Figure 6. UART0 Bootloader Example: “Get Bootloader Version” Command 0x55 0x00 and Reply 0xb5

SPI Bootstrap Protocol

The SPI bootstrap protocol uses 32-bit datagrams with the MSB being transmitted first. All bootloader commands require one SPI datagram. After transmission, the command gets executed and any reply is placed into the SPI transmit buffer of the TMC8100. Signal SPI_DATA_AVAILABLE (has to be configured as an alternate function to GPIO6 and output to be visible externally) is pulled high to indicate new data available. A second SPI transmission is necessary to read out the reply. As SPI transmissions always include data in both directions, another command may be already included with this transmission.

The 32-bit SPI command datagram includes a read (high) or write (low) bit (RnW) as MSB, 4-bit for command encoding (CMD[3:0]), optional 11-bit for address (ADDR[10:0]), and 16-bit for data (DATA[15:0]).

Table 2. SPI Bootloader Commands

COMMAND	REPLY	DESCRIPTION
RnW = 0/1, CMD[3:0] = 0, ADDR[10:0] = 0, DATA[15:0] = 0	$b5, $00, $00, $00	Get bootloader version.
RnW = 0, CMD[3:0] = 1, ADDR[10:0] = address, DATA[15:0] = instruction	No reply	Write program memory data – 16-bit instruction at specified address.
RnW = 1, CMD[3:0] = 1, ADDR[10:0] = address, DATA[15:0] = 0	DATA[15:0] = instruction, other bits copied from command	Read program memory data – 16-bit instruction from specified address.
RnW = 0/1, CMD[3:0] = 2, ADDR[10:0] = 0, DATA[15:0] = 0	No reply	Start program – stop bootloader program execution and start program execution from SRAM/address $0000.
RnW = 0/1, CMD[3:0] = 3, ADDR[10:0] = 0, DATA[15:0] = BAUD	No reply	Enable I^2C interface and set I^2C baud rate. The command includes the I^2C baud-rate divider.
RnW = 0, CMD[3:0] = 4, ADDR[10:0] = address, DATA[15:0] = instruction	Copy of command	Write EEPROM - address and 16-bit data/instruction word must be provided. As the write access to the external EEPROM takes some time, there is a reply after the write access is finished - copy of the original command.
RnW = 1, CMD[3:0] = 4, ADDR[10:0] = address, DATA[15:0] = 0	DATA[15:0] = instruction, Other bits copied from command	Read EEPROM – address must be provided. Reply contains the 8-bit data at the specified EEPROM address and the 8-bit at the subsequent address (one 16-bit instruction word).
RnW = 0, CMD[3:0] = 5, ADDR[10:0] = *xx*, DATA[15:0] = *yy*	RnW = 0, CMD[3:0] = 5, ADDR[10:0] = *xx*, DATA[15:0] = *yy*	Reply with copy of command.
RnW = 1, CMD[3:0] = 10, ADDR[10:0] = *xx*, DATA[15:0] = *yy*	RnW = 1, CMD[3:0] = 2, ADDR[10:0] = *xx*, DATA[15:0] = *yy*	Reply with copy of command.
RnW = 1, CMD[3:0] = 6, ADDR[10:0] = 0, DATA[15:0] = 0	$38, $31, $30, $30	Get Chip ID – “8100” as ASCII character.
RnW = 1, CMD[3:0] = 7, ADDR[10:0] = 0, DATA[15:0] = 0	$11, $00, $00, $00	Get Chip Revision.

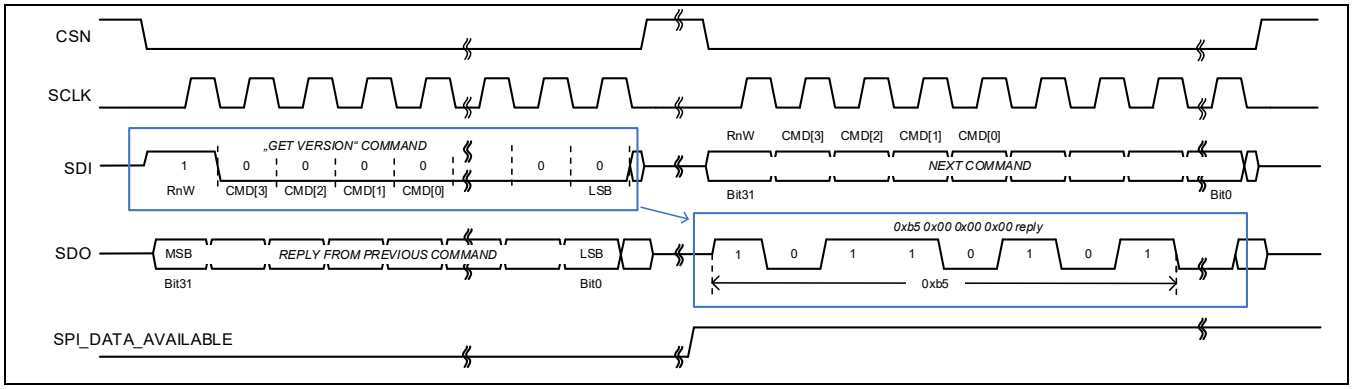


Figure 7. SPI Bootloader Example: “Get Bootloader Version” Command and Reply 0xb5, 0x00, 0x00, 0x00

Data Bus

All peripherals including UART (2x), SPI, I^2C, GPIO, and a data memory (64x8) are connected to the data bus. The data bus supports reading and writing 8-bit data with an 8-bit address (0..255). All read and write accesses take one clock cycle.

Table 3. Data Bus Address Range Assignment

ADDRESS	PERIPHERAL	DESCRIPTION
0x80 to 0xBF	SRAM 64x8 data memory	Data memory with 64 entries for storing intermediate values.
0x60 to 0x78	A/B/Z encoder interface	Read/write and configure A/B/Z incremental encoder interface.
0x40 to 0x4E	GPIO – GPIO interface and configuration	Read/write and configure GPIO0..6 pins and select alternate pin functions.
0x30 to 0x34	SPI	Read/write and configure SPI.
0x28 to 0x2B	I^2C	Read/write and configure I^2C interface.
0x20 to 0x23	DIRECT – DIRECT interface configuration	Configure DIRECT_IN0..3 and DIRECT_OUT0..3 pin signals and select alternate functions.
0x18 to 0x1C	MEM – program memory write interface	Peripheral required for write access to program memory through the data bus. Supports setting address and writing 16-bit instructions words to program memory.
0x10 to 0x15	UART1	Read/write and configure UART1 interface.
0x08 to 0x0D	UART0	Read/write and configure UART0 interface.

Power Supply

The TMC8100 supports single supply operation between 2.5V and 5V. It includes a linear regulator (LDO) for the core supply voltage. This regulator is internally connected to the V_{CCIO} I/O supply input and requires a 2.2µF ceramic capacitor at the V_{DD1V8} output pin for proper operation. The TMC8100 offers two V_{CCIO} supply inputs on pins 6 and 19, which must be connected externally.

Reset and Clock

Reset

The TMC8100 offers an internal power-on-reset (POR). In addition, there is a dedicated low-active reset input pin. This pin offers an internal pull-up. It can be used to extend the power-on reset or explicitly reset the device during operation.

Clock

The clock generation offers a high degree of flexibility and supports three different clock source options. After power-up, the digital circuit always starts on the internal oscillator (15MHz). For higher clock frequencies, an integrated PLL is available. The PLL requires an input frequency of 1MHz. The pre-diver (RDIV) must be set accordingly. After configuring the pre-divider (RDIV) and the division factor (PLL_FB_DIV), the PLL can be activated. Supported PLL output frequencies are 75MHz, 100MHz, and 128MHz. The integrated bootloader program already configures the PLL in combination with the internal oscillator for 75MHz system frequency.

As an alternative to the integrated oscillator (INT_OSC), an external clock source can be selected (CLK_EXT) or the on-chip crystal oscillator can be used for a more precise clock source. In both cases, the respective GPIO pins must be configured through the GPIO matrix. The crystal oscillator requires an external crystal for operation. After configuring the crystal oscillator, a start-up time is required before the clock signal is stable and can be selected as input for the PLL.

In case the external clock or the crystal oscillator with an external crystal is selected as clock source for the PLL, a clock loss detection is activated. This uses the internal oscillator as reference. In case a clock loss is detected, a system reset is initiated and the clock source is switched back to the internal oscillator.

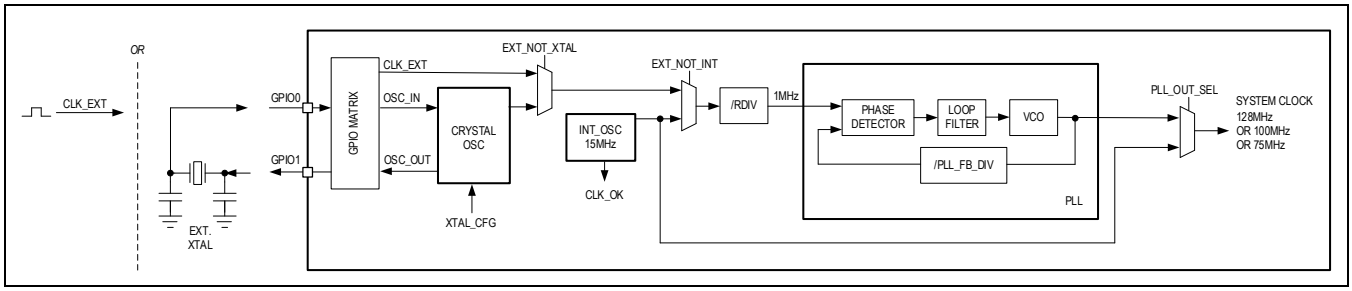


Figure 8. Clock Tree

Code example for setting different PLL output frequencies or changing clock source:

```
    LDI $03, r0 ; for EXT_CLK / XTAL - enable input for GPIO0/GPIO1
    ST GPIO_IN, r0
    LDI $03, r0 ; for EXT_CLK / XTAL - disable pull-up for GPIO0/GPIO1
    ST GPIO_PU, r0


    LDI PLL_FB_CFG, r0
    ST CLK_ADDR, r0
    LDI $36, r0 ; set pll feedback divider for 75MHz
    ;LDI $4f, r0 ; set pll feedback divider for 100MHz
    ;LDI $6b, r0 ; set pll feedback divider for 128MHz
    ST CLK_DOUT, r0 ; write access to clock control register PLL_FB_CFG


    LDI CLK_CTRL_SOURCE, r0 ; use internal clock
    ST CLK_ADDR, r0
    ;LDI $26, r0  ; use XTAL
    ;LDI $21, r0  ; use external clock
    LDI $00, r0 ; use internal clock
    ST CLK_DOUT, r0 ; write access to clock control register CLK_CTRL_SOURCE

    LDI CLK_CTRL_OPT, r0
    ST CLK_ADDR, r0
    LDI %0100_0000, r0 ; enable clock control state machine
    ST CLK_DOUT, r0 ; write access to clock control register CLK_CTRL_OPT


    LDI CLK_CTRL_PLL_CFG, r0
    ST CLK_ADDR, r0
    LDI %1011_1001, r0 ; RDIV = 14 (assuming 15MHz clock), select PLL output, start state machine
    ; LDI %1011_1101, r0 ; RDIV = 15 (assuming 16MHz clock), select PLL output, start state machine
    ST CLK_DOUT, r0 ; write access to clock control register CLK_CTRL_PLL_CFG

    LDI CLK_CTRL_PLL_CFG, r0
    ST CLK_ADDR, r0 ; set address for read from clock control register CLK_CTRL_PLL_CFG
    NOP
    NOP

WAIT_FOR_PLL_LOCK:
    LD CLK_DIN, r0; read from clock control register CLK_CTRL_PLL_CFG
    NOP
    TEST1 $7, r0
    JC WAIT_FOR_PLL_LOCK
    ; continue with 75MHz system clock
```

The clock configuration offers three user programmable registers used in the example code above accessed with the help of the CLK_ADDR, CLK_DIN and CLK_DOUT registers:

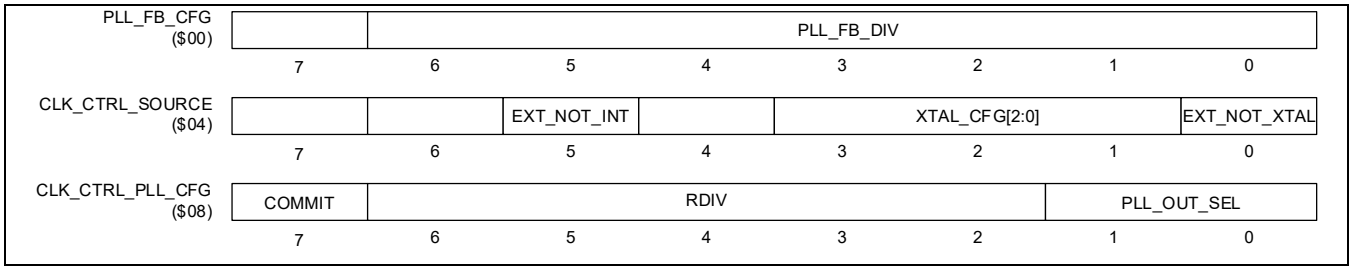


Figure 9. Clock Configuration Registers

- PLL_FB_DIV: Internal PLL divider for setting PLL multiplication factor.
- EXT_NOT_INT: External clock or crystal oscillator output (= 1) instead of internal oscillator (= 0).
- XTAL_CFG[2:0]: Crystal oscillator configuration.
- EXT_NOT_XTAL: External clock (= 1) instead of crystal oscillator output (= 0).
- COMMIT: Apply changes to clock block (= 1).
- RDIV: Clock divider for PLL input. PLL input must be 1 MHz.
- PLL_OUT_SEL: Select PLL output (= $1) instead of internal oscillator (= $0).

Crystal Oscillator

The crystal oscillator is designed to provide a programmable output current based on the quartz crystal frequency, which can be either 8MHz, 16MHz, 24MHz, 25MHz, or 32MHz.

The programmable output current is determined by 3-bit (XTAL_CFG) used to set the code assigned to each quartz crystal frequency, as shown in the following table:

XTAL_CFG	CONDITIONS	IXTAL_OUT [µA]	f_{XTAL}[MHz]
1	ESR(1)<250Ω and CL(2) = 9pF	75µA	8MHz
2	ESR>250Ω and CL = 9pF	150µA	8MHz
3	ESR<70Ω and CL = 9pF	225µA	16MHz
4	ESR<70Ω and CL = 9pF	275µA	24MHz to 25MHz
5	ESR<60Ω and CL = 9pF	450µA	32MHz

(1) ESR is the equivalent series resistance given by the quartz crystal manufacturer.

(2) CL = 9pF is recommended.

GPIO and DIRECT_IN/OUT

GPIO Matrix

All general purpose I/Os (GPIO) can be configured individually as digital input or output. After reset, all GPIOs are configured as inputs with internal pull-up to V_{CCIO}. For each GPIO, polarity of input and output can be defined (GPIO_POLARITY), output can be enabled (GPIO_OUT_EN), and for GPIO2, type of output (either push-pull or open-drain (GPIO_OUT_OD)) can be selected. Alternate function can be also configured individually per pin (GPIO_ALT*x*_FUNCTION). Note that output/input/polarity and type of output must be set correctly for a certain pin in case an alternate function is selected (example, open-drain for I^2C signals, output enable for TXD, etc.). Some peripheral units provide their own output enable signal together with the output signal for the alternate function, which overrides the output enable setting in the GPIO output enable register (example, I^2C).

In case an alternate function is selected, it is still possible to read out the current pin status using the GPIO_IN register.

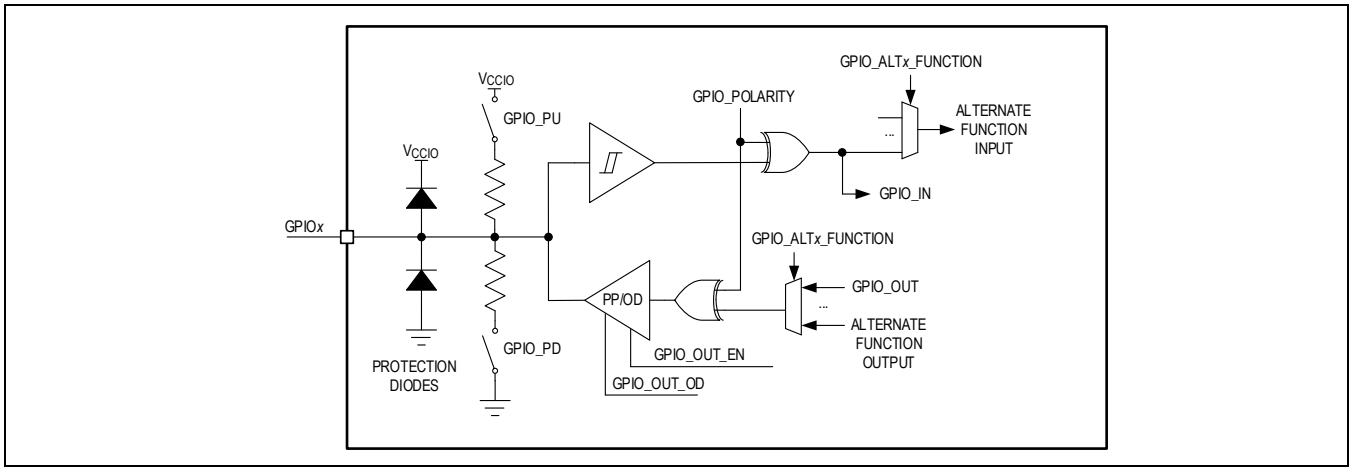


Figure 10. Basic Structure of GPIO Pin Control

Code example for toggling GPIO:

```
    ; set all gpio outputs to zero
    LDI %0000_0000, r0
    ST GPIO_OUT, r0
    ST GPIO_POLARITY, r0
    ST GPIO0_ALT0_FUNCTION, r0
    ST GPIO0_ALT1_FUNCTION, r0
    ; configure all gpio as outputs
    LDI %1111_1111, r0
    ST GPIO_OUT_ENABLE, r0
    LDI %0101_0101, r0
    LDI %00101010, r1
TOGGLE_GPIO_OUTPUT:
    ST GPIO_OUT, r0 ; GPIO6..0 -> "01010101"
    ST GPIO_OUT, r1 ; GPIO6..0 -> "00101010"
    JA TOGGLE_GPIO_OUTPUT
```

DIRECT_IN/DIRECT_OUT Matrix

The TMC8100 offers four direct inputs (DIRECT_IN0..3) and four direct outputs (DIRECT_OUT0..3), which can be accessed individually from within the serial communication engine for fast bit manipulation and sampling of the serial data stream. For each pin, polarity can be programmed individually (DIRECT_POLARITY). As an alternative to setting the output bits for DIRECT_OUT directly, a clock signal from the internal clock/timer block of the serial protocol engine can be selected (DIRECT_ALT_FUNCTION).

All DIRECT_IN pins are configured as inputs with internal pull-ups to V_{CCIO} after reset. While DIRECT_OUT0 and DIRECT_OUT1 are fixed outputs (push-pull), ENC_Z and HOME inputs are selected instead of DIRECT_OUT2 and DIRECT_OUT3 with internal pull-ups after power-up.

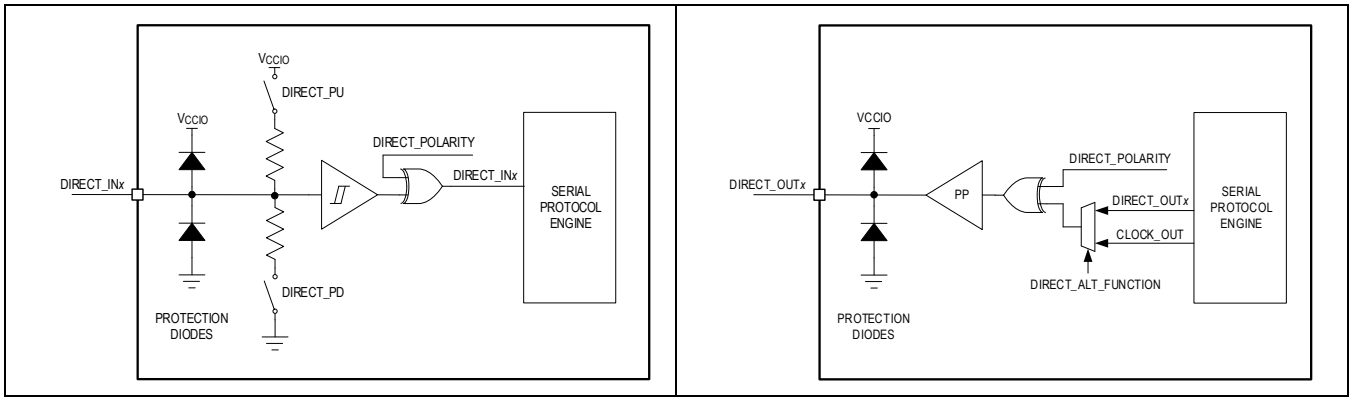


Figure 11. Basic Structure of DIRECT_IN (Left) and DIRECT_OUT (Right) Pin Control

Code example for toggling DIRECT_OUT:

```
    ; set all DIRECT_OUT to zero
    SFCLR WAIT0SF NO_WAIT, 0, 0
    SFCLR WAIT0SF NO_WAIT, 0, 1
    SFCLR WAIT0SF NO_WAIT, 0, 2
    SFCLR WAIT0SF NO_WAIT, 0, 3
    ; select DIRECT_OUT0..3 for all 4 direct connections
    LDI %0000_0000, r0
    ST DIRECT_POLARITY, r0
    ST DIRECT_ALT_FUNCTION, r0
TOGGLE_OUTPUT:
    ; toggle DIRECT_OUT(0..3): 0 -> 1 -> 0
```

```
SFSET WAIT0SF NO_WAIT, 0, 0
SFSET WAIT0SF NO_WAIT, 0, 1
SFSET WAIT0SF NO_WAIT, 0, 2
SFSET WAIT0SF NO_WAIT, 0, 3
SFCLR WAIT0SF NO_WAIT, 0, 0
SFCLR WAIT0SF NO_WAIT, 0, 1
SFCLR WAIT0SF NO_WAIT, 0, 2
SFCLR WAIT0SF NO_WAIT, 0, 3
JA TOGGLE_OUTPUT
```

Serial Communication Engine

Overview

The serial communication engine is the core part of the TMC8100. It includes a controller operating on 16-bit instructions with an 8x8-bit general purpose register set (R0…R7). The command execution pipeline includes two fetch stages and one decode/execute stage. An additional write back stage offers a bypass to reduce pipeline delays. An 11-bit program counter (PC) selects the next address from the on-chip program memory.

Also part of the core engine is a timer unit for clock generation and sampling of the incoming data stream. A programmable CRC unit supports on-the-fly CRC generation while data is being shifted in or out.

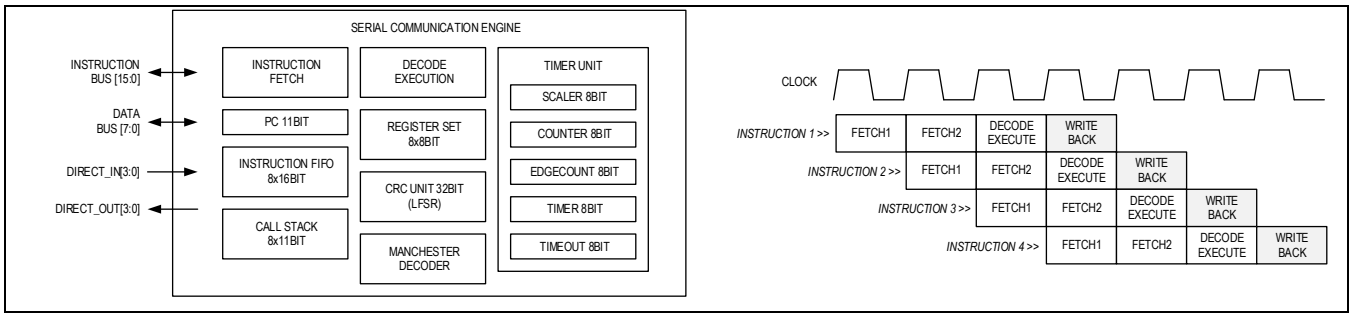


Figure 12. Serial Communication Engine Block Diagram and Instruction Pipeline

In a typical application, incoming serial data is sampled through DIRECT_IN from an attached encoder with on-the-fly extraction and alignment of encoder values. As soon as all relevant data is received, it is forwarded through one of the serial interfaces (SPI or UART) to the attached motion controller or microcontroller (store-and-forward). This way, any processing delays are minimized.

Loop Support in Hardware

When shifting in or out data, the shift operation usually must be repeated several times until all bits are in or out. A loop can be used to reduce the number of shift operations in the program code. Nevertheless, a loop requires loop cycle counting, compare, and conditional branch instructions, which introduce significant overhead not just with respect to code size but also instruction execution time.

To reduce the overhead, hardware loops are supported. During program execution, the last four instructions are always remembered in an instruction FIFO buffer. In addition, there is a dedicated hardware loop counter (up-to 8x). With the help of the REP instruction, the loop size (number of instructions) and loop counter limit are specified. The loop starts immediately after the REP instruction. This hardware loop allows for similar performance as unrolling loops during compile time while reducing code size to minimum.

Set of Counter/Timer

The serial communication engine contains a number of counter and timer units for all time dependent program execution, insertion of delay, clock generation, number of clock pulses, and timeout for all commands with variable execution time.

The programmable pre-scaler divides the main system clock by 1…256. The pre-scaler is used by the counter unit and optional (programmable) for the timer and timeout counters.

The integrated 8-bit counter uses the pre-scaler output as clock input. It is an up-counter with automatic wrap-around at its programmable upper limit (sawtooth). It can be used for clock generation. In this case, the clock output toggles at each overflow of the counter. The limit value for the counter can be calculated using the following formular:

$$SYSTEM\_TIMER\_COUNTER\_LIMIT = \frac{fprescaler}{foutput} - 1$$

Another 8-bit edge counter is available to limit the number of rising and falling edges the counter generates. The edge counter is also an up-counter that stops when reaching its upper limit (a limit of zero disables the edge counter). This way, clock signals with up-to 255 rising and falling edges can be generated (up-to 128 clock cycles with selectable rising or falling edge at the end). Commands including a wait condition offer the possibility to stop any further program execution until the counter or edge counter reaching their limits or overflow. There are instructions available for incrementing the edge counter to compensate, example, for additional processing time required by the external peripheral that receives the clock signal.

Code example for generating 3x pulses (6 edges) with a frequency of 9.375MHz (75MHz system clock):

```
LDI $01, r0
ST  DIRECT_ALT_FUNCTION, r0 ; configure DIRECT_OUT(0) as clock output
LDI $03, r0
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_COUNTER_LIMIT_W ; 75MHz / 4 toggle rate
LDI 6, r0 ; number of clock edges
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_PULS_COUNTER_LIMIT_W
LDI 1, r0 ; enable counter
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_CTRL_W
```

In addition to the clock generator, another 8-bit timer is available. This timer also offers a programmable upper limit and automatically wraps around when reaching this limit while counting up (sawtooth). The timer supports operations where a programmable amount of time must be waited before, example, data is shifted in or out through DIRECT_IN/DIRECT_OUT. The timer may also take the output of the pre-scaler as clock input in case longer delays are required.

Finally, there is a timeout counter. This is another 8-bit up-counter with programmable limit (sawtooth). It must be used together with a timeout target address register. In case the timeout limit is not zero, the timeout counter is enabled. As soon as the executed instruction includes a wait condition temporarily halting program execution, this counter starts counting. If the timeout counter reaches its limit before the wait condition is met and program execution resumed, regular program execution stops. Instead, program execution continues with the instruction at the address specified in the timeout target address register.

Description of instructions STS/LDS in the appendix contains more details on setting the timer/counter limit values.

Cyclic Redundancy Check (CRC)

The serial communication engine includes on-the-fly CRC calculation in hardware as an option for the serial bits shifted in or out through the DIRECT_IN or DIRECT_OUT pins. The generator polynomial and the start value for CRC calculation can be programmed. The CRC unit uses linear feedback shift register (LFSR) for CRC calculation. Generator polynomials up to 32-bit are supported.

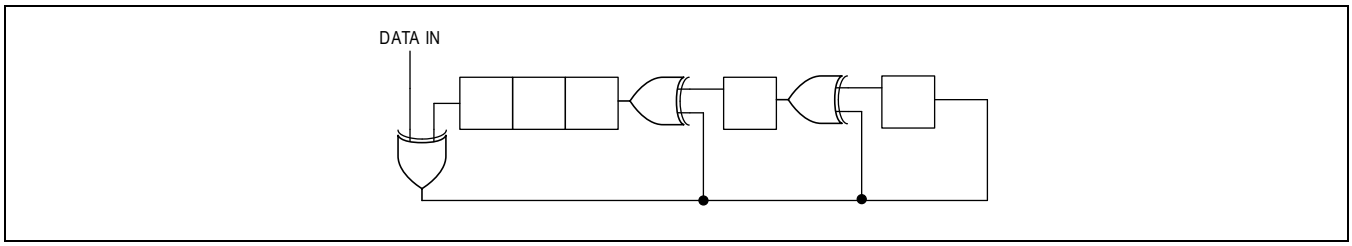
Example:

Generator polynomial: $g = x^5 \oplus x^2 \oplus x \oplus 1$

The bit sequence for this generator polynomial is 100111. This must be written to the CRC polynomial register.

An optional start value can be written to the CRC start register. Otherwise, the start value is zero.

The resulting shift register in hardware for this polynomial looks like this:



For example, if the input data stream is 10010011, the CRC checksum after shifting in these 8-bit/after 8 shifts is 1010. There are no additional cycles required for CRC checksum calculation.

The result can be read out through the CRC result register.

Note that these registers are part of the core, and therefore special load and store instructions (LDS/STS) must be used. The data bits itself can be shifted into the CRC unit in parallel with shifting in through DIRECT_IN or shifting out through DIRECT_OUT using shift-left and shift-right commands. For each shift operation, it can be decided whether the bit shifted in or out is part of the CRC calculation or not.

Description of instructions STS/LDS in the appendix contains more details on setting the CRC start/polynomial values and accessing the result.

Universal Asynchronous Receiver-Transmitter (UART)

Overview

The universal asynchronous receiver transmitter (UART) supports full-duplex data exchange with external devices using industry standard NRZ asynchronous serial data format. The UART supports autobaud (character 0x55) and offers separate transmit and receive buffers with programmable time-out. Transmission format is fixed 8n1.

Main Features

- Full duplex, asynchronous communication
- NRZ standard format (mark/space)
- Separate configurable signal polarity for transmitter/receiver
- Programmable filter for receiver input
- Configurable oversampling by a factor 16 or by a factor of 8
- Programmable baud rate generator
- Auto baud rate detection (character 0x55)
- 8-bit data word length
- One stop bit
- Transmit FIFO buffer with up to eight character entries
- Receive buffer with programmable length up-to eight characters and programmable timeout (reset buffer contents)

Functional Description

The TMC8100 includes two UART peripheral blocks, UART0 and UART1. For bidirectional connection, two pins are required for each UART: receive data (UART*x*_RXD) and transmit data (UART*x*_TXD). In case one or both UARTs are used, the GPIO matrix must be programmed accordingly to make the communication pins available externally. The features of both UARTs are the same and they operate completely independent of each other. Therefore, the following functional description covers both UARTs.

The communication format is fixed: one start bit, 8 data bits with least significant bit (LSB) first, no parity, and one stop bit (8n1). An integrated baud rate generator is available that uses the system clock as input. Either 8x or 16x oversampling can be selected and there is an optional input filter for the incoming data. The baud rate is the same for the receiver and transmitter circuit. The baud rate generator register limit value (UART*x*_BAUD_L/H) can be calculated using the following formular:

$$UARTx\_BAUD = \frac{fPLL\_CLK}{bits\_per\_second\ x\ 8} - 1$$

For x16 oversampling, the 8 in the formular must be replaced with 16. Values for common baud rates and system clock settings are:

	f_{PLL_CLK} = 75MHz, x8	f_{PLL_CLK} = 100MHz, x8	f_{PLL_CLK} = 128MHz, x8
9600bit/s	976	1301	1666
115200bit/s	81	108	138
1Mbit/s	9	12	15
8Mbit/s			1
16Mbit/s			0

Automatic baud rate detection is supported. In case autobaud is enabled, any further transmission of data through UART*x*_TXD is disabled, and the receiver expects to receive the character 0x55. The baud rate generator starts counting with the first low-to-high transition after the start bit and stops with the last low-to-high transition at the beginning of the stop bit. The result is then scaled and used as new baud rate generator limit value. The autobaud bit is reset automatically, indicating the end of the autobaud mode.

The receiver contains an 8-bit shift register for the incoming serial data and a buffer with max. eight entries. As soon as a new character is received, the data is copied to the next available receive buffer entry. The number of bytes expected for one message can be programmed (RX_BUFFER_LENGTH = 1..8 in UART*x*_CTRL). As soon as a complete message according to the programmed message length is received, the RX_FULL flag in the status register (UART*x*_STATUS) is set. The received bytes being part of one message can be read-out first-in first-out afterwards through the bus interface using register UART*x*_BUFFER. After a complete message is read-out, the buffer is ready for receiving the next message.

There is a 16-bit timeout counter available for the receiver that starts counting after the stop bit of a character is received. It continues counting as long as the receiver line remains idle. Each new character on the receiver line resets this counter. In case the timeout limit (UART*x*_TIMEOUT) is reached before the next character within a message is received, the receive buffer contents are reset, deleting any non-complete message.

The transmitter contains an eight entry FIFO transmit buffer. Any value written to the transmit buffer (UART*x*_BUFFER) is forwarded to the transmit shift register as soon as the transmit shift register is empty and any previous value in the FIFO buffer is sent out. Separate flags in the status register indicate full (TX_FULL) and empty (TX_EMPTY) FIFO buffer.

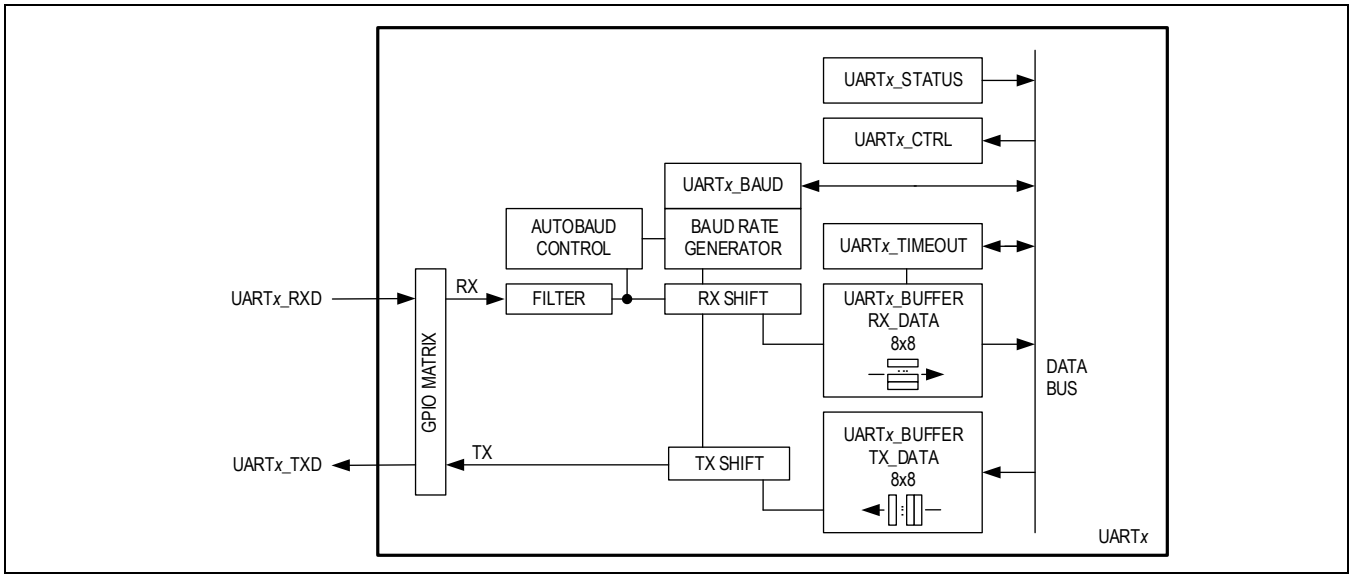


Figure 13. UART Block Diagram

Code example for UART communication:

```
LDI %0000_0101, r0
ST GPIO0_ALT1_FUNCTION, r0 ; TXD and RXD on GPIO4/5
LDI %0001_0000, r0
ST GPIO_OUT_ENABLE, r0 ; GPIO4 / TXD output
LDI %0000_0101, r0
ST UART0_CTRL, r0 ; x8, no filter, autobaud enabled
```

```
LDI UART0_STATUS, r2
 WAIT1 $0, r2 ; wait for incoming byte
 ; byte received
 LD UART0_BUFFER, r0 ; load received data into r0
...
```

Serial Peripheral Interface (SPI)

Overview

SPI block offers SPI peripheral device functionality and supports standard SPI mode 0. The SPI is one of the available serial interfaces supported by the bootloader and intended for communication with a motion-controller or microcontroller. A deep 64x32-bit entry transmit buffer for sending data back to the controller allows for high data rates while minimizing the interrupt frequency on controller side.

Main Features

The SPI peripheral block supports the following main features:

- SPI peripheral device support
- SPI mode 0
- MSB first
- 32-bit receive buffer
- 64x32-bit FIFO transmit buffer
- SPI clock up to 25MHz

Functional Description

The SPI bus interface is intended to be connected to a microprocessor or motion controller with an SPI controller interface. The SPI supports SPI mode 0 (clock polarity = 0 and clock phase = 0). In addition to four SPI signals: serial-data-out (SDO), serial-data-in (SDI), serial clock (SCLK), and chip select (CSN), an additional signal SPI_DATA_AVAILABLE is available that indicates new data available in the transmit buffer. Maximum SPI data length for a single transfer supported in hardware is 32-bit. Data is always shifted in and out MSB first.

For receiving data from the external controller, a single 32-bit buffer is available. During SPI transfer, the serial data from the SPI controller is shifted in and copied from the shift register to this buffer as soon as the SPI data transfer is completed with the rising edge of the chip select signal SPI_CSN.

For transmission of data, a FIFO buffer with 64 entries (32-bit each) is available. This way, the serial engine can fetch encoder counter values at a fixed rate while the host/microcontroller can read them out in bursts, keeping the interrupt frequency and the overhead low. In case the transmit buffer reaches its capacity fetching, further encoder data by the serial engine can be stopped (default) or older values can be discarded, keeping always the most recent ones (TX_SKIP in register SPI_CTRL). This can come into place if the controller requesting the encoder data is not fast enough or not available from time to time, and the latest data is always more important for system control than any historic values.

The transmit buffer is 32-bit in size and therefore four write accesses through the 8-bit data bus are required to fill it. The bytes must be written into the buffer most significant byte first (MSB, register SPI_BUFFER3) and least significant byte last (LSB, register SPI_BUFFER0). Following this rule, the control logic is able to detect a new 32-bit value and can automatically transfer the content of the transmit buffer to the 64 entry FIFO buffer.

The FIFO also contains an output buffer between the FIFO and transmit shift register. As soon as the shift register is empty or the last SPI transfer is finished, the content of this buffer is transferred to the shift register and the next value for the buffer is fetched from the FIFO. At the same time, the signal SPI_DATA_AVAILABLE is set to '1'. This output signal can be selected as an alternate function to pin GPIO6 and indicate any attached controller that new data is available and another SPI transaction should be initiated to read this data.

Flags in the status register (SPI_STATUS) indicate an end of SPI transmission with new data available in the receive buffer (EOT), currently no SPI transfer on-going (NO_TRANSFER), and transmit buffer full (TX_FULL).

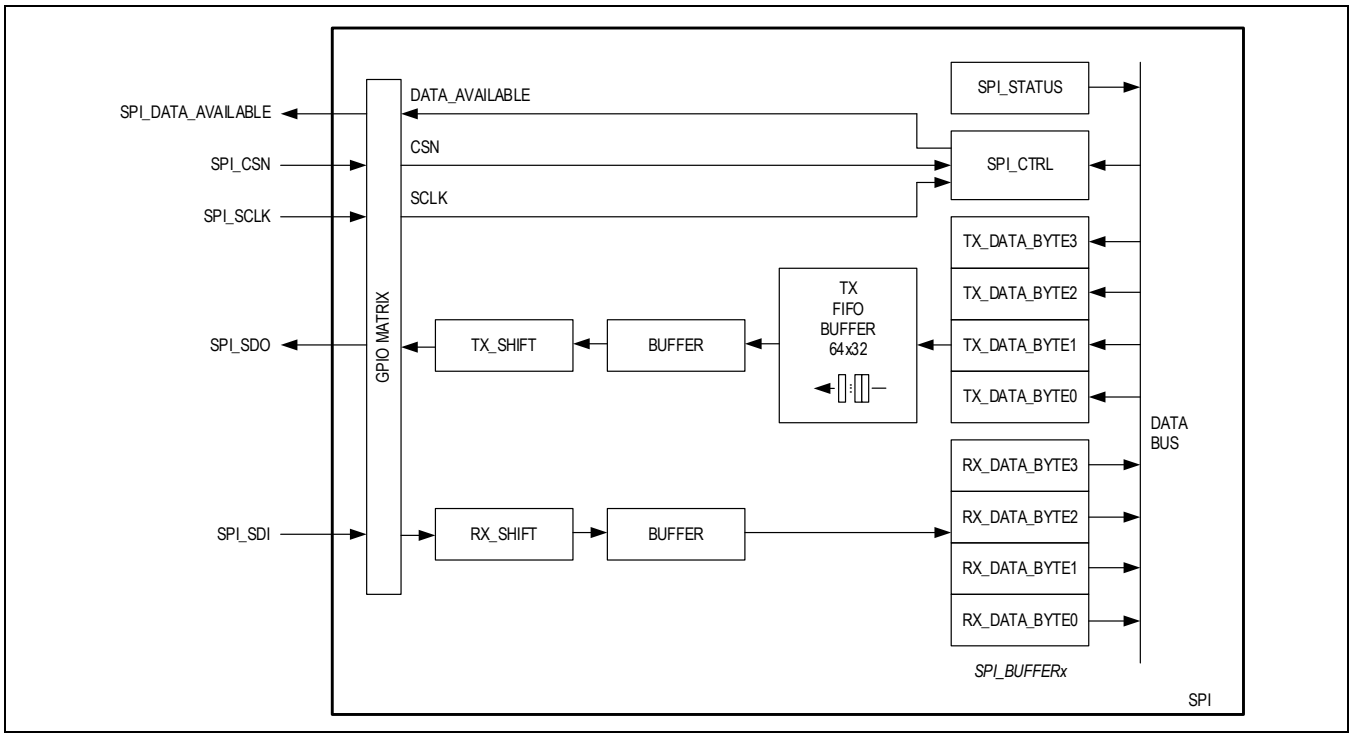


Figure 14. SPI Block Diagram

Code example for SPI communication:

```
WAIT_FOR_CMD:
    LD SPI_STATUS, r0
    NOP
    TEST1 $0, r0 ; new SPI datagram received ?
    JC SPI_CMD
    JA WAIT_FOR_CMD
    ...

SPI_CMD:
   LD SPI_BUFFER_3, r0
   LDI $01, r1
   CMP NE r0, r1 ; compare MSB of datagram with $01
   JC WAIT_FOR_CMD
   LDI $38, r0 ; "8" ; put "8100" into SPI transmit buffer
   ST SPI_BUFFER_3, r0
   LDI $31, r0 ; "1"
   ST SPI_BUFFER_2, r0
   LDI $30, r0 ; "0"
   ST SPI_BUFFER_1, r0
   ST SPI_BUFFER_0, r0
   JA WAIT_FOR_CMD
```

I²C

Overview

The I²C block supports host/controller operation. Usually, either an external I²C EEPROM for standalone operation/bootstrap or additional sensors (example, temperature) are connected here.

Main Features

- Host/Controller
- Receive shift register
- Transmit shift register
- Command buffer
- Configurable start/stop repeated start stop conditions
- 7-bit address mode
- Standard mode

Functional Description

The TMC8100 contains an I^2C host interface. This interface supports I^2C standard mode. The physical interface consists of the bidirectional serial data line I2C_SDA and the serial clock output I2C_SCL (alternate pin functions to GPIO2 and GPIO3). Note that these serial interface signals must be selected individually in the GPIO matrix to make them available externally. Also, open-drain operation instead of push-pull (default) for the SDA output must be activated explicitly in the GPIO matrix. The pull-ups to V_{CCIO} must be added externally for valid signal levels.

An integrated baud rate generator is available, which uses the system clock as input. The limit value for the baud rate generator (I2C_BAUD_L/H) can be calculated using the following formular:

$$I2C\_BAUD = \frac{fPLL\_CLK}{fI2C\_SCL\ x\ 4} - 1$$

The I^2C interface is optimized to support byte and page read and write operations in combination with an 24LC64 EEPROM or similar. Nevertheless, the I^2C host interface can be used for communication with other peripherals also.

For control of I^2C host operation, a command register is available. The following I^2C commands are supported:

COMMAND LABEL	COMMAND CODE	DESCRIPTION
I2C_CMD_STOP	0x00	Send stop condition.
I2C_CMD_START_TXD_ACK	0x01	Send start signal and transmit one byte afterwards (usually command byte). Sample/check target acknowledge.
I2C_CMD_TXD_ACK	0x02	Transmit one byte and check/sample target acknowledge.
I2C_CMD_RXD_ACK	0x03	Receive one byte and send acknowledge.
I2C_CMD_RXD_NO_ACK	0x04	Receive one byte and send no acknowledge.

In case the last command is executed and there is no new command available, an I^2C stop condition is sent automatically. In case the command does include transmission of a byte, this must be written into the transmit shift register (I2C_BUFFER) prior to command initiation. A byte received is available in the receive shift register I2C_BUFFER at the end of command execution. The status register indicates successful command execution (CMD_RDY), any acknowledge bit received (RCV_ACK), and its value (RCV_ACK_VALUE).

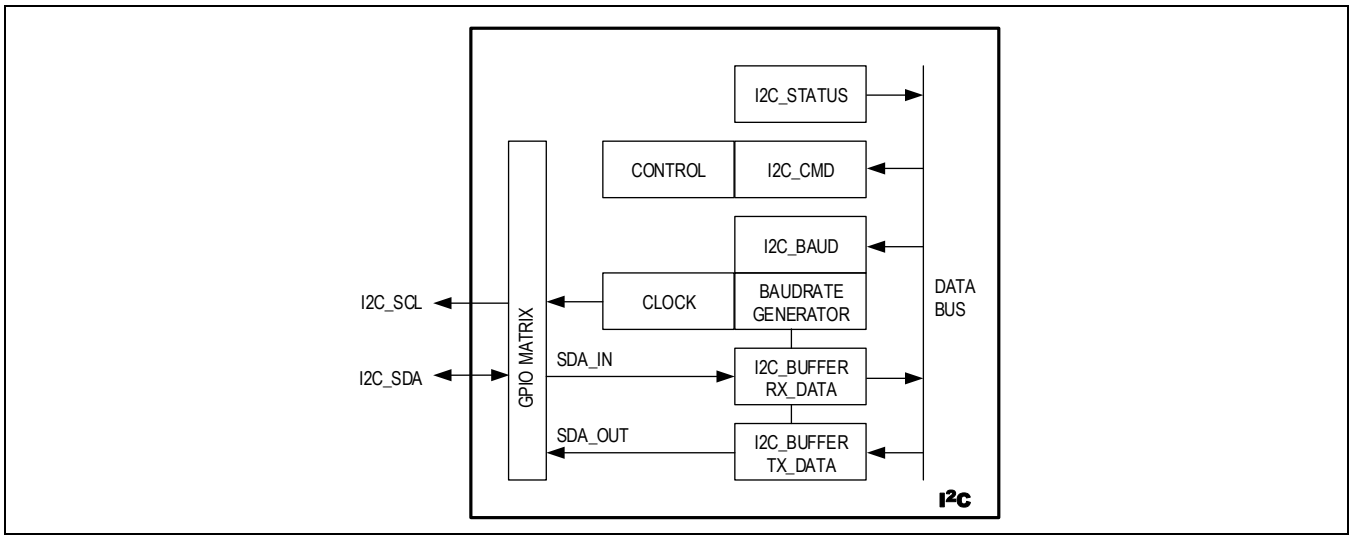


Figure 15. I²C Block Diagram

Code example for I²C communication:

```
; send command + write address + data (I2C EEPROM)
LDI $a0, r0
ST I2C_TX_BUFFER, r0
LDI I2C_CMD_START_TXD_ACK, r0
ST I2C_CMD, r0
LDI I2C_STATUS, r0
WAIT1 $0, r0
; send address high byte
ST I2C_TX_BUFFER, r4
LDI I2C_CMD_TXD_ACK, r0
ST I2C_CMD, r0
LDI I2C_STATUS, r0
WAIT1 $0, r0
; send address low byte
ST I2C_TX_BUFFER, r3
LDI I2C_CMD_TXD_ACK, r0
ST I2C_CMD, r0
LDI I2C_STATUS, r0
WAIT1 $0, r0
; send data byte
ST I2C_TX_BUFFER, r6
LDI I2C_CMD_TXD_ACK, r0
ST I2C_CMD, r0
LDI I2C_STATUS, r0
WAIT1 $0, r0
; send stop
LDI I2C_CMD_STOP, r0
ST I2C_CMD, r0
LDI I2C_STATUS, r0
WAIT1 $0, r0
```

A/B/Z Encoder Interface

Overview

The TMC8100 offers a timer block with 32-bit position counter with programmable input decoder supporting incremental (quadrature) encoder signals.

Main Features

- 32-bit position counter
- Programmable input decoder supporting A/B/Z, x1, x2, CW/CCW, STEP/DIR
- Decoder output for synchronization of external devices (with programmable pulse length)
- Programmable input filter and sampling frequency
- Programmable position counter reset on Z-channel and/or HOME switch event (once/always, programmable)
- 32-bit position capture register
- Capture encoder counter value on Z-channel/HOME switch event (once/always)
- 2x 32-bit compare register for output waveform based on position counter value
- Output pulse generation with programmable length (16-bit counter)

Functional Description

The TMC8100 contains a 32-bit counter with quadrature decoder for incremental encoder with A/B channel and optional Z channel. These encoder inputs are available as alternate functions of the DIRECT_IN pins. The matrix must be programmed accordingly to use these inputs. The encoder inputs must pass an optional filter with programmable sample rate before decoding and the main 32-bit encoder counter is incremented or decremented accordingly. The decoder supports quadrature (x4) decoding for the standard incremental encoder A and B channel signals and several other codes too (x1, x2, CW/CCW, PULSE/DIR). The encoder counter can be captured and/or reset to its start value depending on a programmable signal pattern in case of an Z channel event or an external trigger signal. This signal input has its own optional filter and programmable sample rate and can be used as single trigger source for capturing the encoder counter value or in combination with the Z channel event. The same trigger options are available for resetting the encoder counter to its programmable start value. Both capture and reset events can be enabled and accepted continuously or just once. This can be used for homing with reset and/or capture of encoder value once the home position is reached. Also, more complex homing operations are supported, example, as soon as the home switch gets activated, the next encoder Z channel event defines the precise home position (usually more precise than a mechanical home switch). The definition of a Z channel event is fully programmable (rising or falling edges of one of the A/B or Z channel can be selected while the other channels are either ignored, low, or high, for full flexibility.

For the 32-bit encoder counter, an upper wraparound limit can be defined. This way cyclic counting, example, adjusted to one motor turn is supported.

For synchronization of external devices, the encoder counter offers two programmable outputs. The decoder output (DECODER_OUT) generates one pulse with programmable length for each encoder counter increment or decrement. The additional compare output signal (COMPARE_OUT) can be configured to generate a high signal of programmable length in case the compare registers 0 and 1 are less or equal or greater than the encoder counter value.

Input and output polarities of all signals are programmable through the GPIO and DIRECT_IN matrix.

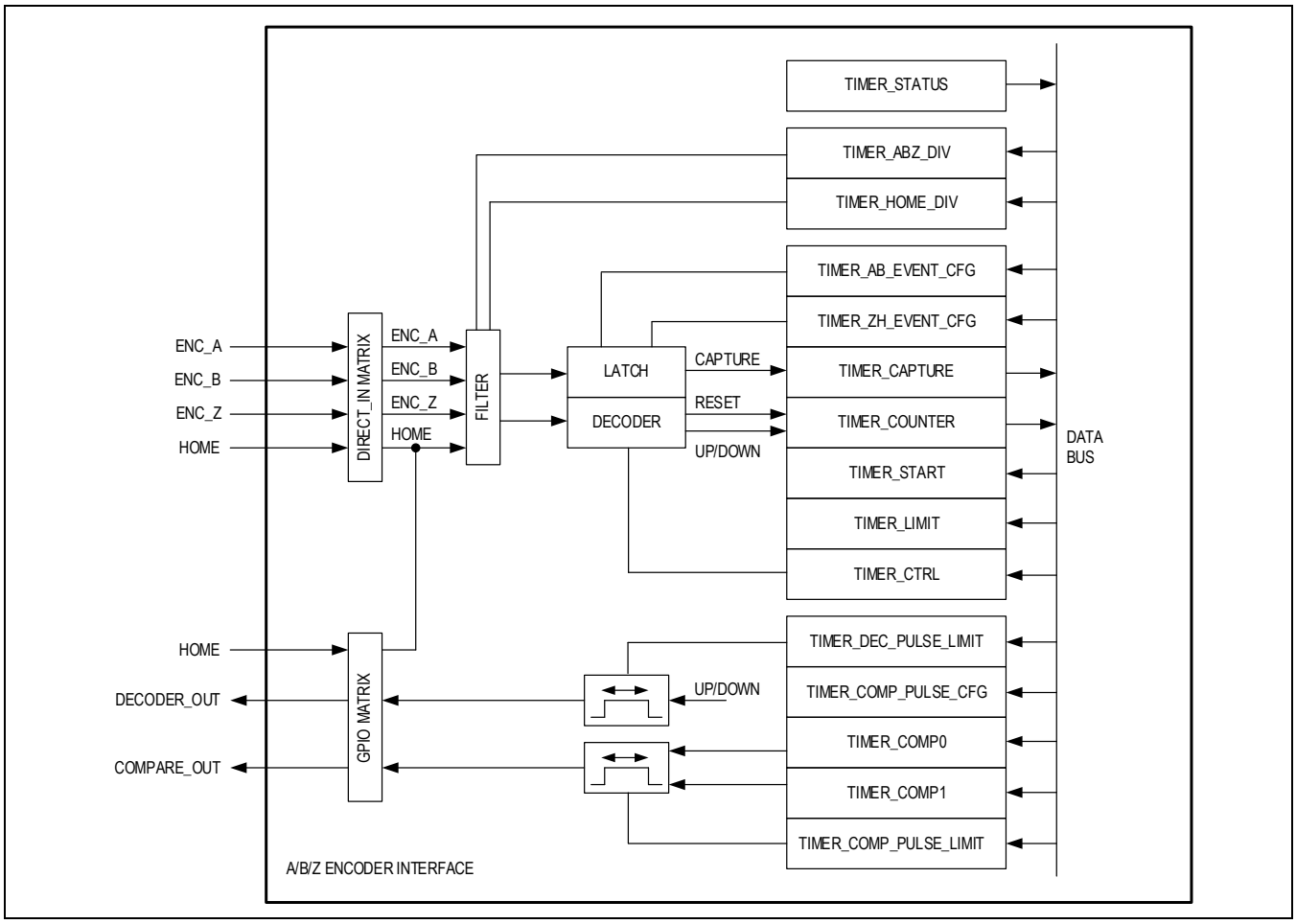
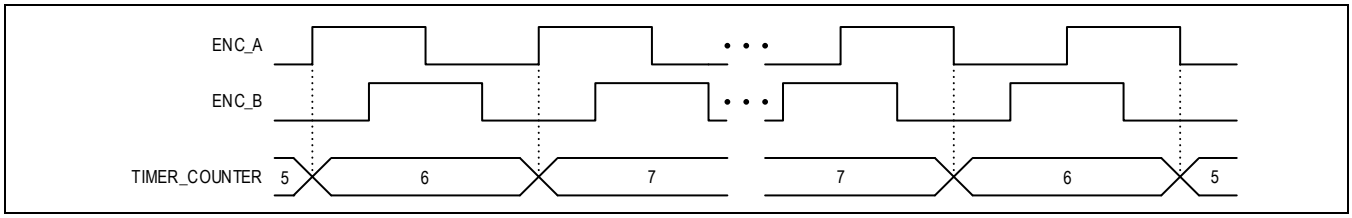


Figure 16. A/B/N Encoder Interface Block Diagram

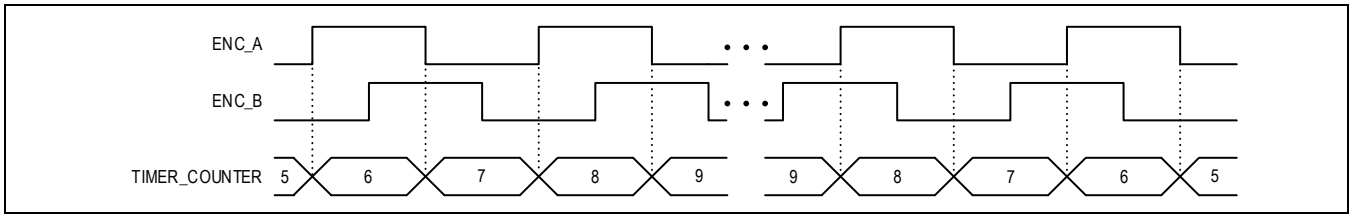
x1 Code Incremental Encoder Input

With x1 incremental code, the encoder position counter is incremented at the rising edge of channel A in case channel A is leading and decremented at the falling edge of channel A if channel B is leading.



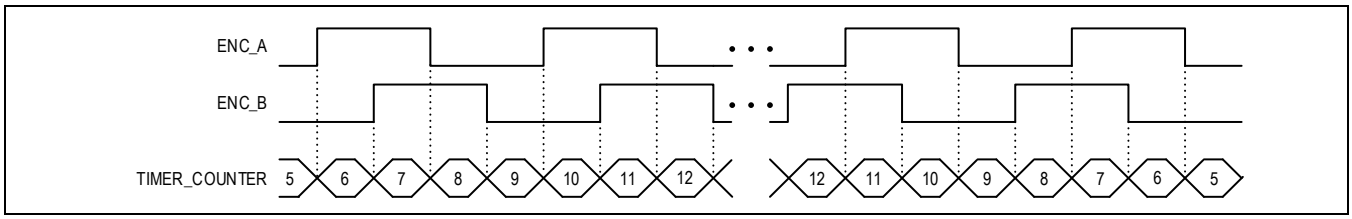
x2 Code Incremental Encoder Input

With x2 incremental code, the encoder position counter is incremented at both edges of channel A in case channel A is leading and decremented at both edges of channel A if channel B is leading.



x4 Code, A/B Incremental Encoder Input

With x4 incremental code, the encoder position counter is incremented at both edges of channel A and both edges of channel B in case channel A is leading, and decremented at both edges of channel A and both edges of channel B if channel B is leading. An additional channel N(neutral) or Z(zero) can be used to indicate zero/null position within one rotation of the encoder. A pulse on this channel can be directly indicating zero position (example, rising or falling edge) or just qualify a rising or falling edge on channel A or B as null/zero position.



Code example for A/B/N incremental encoder:

```
; capture on z-channel high and channel b rising edge
 LDI %0001_0111, r0
 ST TIMER_AB_EVENT_CFG, r0
 LDI %0011_1001, r0
 ST TIMER_ZH_EVENT_CFG, r0
 ; select x4 code, capture on z-channel
 LDI %0010_0010, r0
 ST TIMER_CTRL, r0
 ; set length of decode output signal
 LDI %0000_0010, r0
 ST TIMER_DEC_PULSE_LIMIT, r0
 ; set counter limit to max and reset counter
 LDI $ff, r0
 ST TIMER_LIMIT0, r0
 ST TIMER_LIMIT1, r0
 ST TIMER_LIMIT2, r0
 ST TIMER_LIMIT3, r0
 ...
 ; read abz encoder value
 LD TIMER_COUNTER3, r3
 LD TIMER_COUNTER2, r2
 LD TIMER_COUNTER1, r1
 LD TIMER_COUNTER0, r0
```

CW and CCW Incremental Input

With this decoder configuration, different signals are used for counting up/clock-wise (cw) counting and counting down/counter-clock-wise (ccw) counting of the encoder position counter.

PULSE/DIR Incremental Input

With this configuration, different signals are used for counting up/down and for direction control. The encoder position counter either counts up or counts downwards with each pulse/step depending on polarity of the direction input.

Appendix

Commands

The protocol engine inside the TMC8100 contains a programmable state machine. The architecture and command set are optimized for the specific purpose of converting serial data into parallel and vice versa. This way, synchronous and asynchronous bit-streams are supported with up-to 16 Mbit/s (with 128 MHz core clock frequency and eight times oversampling). The protocol engine offloads the motion controller or main general-purpose microcontroller from this conversion task, and in contrast to fully hardware-based solutions, offers a high degree of flexibility for current protocol implementations, customization, and future protocol extensions.

The protocol engine accepts a set of 16-bit wide commands while operating on 8-bit data. The command execution pipeline includes two fetch stages and one decode/execute stage. An additional write-back stage offers a bypass to reduce pipeline delays. A 12-bit program counter selects the next address from the 2048 x 16 on-chip program memory. For program branches, conditional and unconditional jumps are supported. While most instructions are executed in one clock cycle, branch instructions usually require three cycles as the command pipeline must be refilled. Nevertheless, to be able to use the otherwise empty slots after a taken branch, delayed jumps are supported. For delayed jumps, the two instructions after the jump are always executed before continuing at the jump target address. A hardware stack with eight entries supports nested subroutines with call/return instructions. Also, for small command loops with known number of cycles, hardware loopbacks with integrated instruction cache are available for loop unrolling without any instruction overhead or pipeline delay.

The load/store architecture operates on 8x 8-bit general-purpose registers. In addition, there are a number of flag registers and system registers available for accessing several timers/counters and the CRC unit integrated into the core. For the main purpose of serial/parallel data conversion, several shift and bit tests and manipulate commands are available that can be linked to timer/clock events to synchronize command processing to the serial bit stream.

To ensure highly deterministic program execution times, each instruction contains a conditional execution flag (instruction basically requires the same time whether executed or not) and there are no interrupts. Nevertheless, in combination with the core timer, block timeouts are supported while processing the data stream.

Overview

Program Flow Control

COMMAND	SYNTAX	DESCRIPTION
JA/JC	JA <addr> JC <addr>	Jump always (JA) or jump conditionally (JC) to immediate program memory address. In case the jump is taken, two additional idle cycles are inserted after this instruction before the first instruction at the target address is executed.
JFA/JFC	JFA <addr> JFC <addr>	Jump (fast) to immediate program memory address (without inserting idle cycles). Always execute the two instructions immediately after this instruction before the next instruction or the first instruction at the jump target address is executed (without idle cycles).
CALL	CALL <addr>	Jump to immediate address, remember address of next instruction on return address stack.
RSUB	RSUB	Return from sub-routine (jump back to address on top of return address stack).
REP	REP <loops>, <instr>	Hardware loop consisting of <instr> subsequent instructions (<instr> = 1...4 instructions supported). Loop is executed <loops> + 1 time without jump back overhead (no additional cycles) using instruction loop cache (<loops> = 0...7 - 0...7 jump backs/1..8 loop execution supported in hardware).
WAIT0	WAIT0 <bit>, <reg>	Stop program execution until <bit> of register at peripheral address <reg> is zero.
WAIT1	WAIT1 <bit>, <reg>	Stop program execution until <bit> of register at peripheral address <reg> is one.
WAIT0SF	WAIT0SF <wait_flag>, <wait_ctrl>	Stop program execution until <wait_flag> is zero, then perform action according to <wait_ctrl>.
WAIT1SF	WAIT1SF <wait_flag>, <wait_ctrl>	Stop program execution until <wait_flag> is one, then perform action according to <wait_ctrl>.
NOP	NOP	No operation.

HALT	HALT	Stop program counter (do not use during regular program flow).

<addr> immediate 11-bit address value 0...2047

<bit> bit within one byte 0...7

<reg> any general purpose register 0...7

<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Pulse counter has reached pulse counter limit
6	Overflow timer
7	No wait

<wait_ctrl>	DESCRIPTION
0	No action
1	Start timer
2	Stop timer
3	No action
4	If DIRECT_IN[0] is 0/1 increment pulse counter limit
5	If DIRECT_IN[1] is 0/1 increment pulse counter limit
6	If DIRECT_IN[2] is 0/1 increment pulse counter limit
7	If DIRECT_IN[3] is 0/1 increment pulse counter limit

Load/Store/Move Operations

COMMAND	SYNTAX	DESCRIPTION
LD	LD <addr>, <reg>	Read from data memory/peripheral address <addr> and load into register <reg>.
ST	ST <addr>, <reg>	Store register contents <reg> at data memory/peripheral address <addr>.
LDI	LDI <data>, <reg>	Load <data> value into register.
LDR	LDR <regy>, <regz>	Load value from data memory/peripheral at address provided in <regy> and store value in register <regz>.
STR	STR <regy>, <regz>	Store register <regz> value at data memory/peripheral address given in register <regy>.
LDS	LDS <system_unit>, <system_reg_read>, <reg>	Store contents of <system_reg> part of <system_unit> in <reg>.
STS	STS <reg>, <system_unit>, <system_reg_write>	Store contents of <reg> in <system_reg> part of <system_unit>.

<addr> immediate (part of the instruction word) 8-bit data memory/peripheral address 0...255

<data> immediate (part of the instruction word) 8-bit data 0...255

<reg>, <regy>, <regz> any general purpose register 0...7

<system_unit>	<system_reg_read>	DESCRIPTION
0: Core	0	Program source Bit 0 – 0: ROM bootloader Bit 0 – 1: SRAM program memory
1: Timer	1	Counter value
	2	Pulse counter value
	3	Timer value
	4	Timeout counter value
2: CRC	0	CRC result [7:0]

	1	CRC result [15:8]
	2	CRC result [23:16]
	3	CRC result [31:24]

<system_unit>	<system_reg_write>	DESCRIPTION
0: Core	0	Select program source Bit 0 – 0: ROM bootloader Bit 0 – 1: SRAM program memory
	1	Bit 0 – DIRECT_IN[3:0] input filter enable Bit 2, 1 – DIRECT_IN[3:0] filter sample scaler (/1, /8, /64, /512) Bit 3 – Select Manchester decoder
	2	Timeout jump target address [7:0]
	3	Timeout jump target address[10:8]
	4	Manchester decoder sample window low [4:0]
	5	Manchester decoder sample window high [4:0]
1: Timer	0	Pre-scaler limit
	1	Counter limit
	2	Pulse counter limit
	3	Timer limit
	4	Timeout counter limit
	5	Bit 0 – Counter enable Bit 1 – Timer enable Bit 2 – Select pre-scaler for timer
	7	Timer limit (without resetting timer)
2: CRC	0	Circular buffer for writing 32-bit CRC start value beginning with the LSB (CRC start value[7:0])
	1	Circular buffer for writing 32-bit CRC polynomial beginning with the LSB (CRC polynomial[7:0])
	2	Bit 0 – CRC polynomial[32] Bit 1 – CRC out in reverse order When writing to this register the write buffer pointer for the 32-bit CRC start value and 32-bit CRC polynomial value is reset to the first entry/LSB.

Set/Clear/Move Individual Bits

COMMAND	SYNTAX	INSTRUCTION FORMAT
SET	SET <bit>, <regy>, <regz>	Copy contents of <regy> to <regz> and set <bit> to '1'.
CLR	CLR <bit>, <regy>, <regz>	Copy contents of <regy> to <regz> and clear <bit> to '0'.
SFSET	SFSET WAIT0SF <wait_flag>, <flag_reg_out>, <bit>	Write '1' to <bit> of <flag_reg_out> as soon as <wait_flag> condition is '0'.
	SFSET WAIT1SF <wait_flag>, <flag_reg_out>, <bit>	Write '1' to <bit> of <flag_reg_out> as soon as <wait_flag> condition is '1'.
SFCLR	SFCLR WAIT0SF <wait_flag>, <flag_reg_out>, <bit>	Write '0' to <bit> of <flag_reg_out> as soon as <wait_flag> condition is '0'.
	SFCLR WAIT1SF <wait_flag>, <flag_reg_out>, <bit>	Write '0' to <bit> of <flag_reg_out> as soon as <wait_flag> condition is '1'.
MOVB0	MOVB0 <bit>, <regy>, <regz>	Overwrite bit 0 of <regz> with <bit> of <regy>.
MOVB7	MOVB7 <bit>, <regy>, <regz>	Overwrite bit 7 of <regz> with <bit> of <regy>.
MOVCRC	MOVCRC <bit>, <regz>	Move <bit> of <regz> to serial input of CRC unit.
MOVNCRC	MOVNCRC <bit>, <regz>	Move inverted <bit> of <regz> to serial input of CRC unit.
MOVF	MOVF <bit>, <regz>	Overwrite <bit> of <regz> with flag status.
MOVNF	MOVNF <bit>, <regz>	Overwrite <bit> of <regz> with inverted flag status.

<bit>: bit within register byte 0...7

<regy>, <regz>: any general purpose register 0...7

<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow flag counter
5	Overflow flag timer
6	Overflow flag pulse counter
7	No wait

<flag_reg_out>	DESCRIPTION
0	Bit 0 – DIRECT_OUT[0] Bit 1 – DIRECT_OUT[1] Bit 2 – DIRECT_OUT[2] Bit 3 – DIRECT_OUT[3] Bit 4 – DIRECT_OUT[0] + CRC unit serial in Bit 5 – DIRECT_OUT[1] + CRC unit serial in Bit 6 – DIRECT_OUT[2] + CRC unit serial in Bit 7 – DIRECT_OUT[3] + CRC unit serial in
1	Bit 0 – DIRECT_OUT[0] enable (push-pull) Bit 1 – DIRECT_OUT[1] enable (push-pull) Bit 2 – DIRECT_OUT[2] enable (push-pull) Bit 3 – DIRECT_OUT[3] enable (push-pull)
2	Bit 0 – CRC unit
3	Bit 0 – counter enable Bit 1 – timer enable Bit 2 – timeout counter enable
4	Bit 0 – counter reset Bit 1 – timer reset Bit 2 timeout counter reset

Arithmetic and Logic Operations

COMMAND	SYNTAX	DESCRIPTION
AND	AND <regx>, <regy>, <regz>	Store result of <regx> and (bitwise) <regy> in <regz>.
OR	OR <regx>, <regy>, <regz>	Store result of <regx> or (bitwise) <regy> in <regz>.
XOR	XOR <regx>, <regy>, <regz>	Store result of <regx> exclusive or (bitwise) <regy> in <regz>.
NOT	NOT <regy>, <regz>	Store inverted (bitwise) value of <regy> in <regz>.
REV	REV <regy>, <regz>	Reverse bits in <regy> and store result in <regz>.
ADD	ADD <regx>, <regy>, <regz>	Add <regx> to <regy> and store result in <regz>.
SUB	SUB <regx>, <regy>, <regz>	Substract <regy> from <regx> and store result in <regz>.
INC	INC <regy>, <regz>	Increment <regy> and store result in <regz>.
DEC	DEC <regy>, <regz>	Decrement <regy> and store result in <regz>.

Compare and Test Operations

COMMAND	SYNTAX	DESCRIPTION
COMP LT	COMP LT <regy>, <regz>	If <regy> is less than <regz>, the flag is set – otherwise cleared.
COMP LE	COMP LE <regy>, <regz>	If <regy> is less than or equal to <regz>, the flag is set – otherwise cleared.
COMP EQ	COMP EQ <regy>, <regz>	If <regy> is equal to <regz>, the flag is set – otherwise cleared.
COMP NE	COMP NE <regy>, <regz>	If <regy> is not equal to <regz>, the flag is set – otherwise cleared.
TEST0	TEST0 <bit>, <reg>	If <bit> of <reg> is ‘0’, the flag is set – otherwise cleared.

TEST1	TEST1 <bit>, <reg>	If <bit> of <reg> is '1', the flag is set – otherwise cleared.
SFTEST0	SFTEST0 <flag_reg_in>, <bit>	If <bit> of <flag_reg_in> is '0', the flag is set – otherwise cleared.
SFTEST1	SFTEST1 <flag_reg_in>, <bit>	If <bit> of <flag_reg_in> is '1', the flag is set – otherwise cleared.

<regx>, <regy>, <regz>: any general purpose register

<bit>: bit within byte 0...7

<flag_reg_in>	DESCRIPTION
0	Bit 0 – DIRECT_IN[0] Bit 1 – DIRECT_IN[1] Bit 2 – DIRECT_IN[2] Bit 3 – DIRECT_IN[3]
1	Bit 0 – clock generator output Bit 1 – set to one in case pulse counter has reached limit value

Shift Operations

COMMAND	SYNTAX	INSTRUCTION FORMAT
SHLO	SHLO WAIT0SF <wait_flag>, <out_flag>, <reg>	Shift <reg> left one bit as soon as <wait_flag> is '0' and output MSB to <out_flag>.
	SHLO WAIT1SF <wait_flag>, <out_flag>, <reg>	Shift <reg> left one bit as soon as <wait_flag> is '1' and output MSB to <out_flag>.
SHLI	SHLI WAIT0SF <wait_flag>, <reg>, <in_flag>	Shift <reg> left one bit as soon as <wait_flag> is ,0' with LSB from <in_flag>.
	SHLI WAIT1SF <wait_flag>, <reg>, <in_flag>	Shift <reg> left one bit as soon as <wait_flag> is ,1' with LSB from <in_flag>.
SHRO	SHRO WAIT0SF <wait_flag>, <reg>, <out_flag>	Shift <reg> right one bit as soon as <wait_flag> is '0' and output LSB to <out_flag>.
	SHRO WAIT1SF <wait_flag>, <reg>, <out_flag>	Shift <reg> right one bit as soon as <wait_flag> is '1' and output LSB to <out_flag>.
SHRI	SHRI WAIT0SF <wait_flag>, <in_flag>, <reg>	Shift <reg> right one bit as soon as <wait_flag> is ,0' with MSB from <in_flag>.
	SHRI WAIT1SF <wait_flag>, <in_flag>, <reg>	Shift <reg> right one bit as soon as <wait_flag> is ,1' with MSB from <in_flag>.

<reg>: any general purpose register 0...7

<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Overflow pulse counter
6	Overflow timer
7	No wait

<out_flag>	DESCRIPTION
0	DIRECT_OUT[0]
1	DIRECT_OUT[1]
2	DIRECT_OUT[2]
3	DIRECT_OUT[3]
4	DIRECT_OUT[0] and CRC unit serial in

5	DIRECT_OUT[1] and CRC unit serial in
6	DIRECT_OUT[2] and CRC unit serial in
7	DIRECT_OUT[3] and CRC unit serial in

<in_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	DIRECT_IN[0] and CRC unit serial in
5	DIRECT_IN[1] and CRC unit serial in
6	DIRECT_IN[2] and CRC unit serial in
7	DIRECT_IN[3] and CRC unit serial in

JA/JC (Jump Always/Jump Conditionally)

Operation:

Jump always (JA) or jump conditionally (JC) to immediate program memory address. The immediate address is always the address of an instruction word (16-bit) in program memory (either bootloader ROM or program memory SRAM). Execution of the instruction itself requires one clock cycle. In case the jump is taken, there is an additional pipeline delay of two clock cycles before the instruction at the specified jump target program memory address is executed.

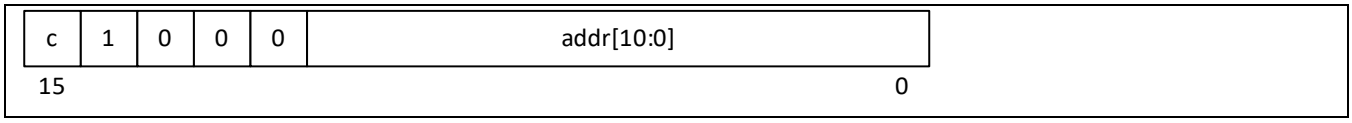
Assembler Syntax:

JA <addr>

JC <addr>

<addr>: program memory address (jump target) 0..2047

Instruction Format:



c: condition flag

- 0: Always execute jump instruction/jump always (JA)
- 1: Execute jump instruction in case flag is '1'/jump conditionally (JC)

addr[10:0] immediate address of jump target instruction. Specifies any instruction within 2Kx16 (4KB) program memory area 0...2047.

Example:

```
...
CLK_DIN = $4a
...

WAIT_FOR_PLL:
  LD CLK_DIN, r0
  NOP
  TEST1 $7, r0
  JC WAIT_FOR_PLL
```

In this example, the jump back to the start of the loop takes place in case the TEST1 instruction immediately before the JC instruction is successful and the flag bit is set. The assembler supports symbolic names for jump addresses and calculates the address automatically (in this case "WAIT_FOR_PLL"). Note the ':' behind the placeholder for the address - indicating that the current program memory address is assigned to this placeholder instead of a value.

JFA/JFC (Jump Fast Always/Jump Fast Conditionally)

Operation:

Jump fast always (JFA) and jump fast conditional (JFC) to immediate program memory address. The immediate address is always the address of an instruction word (16-bit) in program memory (either bootloader ROM or program memory SRAM). Execution of the instruction itself requires one clock cycle. The next two instructions located immediately after the jump instruction in the program code are always executed (whether the jump is taken or not). This way, no additional wait cycles are necessary in case the jump is taken.

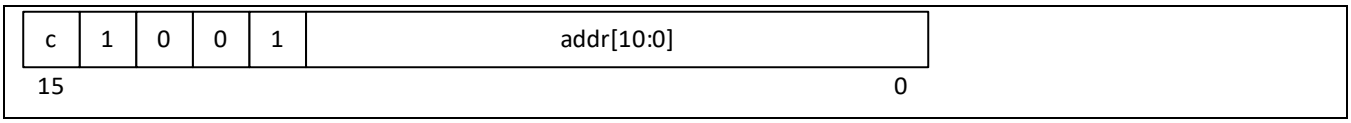
Assembler Syntax:

JFA <addr>

JFC <addr>

<addr>: program memory address (jump target) 0..2047

Instruction Format:



C: condition flag

- 0: Always execute jump instruction/jump fast always (JFA)
- 1: Execute jump instruction in case flag is '1'/jump fast conditionally (JFC)

addr[10:0] immediate address of jump target instruction. Specifies any instruction within 2Kx16 (4KB) program memory area 0…2047.

Example:

```
...
GPIO_OUT = $40
...

WAIT:
  LDI %0101_0101, r0
  ST GPIO_OUT, r0
  JFA WAIT
  LDI %1010_1010, r0
  ST GPIO_OUT, r0
```

In this example, the jump back to the start of the loop always takes place. The two instructions after the JFA WAIT command at the end of the example code snippet are executed before the first instruction at the start of the loop is executed again. The code sequence results in toggling of the GPIO outputs (01010101 → 10101010 → 01010101 → ...). The assembler supports symbolic names for jump addresses and calculates the address automatically (in this case "WAIT"). Note the ':' behind the placeholder for the address - indicating that the current program memory address is assigned to this placeholder instead of an explicitly assigned value.

CALL (Call Subroutine)

Operation:

Branch to subroutine. The immediate address is always the address of an instruction word (16-bit) in the program memory (either bootloader ROM or program memory SRAM). Execution of the instruction itself requires one clock cycle. For the unconditional CALL command, the next two instructions located immediately after the CALL instruction in the program code are always executed before the program jump takes place. This way, no additional wait cycles are necessary. For the conditional CCALL instruction, there is an additional delay of two clock cycles automatically inserted before the instruction at the specified branch target program memory address is executed. In case the branch is taken, the return address (the address of the instruction immediately after the CALL instruction) is stored on a return stack. The dedicated return stack avoids any additional clock cycles required otherwise for memory access to store the return address. The return stack offers a maximum of eight entries. This limits the number of nested branches to subroutines (call of another subroutine within a subroutine) to 8.

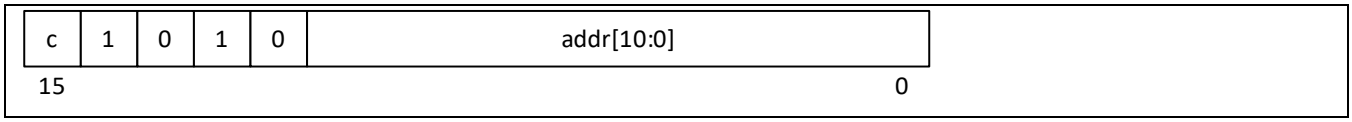
Assembler Syntax:

CALL <addr>

CCALL <addr>

<addr>: program memory address (start of subroutine) 0…2047

Instruction Format:



C: condition flag

- 0: Always execute call instruction/branch to subroutine (CALL)
- 1: Execute call instruction/branch to subroutine in case flag is '1'/(CCALL)

addr[10:0] immediate address of branch target instruction. Specifies any instruction within 2Kx16 (4KB) program memory area 0…2047.

Example:

```
...
GPIO_OUT  = $40   ; 0100_0000
GPIO_IN   = $40
...
CALL TOGGLE_GPIO
NOP
NOP
...
TOGGLE_GPIO:
  LD GPIO_IN, r0
  LDI $ff, r1
  RSUB
  XOR r0, r1, r0
  ST GPIO_OUT, r0
```

In this example, the program branch/call of the subroutine TOGGLE_GPIO always takes place. The two NOP instructions immediately following the CALL instruction in program code are executed before the first instruction of the subroutine LD GPIO_IN, r0 is executed. At the end of the subroutine, the RSUB command initiates a jump back to the calling routine. The two instructions after the RSUB command (XOR ...) are still executed before the first NOP instruction immediately following the CALL instruction in the main function is executed.

The assembler supports symbolic names for jump addresses and calculates the address automatically (in this case "TOGGLE_GPIO"). Note the ':' behind the placeholder for the address - indicating that the current program memory address is assigned to this placeholder instead of an explicitly assigned value.

RSUB (Return from Subroutine)

Operation:

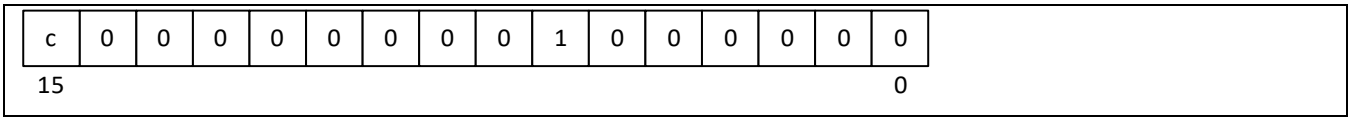
Return from subroutine. This command does not require any parameter. Instead, the branch target address is taken from the top of the hardware return stack. Execution of the instruction itself requires one clock cycle. For the unconditional RSUB command, the next two instructions located immediately after the RSUB instruction in the program code are executed before the instruction at the branch target address is executed. This way, no additional wait cycles are necessary for the jump back. For the conditional RSUB instruction, two idle clock cycles are inserted automatically before the instruction at the branch target is executed.

Assembler Syntax:

RSUB

CRSUB

Instruction Format:



C: condition flag

- 0: Always execute instruction/return from subroutine (RSUB)
- 1: Execute instruction/branch back from subroutine to calling function in case flag is '1'/(CRSUB)

Example:

```
...
GPIO_OUT  = $40   ; 0100_0000
GPIO_IN   = $40
...
CALL TOGGLE_GPIO
NOP
NOP
...
TOGGLE_GPIO:
  LD GPIO_IN, r0
  LDI $ff, r1
  RSUB
  XOR r0, r1, r0
  ST GPIO_OUT, r0
```

In this example, the program branch/call of the subroutine TOGGLE_GPIO always takes place. The two NOP instructions immediately following the CALL instruction in program code are executed before the first instruction of the subroutine LD GPIO_IN, r0 is executed. At the end of the subroutine, the RSUB command initiates a jump back to the calling routine. The two instructions after the RSUB command (XOR ...) still are executed before the first NOP instruction immediately following the CALL instruction in the main function is executed.

The assembler supports symbolic names for jump addresses and calculates the address automatically (in this case "TOGGLE_GPIO"). Note the ':' behind the placeholder for the address - indicating that the current program memory address is assigned to this placeholder instead of an explicitly assigned value.

REP (Repeat/Initialize Hardware Loop)

Operation:

Initialize hardware loop. This command supports loop unrolling in hardware at program execution time to eliminate the additional clock cycles for loop counting and jump back for repeated execution of loop instructions. Traditional loop unrolling at compile time typically increases program length significantly. With loop unrolling in hardware, just the additional command for initialization (REP) is required. Execution of this command takes one clock cycle. The loop starts immediately after this instruction.

During regular program execution, all instructions executed are remembered using a first-in first-out (FIFO) buffer with four entries. This buffer is used for repeated execution of instructions during loop unrolling. Instructions are seamlessly fetched from the FIFO buffer after the loop is executed for the first time avoiding additional clock cycles/overhead for jump back and instruction fetching. There is a hardware counter available that limits the number of loops being executed. A hardware loop may contain up to four instructions (1...4) and supports up-to eight times (1…8) loop execution.

Assembler Syntax:

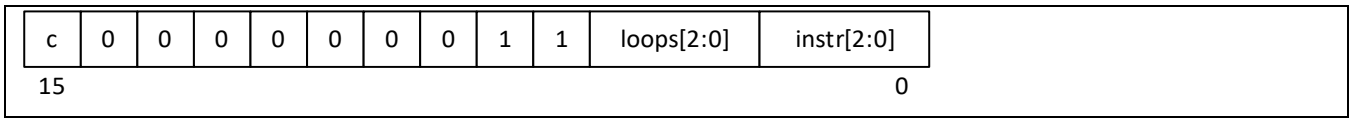
REP <loops>, <instr>

CREP <loops>, <instr>

<loops>: Loop is repeatedly executed <loops> times (<loops> = 1..8x loop execution).

<instr>: Loop consists of <instr> subsequent instructions (<instr> = 1..4 instructions supported).

Instruction Format:



loops[2:0]: 1..8 loops are encoded as 0...7

instr[2:0]: 1..4 instructions are encoded as 0...3

C: condition flag

- 0: Always execute instruction/initialize hardware loop (REP)
- 1: Execute instruction/initialize hardware loop in case flag is '1'/(CREP)

Example:

```
; <wait_flag>
WAIT_OVERFLOW_TIMER = 6

 ; <in_flag>
FLAG_IN1_CRC = 5
...

REP 4, 1
; wait for timer overflow and shift in data
SHRI WAIT1SF WAIT_OVERFLOW_TIMER, FLAG_IN1_CRC, r3
REP 8, 1
; wait for timer overflow and shift in data
SHRI WAIT1SF WAIT_OVERFLOW_TIMER, FLAG_IN1_CRC, r4
...
```

In this example, the first SHRI command (shift data bits in) is repeated four times and the second SHRI command eight times. In both cases, just one command is repeatedly executed. Short loops benefit more from hardware loop unrolling as the overhead in software required otherwise for counting loops and jumping back dominates loop execution time.

WAIT0/WAIT1 (Wait with Program Execution)

Operation:

Wait with further program execution until register bit (example, status flag) of peripheral register connected to data bus has changed to zero (WAIT0) or one (WAIT1). In case the specified bit is already zero/one, execution of the instruction takes just one clock cycle. Otherwise, the specified register is read during each clock cycle and checked for the status of the bit within this register. As soon as the bit has changed, program execution continues. This instruction can be used to synchronize program execution to external signals, serial data received, or timer events.

Assembler Syntax:

WAIT0 <bit>, <reg>

CWAIT0 <bit>, <reg>

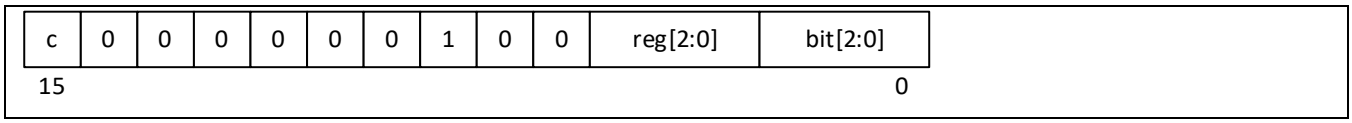
WAIT1 <bit>, <reg>

CWAIT1 <bit>, <reg>

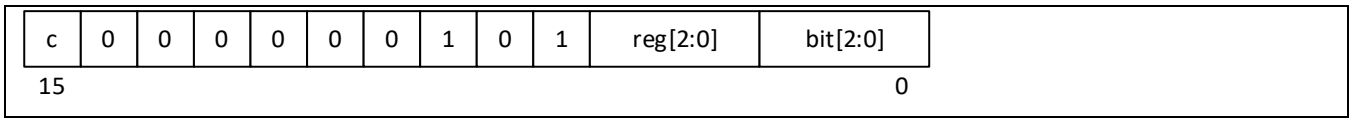
<bit>: bit within byte that is monitored (0…7)

<reg>: register (0...7) with address of peripheral register (0...255)

Instruction Format WAIT0:



Instruction Format WAIT1:



c: condition flag

- 0: Always execute instruction/wait
- 1: Execute instruction/wait in case flag is '1'/(CWAIT0/1)

Example:

```
...
UART0_BUFFER = $08
UART0_STATUS = $0b
...
LDI UART0_STATUS, r2
WAIT1 $0, r2
LD UART0_BUFFER, r1
...
```

In this example, the address of the UART0 status register (UART0_STATUS) is loaded into register r2. Program execution waits until bit 0 of the status register gets one (byte received). Immediately afterwards, data byte received is read out from the UART0 receive buffer register (UART0_BUFFER).

WAIT0SF/WAIT1SF (Wait with Program Execution)

Operation:

Wait with further program execution until selected system flag <wait_flag> has turned to zero (WAIT0) or one (WAIT1). In case the specified system flag is already zero/one, execution of the instruction takes just one clock cycle. Otherwise, the specified flag is read during each clock cycle and status/value is checked. As soon as the flag has changed, the specified action <wait_ctrl> is initiated and program execution continues without any further delay. This instruction can be used to synchronize program execution to external signals or timer events.

Assembler Syntax:

WAIT0SF <wait_flag>, <wait_ctrl>

CWAIT0SF <wait_flag>, <wait_ctrl>

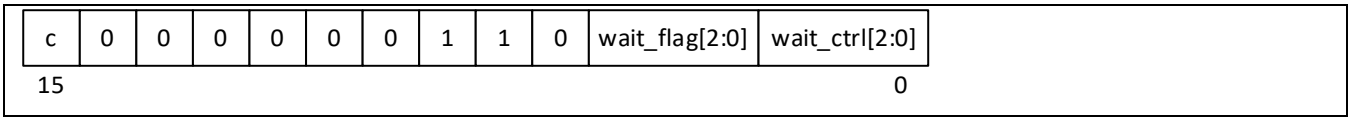
WAIT1SF <wait_flag>, <wait_ctrl>

CWAIT1SF <wait_flag>, <wait_ctrl>

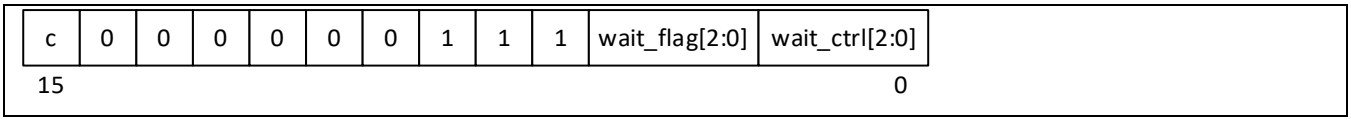
<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Pulse counter has reached pulse counter limit
6	Overflow timer
7	No wait

<wait_ctrl>	DESCRIPTION
0	No action
1	Start timer
2	Stop timer
3	No action
4	If DIRECT_IN[0] is 0/1 increment pulse counter limit
5	If DIRECT_IN[1] is 0/1 increment pulse counter limit
6	If DIRECT_IN[2] is 0/1 increment pulse counter limit
7	If DIRECT_IN[3] is 0/1 increment pulse counter limit

Instruction Format (WAIT0SF):



Instruction Format (WAIT1SF):



C: condition flag

- 0: Always execute instruction/wait
- 1: Execute instruction/wait in case flag is '1'/(CWAIT0SF/CWAIT1SF)

Example:

```
; <wait_flag>
WAIT_IN0 = 0
WAIT_IN1 = 1
```

```
WAIT_IN2 = 2
WAIT_IN3 = 3
WAIT_OVERFLOW_COUNTER = 4
WAIT_OVERFLOW_PULSE = 5
WAIT_OVERFLOW_TIMER = 6
NO_WAIT = 7

; <wait ctrl>
WAIT_NO_ACTION = 0
WAIT_START_TIMER = 1
WAIT_STOP_TIMER = 2
WAIT_IN0_INC_PULSE = 4
WAIT_IN1_INC_PULSE = 5
WAIT_IN2_INC_PULSE = 6
WAIT_IN3_INC_PULSE = 7

...
WAIT0SF WAIT_IN1, WAIT_START_TIMER
...
```

Wait for rising edge (0 → 1) on DIRECT_IN[1] (WAIT_IN1) and then start timer (WAIT_START_TIMER).

NOP (No Operation)

Operation:

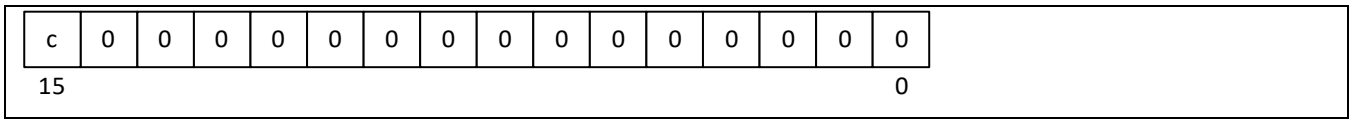
No operation. This command does not require any parameter and executes in one clock cycle. Note: NOP and the conditionally executed CNOP instruction have the same effect on program execution.

Assembler Syntax:

NOP

CNOP

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute instruction in case flag is '1'/(CNOP)

Example:

```
...
STATUS = $4c
...
LD STATUS, r0
NOP
TEST1 $2, r0
...
```

In this example, the contents of a peripheral status register are copied into register r0. As this requires one additional clock cycle, a NOP instruction is inserted before the register contents are available and can be tested with the TEST1 instruction.

HALT (Stop Program Execution)

Operation:

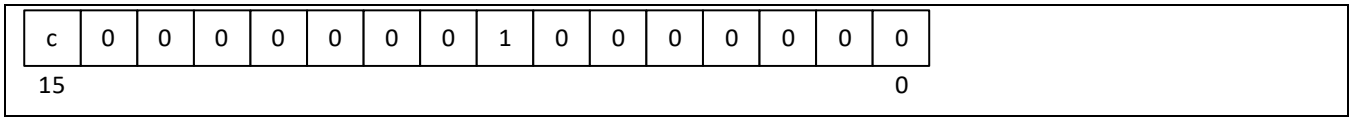
This instruction is automatically inserted into the instruction pipeline in case the execution stage is waiting for some event. The execution of this instruction takes one clock cycle but in contrast to the NOP instruction, the program counter is not incremented. Therefore, this instruction should not be used within regular program code.

Assembler Syntax:

HALT

CHALT

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute instruction in case flag is '1'/(CHALT)

LD (Load Data from Immediate Address)

Operation:

Load value from data memory or peripheral register through data bus into processor register. The data memory/peripheral register address is part of the instruction word. Any processor register can be selected as target register. The execution of this instruction takes one clock cycle. Note that the selected value is not immediately available after execution of this command. It requires one more clock cycle before the value is available in the processor register for further processing due to the data memory pipeline.

Assembler Syntax:

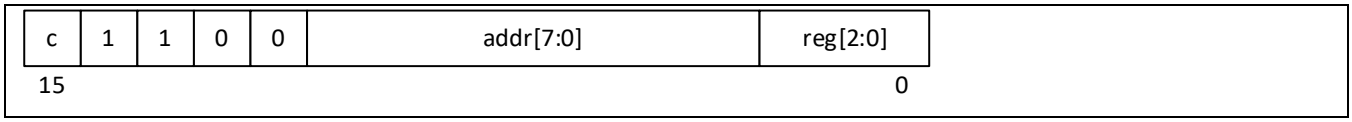
LD <addr>, <reg>

CLD <addr>, <reg>

<addr>: data memory/peripheral register address 0…255

<reg>: target register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute LD instruction in case flag is '1'/load conditionally (CLD)

addr[7:0] immediate data memory/peripheral register address. Specifies any location within 256 byte data memory area 0...255.

Example:

```
STATUS = $4c
...
LD STATUS, r0
NOP
TEST1 $2, r0
...
```

In this example, the contents of the system status register are loaded into processor register 0. A NOP instruction is inserted immediately afterwards before the contents of the register 0 is accessed and tested.

Example:

```
; gpio
GPIO0_ALT1_FUNCTION = $44
GPIO_OUT_ENABLE = $45
...
LD GPIO0_ALT1_FUNCTION, r0
LD GPIO_OUT_ENABLE, r1
SET $0, r0, r0
CLR $5, r1, r1
ST GPIO0_ALT1_FUNCTION, r0
ST GPIO_OUT_ENABLE, r1
...
```

In this second example, the content of GPIO0 alternate function register is loaded into register 0 and the contents of the GPIO output enable register into register 1 before both registers are modified. Note that both registers are loaded with one clock cycle delay before the new contents of the registers are accessed. By rearranging instructions, it is possible to fill the gap with a "useful"/required instruction instead of inserting a NOP.

ST (Store Data at Immediate Address)

Operation:

Store register value at data memory location or peripheral register. The data memory/peripheral register address is part of the instruction word. Any processor register can be selected as source register. The execution of this instruction takes one clock cycle.

Assembler Syntax:

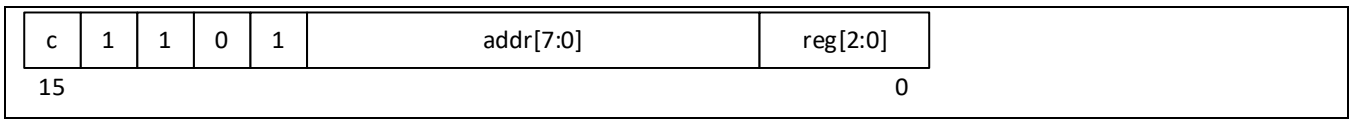
ST <addr>, <reg>

CST <addr>, <reg>

<addr>: data memory/peripheral register address 0...255 of target

<reg>: source register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute ST instruction in case flag is '1'/load conditionally (CST)

addr[7:0] immediate data memory/peripheral register address. Specifies any location within 256 byte data memory area 0…255.

Example:

```
; gpio
GPIO0_ALT1_FUNCTION = $44
GPIO_OUT_ENABLE = $45
...
LD GPIO0_ALT1_FUNCTION, r0
LD GPIO_OUT_ENABLE, r1
SET $0, r0, r0
CLR $5, r1, r1
ST GPIO0_ALT1_FUNCTION, r0
ST GPIO_OUT_ENABLE, r1
...
```

In this example, the contents of GPIO0 alternate function register are loaded into register 0 and the contents of the GPIO output enable register into register 1 before both registers are modified. Both registers are loaded with one clock cycle delay before the new contents of the registers are accessed. By rearranging instructions, it is possible to fill the gap with a "useful"/required instruction instead of inserting a NOP. At the end of the example, both registers r0 and r1 are copied to peripheral register locations (r0 → GPIO_ALT1_FUNCTION, r1 → GPIO_OUT_ENABLE).

LDI (Load Immediate Data)

Operation:

Load immediate 8-bit value (part of the instruction) into processor register. Any processor register can be selected as target register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the value is immediately available for further processing in the next clock cycle/with the next instruction.

Assembler Syntax:

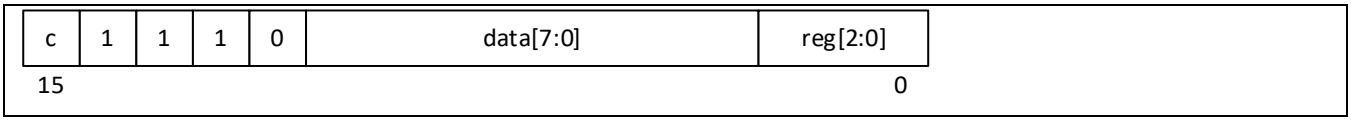
LDI <data>, <reg>

CLDI <data>, <reg>

<data>: immediate data value 0...255 (part of the instruction word)

<reg>: processor target register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute LDI instruction in case flag is '1'/load conditionally (CLDI)

data[7:0] immediate data value 0...255.

reg[2:0] processor register

Example:

```
UART0_CTRL = $0b
...
; 8x sampling, filter, autobaud enable, message_size = 0
LDI %0000_0101, r1
ST UART0_CTRL, r1
...
```

In this example, the peripheral control register of UART0 is initialized with a constant value.

LDR (Load Data from Register Address)

Operation:

Load value from data memory/peripheral register at address taken from processor register into target register. Any general-purpose processor register can be selected as register with address value and as target register. The execution of this instruction takes one clock cycle. Note that the data transfer from data memory to processor register takes another clock cycle due to the data memory access pipeline. Therefore, the value from data memory/peripheral register is available with one cycle delay in the target register for further processing.

Assembler Syntax:

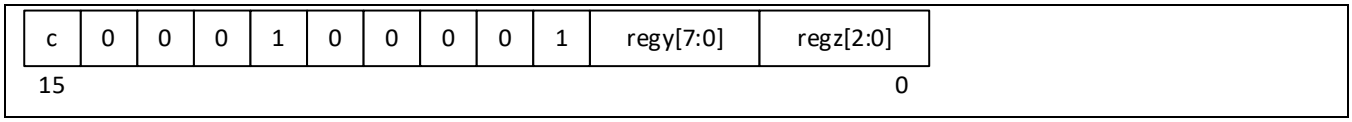
LDR <regy>, <regz>

CLDR <regy>, <regz>

<regy>: general purpose register with data memory address location 0...7

<regz>: general purpose target register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute LDR instruction in case flag is '1'/load conditionally (CLDR)

Example:

```
...
DATA_MEM_BASE = $C0 ; data memory start address
...
LDI DATA_MEM_BASE, r3
LDR r3, r0
LDI $02, r1
ADD r0, r1, r2
```

In this example, the start address of the data memory is stored in register r3. With the next LDR instruction, the value stored at this address is loaded into register r0. A constant value ($02) is then loaded into register r1, filling in also the additional cycle required until the value from memory is available in the register set for further processing. Finally, the constant value and the value loaded from data memory are added and the result is stored in register r2.

STR (Store Data at Register Address)

Operation:

Store contents of processor register in data memory or peripheral register at address given in another processor register. Any general-purpose processor register can be selected as source register and address register. The execution of this instruction takes one clock cycle. Note that the data transfer from the processor to data memory or peripheral block takes another clock cycle due to the data memory access pipeline.

Assembler Syntax:

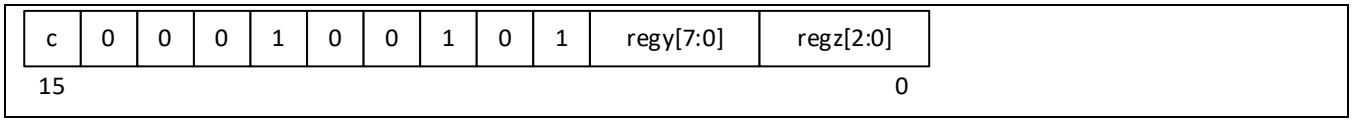
STR <regy>, <regz>

CSTR <regy>, <regz>

<regy>: general purpose register 0...7 with data memory or peripheral register address (0...255)

<regz>: general purpose register 0...7 with source data value

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute LDR instruction in case flag is '1'/load conditionally (CLDR)

Example:

```
...
DATA_MEM_BASE = $C0 ; data memory start address
...
LDI DATA_MEM_BASE, r0
LDI $05, r1
STR r0, r1
...
```

In this example, the start address of the data memory is stored in register r0 and a constant value ($05) into register r1. With the final STR command, this constant value in register r1 is stored in the data memory block (with the address taken from processor register r0).

LDS (Load Data from System Register)

Operation:

Load value from system register into processor register. Any readable system unit register is supported as source register. Any general-purpose processor register can be selected as target.

Assembler Syntax:

LDS <system_unit>, <system_reg>, <reg>

CLDS <system_unit>, <system_reg>, <reg>

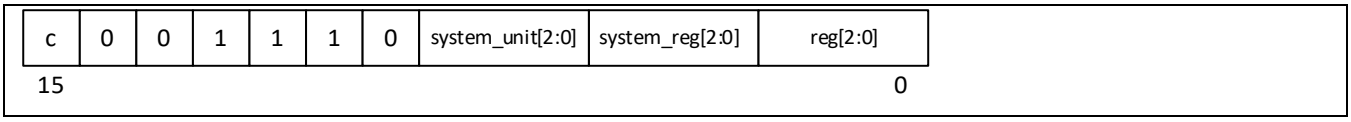
<system_unit>: system unit 0...7

<system_reg>: system register 0...7 in system unit selected

<reg>: general purpose processor register 0...7

<system_unit>	<system_reg>	DESCRIPTION
0: Core Unit	0	Program memory selected for execution Bit 0 – 0: ROM bootloader Bit 0 – 1: SRAM program memory
1: Timer Unit	1	Counter value
	2	Pulse counter value
	3	Timer value
	4	Timeout counter value
2: CRC Unit	0	CRC result [7:0]
	1	CRC result [15:8]
	2	CRC result [23:16]
	3	CRC result [31:24]

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute STS instruction in case flag is '1'/load conditionally (CSTS)

Example:

```
...
; system unit
SYSTEM_CRC = $2
...
; system crc unit
SYSTEM_CRC_RESULT0_R = $0
...
LDS SYSTEM_CRC, SYSTEM_CRC_RESULT0_R, r0
...
```

In this example, the result from the CRC calculation in system register SYSTEM_CRC_RESULT0 of the CRC unit is loaded into the general-purpose processor register r0.

STS (Store Data in System Register)

Operation:

Store value from processor register in system register. Any general-purpose processor register can be used as source register. Any writable system register is supported as target.

Assembler Syntax:

STS <reg>, <system_unit>, <system_reg>

CSTS <reg>, <system_unit>, <system_reg>

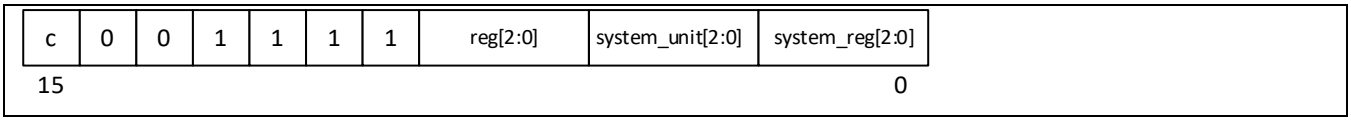
<reg>: general purpose processor register 0...7

<system_unit>: system unit 0...7

<system_reg>: system register 0...7 in selected system unit

<system_unit>	<system_reg>	DESCRIPTION
0: Core	0	Select program memory for execution. Bit[0] - 0: ROM bootloader Bit[0] - 1: SRAM program memory
	1	Bit 0: DIRECT_IN[3:0] input filter enable Bit 2,1: DIRECT_IN[3:0] filter sample scaler (/1, /8, /64, /512) Bit 3: Select manchester decoder
	2	Timeout jump target address [7:0]
	3	Timeout jump target address [10:8]
	4	Manchester decoder sample window low [4:0]
	5	Manchester decoder sample window high [4:0]
1: Timer	0	Pre-scaler limit
	1	Counter limit (reset counter)
	2	Pulse counter limit (reset pulse counter)
	3	Timer limit (reset timer)
	4	Timeout counter limit (reset timeout counter)
	5	Bit 0: Counter enable (0: reset counter) Bit 1: Timer enable (0: reset timer) Bit 2: Select pre-scaler for timer
	7	Timer limit (without resetting timer)
2: CRC	0	Circular buffer for writing 32-bit CRC start value: 1st write: CRC start value [7:0] ...
	1	Circular buffer for writing 32-bit CRC polynomial: 1st write: CRC polynomial[7:0] ...
	2	Bit 0: CRC polynomial[32] Bit 1: Reverse CRC result[31:0] When writing to this register, the write buffer pointer for the 32-bit CRC start value and 32-bit CRC polynomial value is reset to the first entry - to CRC start value [7:0]/CRC polynomial [7:0].

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute STS instruction in case flag is '1'/load conditionally (CSTS)

Example:

```
...
; system register
SYSTEM_TIMER = $1
...
```

```
 ; system timer unit
SYSTEM_TIMER_CTRL_W = $5
...
LDI 1, r0 ; enable counter
STS r0, SYSTEM_TIMER, SYSTEM_TIMER_CTRL_W
...
```

With the first instruction, a constant value is loaded into the general-purpose processor register r0. With the second instruction, this value is then stored in the timer control register SYSTEM_TIMER_CTRL_W of the system timer unit SYSTEM_TIMER.

SET (Set Register Bit)

Operation:

Set selected bit 0…7 (one bit) of source register value to '1' and store result in destination register. Any general-purpose register can be selected as source and destination register. The contents of the destination register are overwritten while the content of the source register remains untouched. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified target register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

SET <bit>, <regy>, <regz>

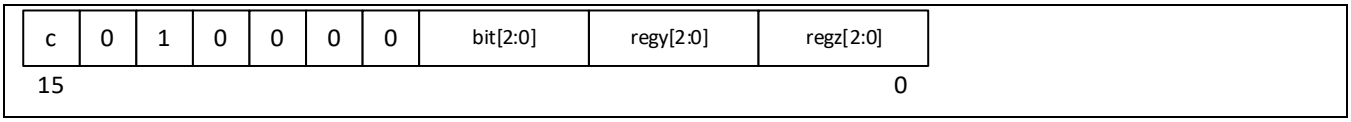
CSET <bit>, <regy>, <regz>

<bit>: bit within register 0...7

<regy>: processor source register 0…7

<regz>: processor destination register 0…7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute SET instruction in case flag is '1'/load conditionally (CSET)

Example:

```
...
SET $2, r0, r3
...
```

In this example, bit 2 of processor register r0 is set to '1' and the result is written back to register r3.

CLR (Clear Register Bit)

Operation:

Clear selected bit 0...7 (one bit) of source register value to '0' and store result in destination register. Any general-purpose register can be selected as source and destination register. The contents of the destination register are overwritten while the contents of the source register remain untouched. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified target register can be used already as source for the next instruction during the next clock cycle.

Assembler Syntax:

CLR <bit>, <regy>, <regz>

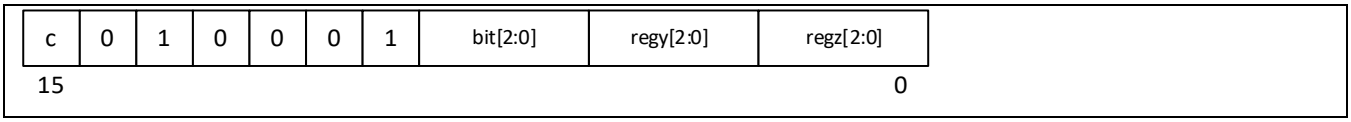
CCLR <bit>, <regy>, <regz>

<bit>: bit within register 0...7

<regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute CLR instruction in case flag is '1'/load conditionally (CCLR)

Example:

```
...
CLR $3, r0, r0
...
```

In this example, bit 3 of the content of general-purpose processor register r0 is cleared/set to '0' and the result is written back into register r0, overwriting the contents of r0.

SFSET (Set System Register Bit)

Operation:

Wait with further program execution until selected system flag has turned to zero (WAIT0SF) or one (WAIT1SF). In case the specified wait flag is already zero/one, execution of the instruction takes just one clock cycle. Otherwise, the specified wait flag is read during each clock cycle and status/value is checked. As soon as the flag has changed, the specified bit <bit> within the specified system flag register <flag_reg> is set to '1' and program execution continues. This instruction can be used to synchronize flag modification and further program execution to external signals or timer events.

Assembler Syntax:

SFSET WAIT0SF <wait_flag>, <flag_reg>, <bit>

CSFSET WAIT0SF <wait_flag>, <flag_reg>, <bit>

SFSET WAIT1SF <wait_flag>, <flag_reg>, <bit>

CSFSET WAIT1SF <wait_flag>, <flag_reg>, <bit>

<wait_flag>: bit within register 0...7

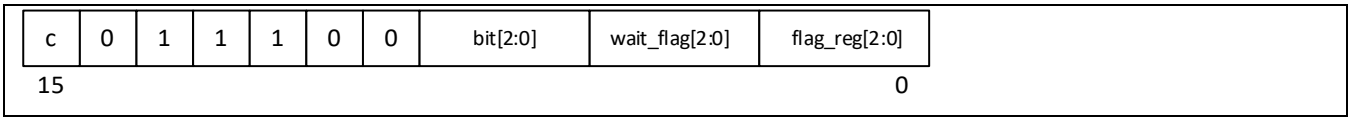
<flag_reg>: system flag register 0...7

<bit>: bit within system register 0...7

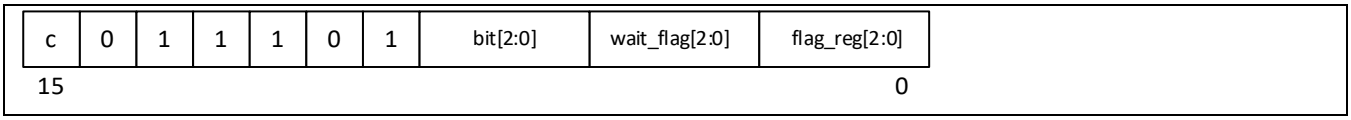
<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Overflow timer
6	Overflow pulse counter
7	No wait

<flag_reg>	DESCRIPTION
0	Bit 0: DIRECT_OUT[0] Bit 1: DIRECT_OUT[1] Bit 2: DIRECT_OUT[2] Bit 3: DIRECT_OUT[3] Bit 4: DIRECT_OUT[0] + CRC unit in Bit 5: DIRECT_OUT[1] + CRC unit in Bit 6: DIRECT_OUT[2] + CRC unit in Bit 7: DIRECT_OUT[3] + CRC unit in
1	Bit 0: DIRECT_OUT[0] enable Bit 1: DIRECT_OUT[1] enable Bit 2: DIRECT_OUT[2] enable Bit 3: DIRECT_OUT[3] enable
2	Bit 0: CRC unit
3	Bit 0: counter enable Bit 1: timer enable Bit 2: timeout counter enable
4	Bit 0: counter reset Bit 1: timer reset Bit 2: timeout counter reset

Instruction Format (SFSET WAIT0SF):



Instruction Format (SFSET WAIT1SF):



c: condition flag

- 0: Always execute instruction
- 1: Execute SFSET instruction in case flag is '1'/set conditionally (CSFSET)

Example:

```
...
SFSET WAIT0SF NO_WAIT, 0, 1
...
```

In this example, DIRECT_OUT[1] is set to '1'

SFCLR (Clear System Register Bit)

Operation:

Wait with further program execution until selected system flag has turned to zero (WAIT0SF) or one (WAIT1SF). In case the specified wait flag is already zero/one, execution of the instruction takes just one clock cycle. Otherwise, the specified wait flag is read during each clock cycle and status/value is checked. As soon as the flag has changed, the specified bit <bit> within the specified system flag register <flag_reg> is cleared to '0' and program execution continues. This instruction can be used to synchronize flag modification and further program execution to external signals or timer events.

Assembler Syntax:

SFCLR WAIT0SF <wait_flag>, <flag_reg>, <bit>

CSFCLR WAIT0SF <wait_flag>, <flag_reg>, <bit>

SFCLR WAIT1SF <wait_flag>, <flag_reg>, <bit>

CSFCLR WAIT1SF <wait_flag>, <flag_reg>, <bit>

<wait_flag>: bit within register 0...7

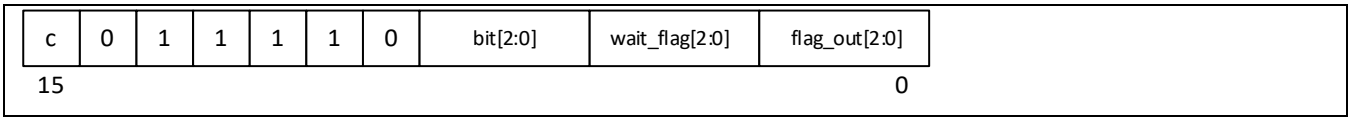
<flag_reg>: system flag register 0...7

<bit>: bit within system register 0...7

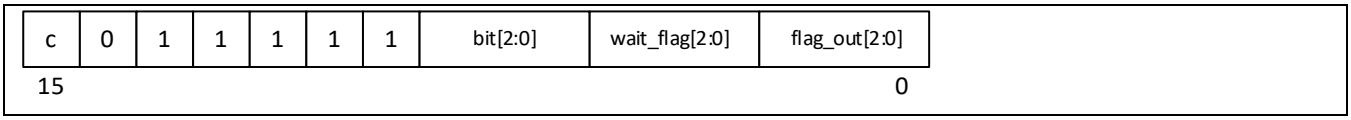
<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Overflow timer
6	Overflow pulse counter
7	No wait

<flag_reg>	DESCRIPTION
0	Bit 0: DIRECT_OUT[0] Bit 1: DIRECT_OUT[1] Bit 2: DIRECT_OUT[2] Bit 3: DIRECT_OUT[3] Bit 4: DIRECT_OUT[0] + CRC unit in Bit 5: DIRECT_OUT[1] + CRC unit in Bit 6: DIRECT_OUT[2] + CRC unit in Bit 7: DIRECT_OUT[3] + CRC unit in
1	Bit 0: DIRECT_OUT[0] enable Bit 1: DIRECT_OUT[1] enable Bit 2: DIRECT_OUT[2] enable Bit 3: DIRECT_OUT[3] enable
2	Bit 0: CRC unit
3	Bit 0: counter enable Bit 1: timer enable Bit 2: timeout counter enable
4	Bit 0: counter reset Bit 1: timer reset Bit 2: timeout counter reset

Instruction Format (SFCLR WAIT0SF):



Instruction Format (SFCLR WAIT1SF):



c: condition flag

- 0: Always execute instruction
- 1: Execute SFCLR instruction in case flag is '1'/clear conditionally (CSFCLR)

Example:

```
...
SFCLR WAIT0SF NO_WAIT, 0, 1
...
```

In this example, DIRECT_OUT[1] is cleared to '0'

MOVB0 (Move Bit to Bit 0)

Operation:

The selected bit of the processor source register is copied to bit 0 (LSB) of the selected processor destination register. The source register remains untouched while for the destination register just bit 0 may be toggled. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

MOVB0 <bit>, <regy>, <regz>

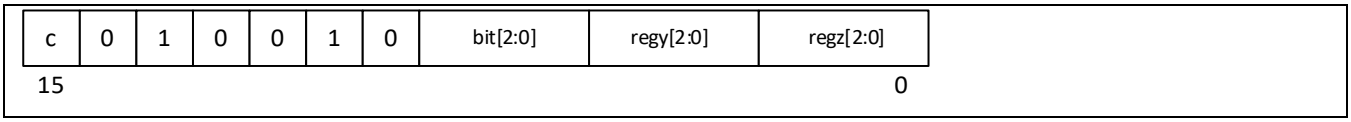
CMOVB0 <bit>, <regy>, <regz>

<bit>: bit within processor source register 0...7

<regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute MOVB0 instruction in case flag is '1'/move bit conditionally (CMOVB0)

Example:

```
...
MOVB0 2, r0, r1
...
```

In this example, bit 2 of processor register r0 overwrites bit 0 (LSB) of processor register r1.

MOVB7 (Move Bit to Bit 7)

Operation:

The selected bit of the processor source register is copied to bit 7 (MSB) of the selected processor destination register. The source register remains untouched while for the destination register just bit 7 may be toggled. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

MOVB7 <bit>, <regy>, <regz>

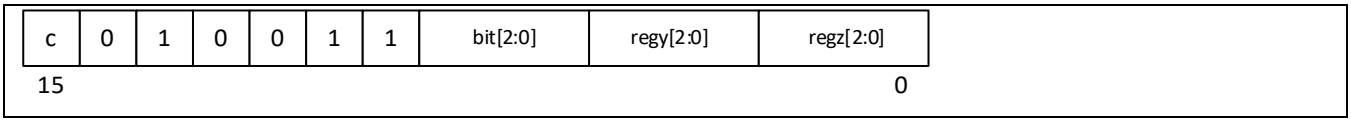
CMOVB7 <bit>, <regy>, <regz>

<bit>: bit within processor source register 0...7

<regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute MOVB7 instruction in case flag is '1'/move bit conditionally (CMOVB7)

Example:

```
...
MOVB7 2, r0, r1
...
```

In this example, bit 2 of processor register r0 overwrites bit 7 (MSB) of processor register r1.

MOVCRC (Move Bit to CRC Unit

Operation:

The selected bit of the processor source register is copied to the serial input stream of the CRC unit for CRC checksum calculation. The source register remains untouched. Any general-purpose register can be selected as source register. The execution of this instruction takes one clock cycle.

Assembler Syntax:

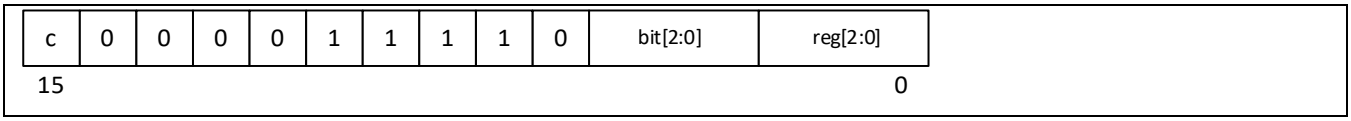
MOVCRC <bit>, <reg>

CMOVCRC <bit>, <reg>

<bit>: bit within processor source register 0...7

<reg>: processor register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute MOVCRC instruction in case flag is '1'/move bit conditionally (CMOVCRC)

Example:

```
...
LDI %0000_0100, r0 ; sync code
MOVCRC 0, r0
MOVCRC 1, r0
MOVCRC 2, r0
...
```

In this example, bit 0, bit 1, and bit 2 are copied to the serial input stream of the CRC unit for CRC checksum calculation (one after the other).

MOVNCRC (Move Inverted Bit to CRC Unit)

Operation:

The selected bit of the processor source register is inverted and then copied to the serial input stream of the CRC unit for CRC checksum calculation. The source register remains untouched. Any general-purpose register can be selected as source register. The execution of this instruction takes one clock cycle.

Assembler Syntax:

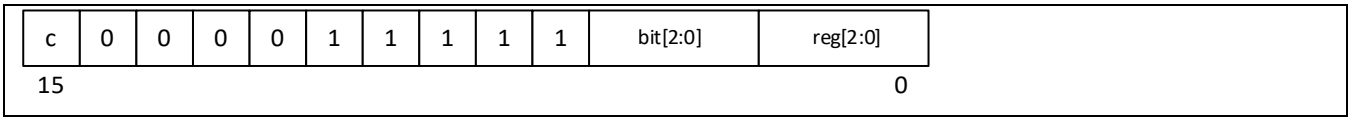
MOVNCRC <bit>, <reg>

CMOVNCRC <bit>, <reg>

<bit>: bit within processor source register 0...7

<reg>: processor register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute MOVNCRC instruction in case flag is '1'/move bit conditionally (CMOVNCRC)

Example:

```
...
MOVNCRC 0, r0
...
```

In this example, bit 0 of processor register r0 is copied to the serial input stream of the CRC unit for CRC checksum calculation.

MOVF (Move Flag to Register Bit)

Operation:

The status flag is copied to the specified bit of the destination register. The flag itself remains untouched. The destination register contents also remains untouched apart from the bit specified that may toggle. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

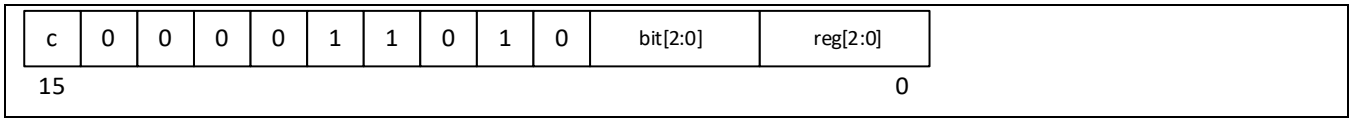
MOVF <bit>, <reg>

CMOVF <bit>, <reg>

<bit>: bit within processor destination register 0...7

<reg>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute MOVF instruction in case flag is '1'/move bit conditionally (CMOVF)

Example:

```
...
COMP EQ r0, r1
MOVF 2, r2
...
```

In this example, processor registers r0 and r1 are compared. In case contents of r0 and r1 are equal, the status flag is set. The status bit is then copied to bit 2 of the destination register r2.

MOVNF (Move Inverted Flag to Register Bit)

Operation:

The inverted value of the status flag is copied to the specified bit of the destination register. The flag itself remains untouched. The destination register contents also remains untouched apart from the bit specified that may toggle. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

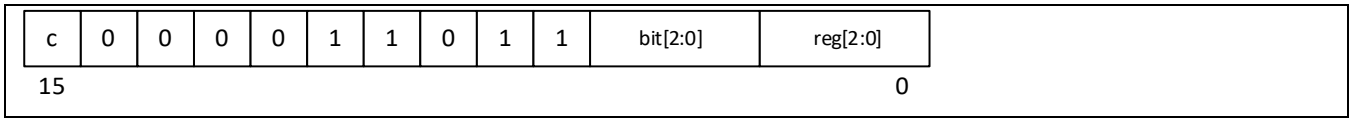
MOVNF <bit>, <reg>

CMOVNF <bit>, <reg>

<bit>: bit within processor destination register 0...7

<reg>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute MOVNF instruction in case flag is '1'/move bit conditionally (CMOVNF)

Example:

```
...
COMP EQ r0, r1
MOVNF 2, r2
...
```

In this example, processor registers r0 and r1 are compared. In case contents of r0 and r1 are equal, the status flag is set to '1'. The inverted status bit ('0' in case r0 and r1 are equal) is then copied to bit 2 of the destination register r2.

AND (Bitwise Logical And)

Operation:

A logical AND operation is performed bit-by-bit on the corresponding bits of two processor registers and the result is stored in the destination register. The source registers remain untouched while the destination register contents are overwritten with the result value. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

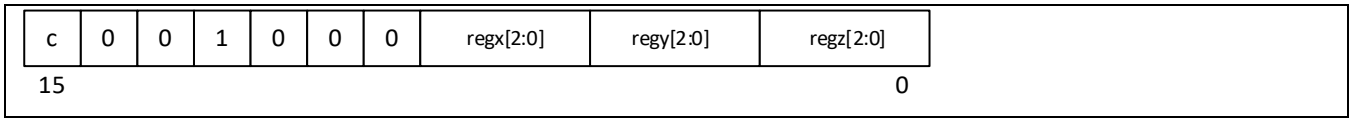
AND <regx>, <regy>, <regz>

CAND <regx>, <regy>, <regz>

<regx>, <regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute AND instruction in case flag is '1'/move bit conditionally (CAND)

Example:

```
...
LDI %1111_0000, r1
AND r0, r1, r0
...
```

In this example, the lower four bits/nibble of register r0 is set to zero.

OR (Bitwise Logical Or)

Operation:

A logical OR operation is performed bit-by-bit on the corresponding bits of two processor registers and the result is stored in the destination register. The source registers remain untouched while the destination register contents are overwritten with the result value. Any general-purpose register can be selected s source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

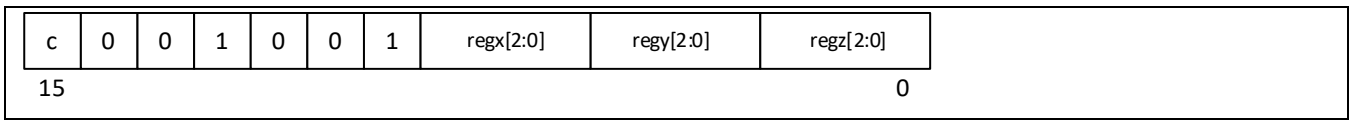
OR <regx>, <regy>, <regz>

COR <regx>, <regy>, <regz>

<regx>, <regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute OR instruction in case flag is '1'/move bit conditionally (COR)

Example:

```
...
LDI %1111_0000, r1
OR r0, r1, r0
...
```

In this example, the upper four bits/nibble of register r0 is set to one.

XOR (Bitwise Logical Exclusive Or)

Operation:

A logical exclusive OR operation is performed bit-by-bit on the corresponding bits of two processor registers and the result is stored in the destination register. The source registers remain untouched while the destination register contents are overwritten with the result value. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

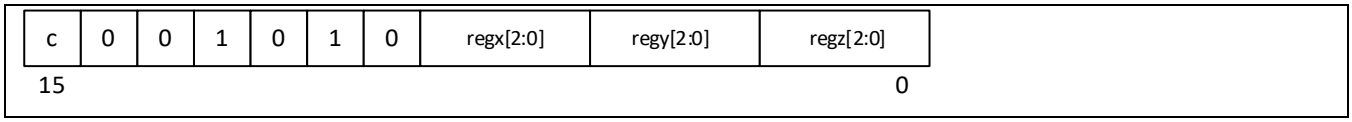
XOR <regx>, <regy>, <regz>

CXOR <regx>, <regy>, <regz>

<regx>, <regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute XOR instruction in case flag is '1'/move bit conditionally (CXOR)

Example:

```
...
LDI %1111_0000, r1
XOR r0, r1, r0
...
```

In this example, the upper four bits/nibble of register r0 is inverted.

NOT (Bitwise Inversion)

Operation:

The value of the source register is inverted, and the result stored in the destination register. The source register remains untouched while the destination register contents are overwritten with the result value. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

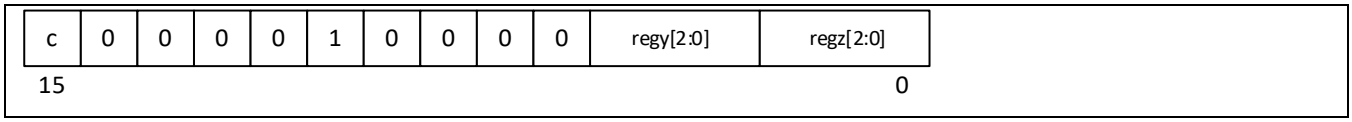
NOT <regy>, <regz>

CNOT <regy>, <regz>

<regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute NOT instruction in case flag is '1'/move bit conditionally (CNOT)

Example:

```
...
LDI $37, r0
NOT r0, r1
...
```

In this example, the value in register r0 ($37) is inverted and the result ($c8) written to destination register r1.

REV (Reverse Bit Order)

Operation:

The order of bits from the source register is reversed (bit7 → bit0, bit6 → bit1, ...) and the result stored in the destination register. The source register remains untouched while the destination register contents are overwritten with the result value. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

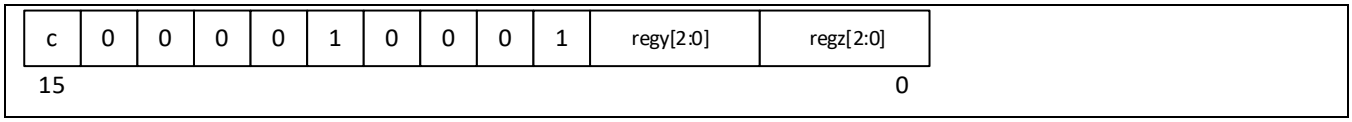
REV <regy>, <regz>

CREV <regy>, <regz>

<regy>: processor source register 0…7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute REV instruction in case flag is '1'/reverse bits conditionally (CREV)

Example:

```
...
LDI $37, r0
REV r0, r1
...
```

In this example, the bit order of the value in register r0 ($37 = %0011_0111) is reversed and the result ($EC = %1110_1100) written to destination register r1.

ADD (Add Registers)

Operation:

The contents of two registers are added (unsigned), the result is written to the destination register, and the flag is updated with the overflow/carry bit. The two source registers remain untouched while the contents of the destination register and the flag are overwritten with the result. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

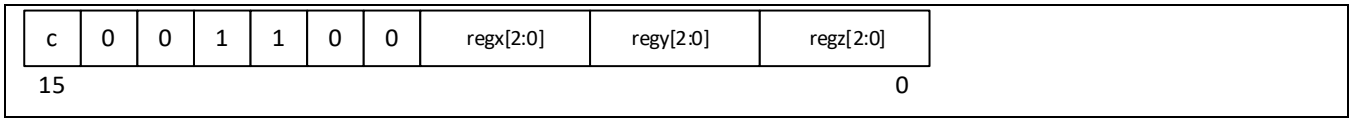
ADD <regx>, <regy>, <regz>

CADD <regx>, <regy>, <regz>

<regx>, <regy>: processor source register 0…7

<regz>: processor destination register 0…7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute ADD instruction in case flag is '1'/add conditionally (CADD)

Example:

```
...
LDI $42, r1
ADD r0, r1, r2
...
```

In this example, $42 is added to the contents of r0 and the result stored in r2.

SUB (Subtract Registers)

Operation:

The value of the register listed as second argument is subtracted from the first register value (both unsigned) and the result is written to the destination register. Standard two's compliment is used for calculation and in case of a negative result, the status flag is set - otherwise cleared. The two source registers remain untouched while the contents of the destination register and the flag are overwritten with the result. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

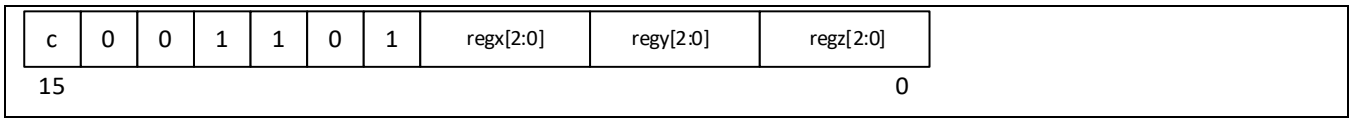
SUB <regx>, <regy>, <regz>

CSUB <regx>, <regy>, <regz>

<regx>, <regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute SUB instruction in case flag is '1'/subtract conditionally (CSUB)

Example:

```
...
LDI $42, r1
SUB r0, r1, r2
...
```

In this example, $42 is subtracted from the contents of r0 and the result stored in r2.

INC (Increment Register)

Operation:

The value of the register is incremented by one and the result is written to the destination register. In case there is an overflow, the status flag is set - otherwise cleared. The source register remains untouched while the contents of the destination register and the flag are overwritten with the result. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

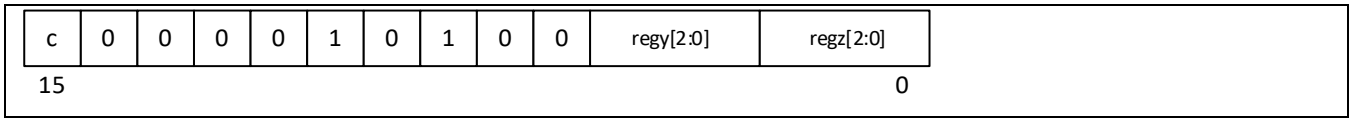
INC <regy>, <regz>

CINC <regy>, <regz>

<regy>: processor source register 0…7

<regz>: processor destination register 0…7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute INC instruction in case flag is '1'/increment conditionally (CSUB)

Example:

```
...
INC r1, r2
...
```

In this example, register r1 is incremented by one and the result written to register r2.

DEC (Decrement Register)

Operation:

The value of the register is decremented by one and the result is written to the destination register. In case there is an underflow, the status flag is set - otherwise cleared. The source register remains untouched while the contents of the destination register and the flag are overwritten with the result. Any general-purpose register can be selected as source and destination register. The execution of this instruction takes one clock cycle. Due to the write-back stage, the modified destination register can be already used as source for the next instruction during the next clock cycle.

Assembler Syntax:

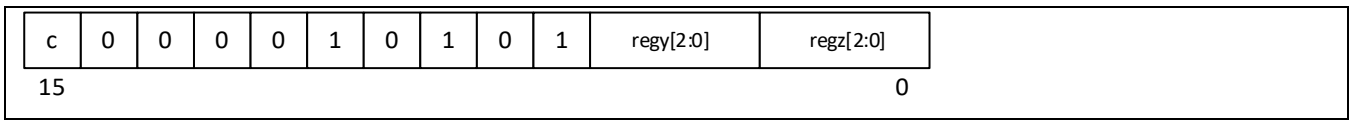
DEC <regy>, <regz>

CDEC <regy>, <regz>

<regy>: processor source register 0...7

<regz>: processor destination register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute DEC instruction in case flag is '1'/decrement conditionally (CDEC)

Example:

```
...
DEC r1, r2
...
```

In this example, register r1 is decremented by one and the result written to register r2.

COMP LT (Compare Registers for Less Than)

Operation:

The values of two registers are compared. In case the value of the first parameter register is less than the value of the second parameter register, the status flag is set - otherwise cleared. The source registers remain untouched and just the status flag is overwritten with the result. Any general-purpose register can be selected as source register. The execution of this instruction takes one clock cycle and the updated status flag is available for evaluation with the next instruction/during the next clock cycle.

Exchanging both registers allow for greater equal comparison.

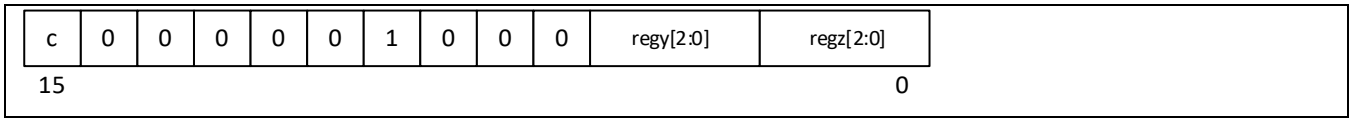
Assembler Syntax:

COMP LT <regy>, <regz>

CCOMP LT <regy>, <regz>

<regy>, <regz>: processor source registers 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute COMP instruction in case flag is '1'/compare conditionally (CCOMP)

Example:

```
LOOP:
  ...
  LDI $42, r1
  COMP LT r0, r1
  JC LOOP
  ...
```

In this example, the register contents of r0 are compared to $42. As long as r0 is less than $42, the status flag is set and the conditional jump JC back to the LOOP label is executed. As soon as r0 is equal or larger than $42, the flag is cleared/set to zero and the program jump is not executed.

COMP LE (Compare Registers for Less or Equal)

Operation:

The values of two registers are compared. In case the value of the first parameter register is less than or equal to the value of the second parameter register, the status flag is set - otherwise cleared. The source registers remain untouched and just the status flag is overwritten with the result. Any general-purpose register can be selected as source register. The execution of this instruction takes one clock cycle and the updated status flag is available for evaluation with the next instruction/during the next clock cycle.

Exchanging both registers allow for greater than comparison.

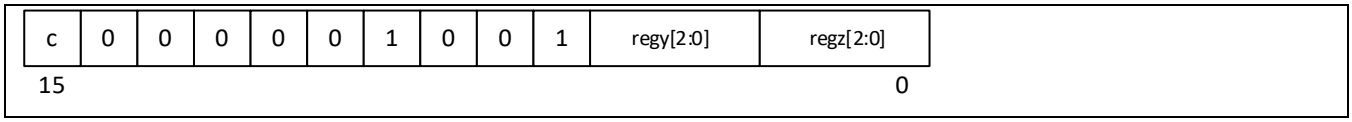
Assembler Syntax:

COMP LE <regy>, <regz>

CCOMP LE <regy>, <regz>

<regy>, <regz>: processor source registers 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute COMP instruction in case flag is '1'/compare conditionally (CCOMP)

Example:

```
LOOP:
  ...
  LDI $42, r1
  COMP LE r0, r1
  JC LOOP
  ...
```

In this example, the register contents of r0 are compared to $42. As long as r0 is less than or equal to $42, the status flag is set and the conditional jump JC back to the LOOP label is executed. As soon as r0 is greater than $42, the flag is cleared/set to zero and the program jump is not executed.

COMP EQ (Compare Registers for Equal)

Operation:

The values of two registers are compared. In case the value of the first parameter register is equal to the value of the second parameter register, the status flag is set - otherwise cleared. The source registers remains untouched and just the status flag is overwritten with the result. Any general-purpose register can be selected as source register. The execution of this instruction takes one clock cycle and the updated status flag is available for evaluation with the next instruction/during the next clock cycle.

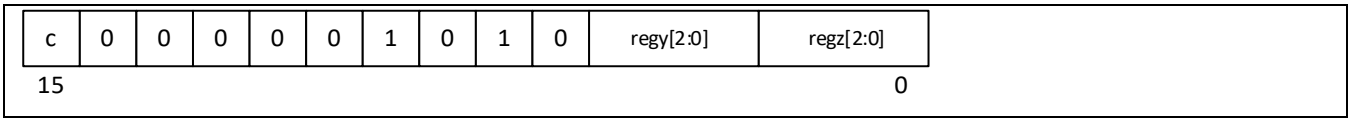
Assembler Syntax:

COMP EQ <regy>, <regz>

CCOMP EQ <regy>, <regz>

<regy>, <regz>: processor source registers 0…7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute COMP instruction in case flag is '1'/compare conditionally (CCOMP)

Example:

```
LOOP:
  ...
  LDI $42, r1
  COMP EQ r0, r1
  JC LOOP
  ...
```

In this example, the register contents of r0 are compared to $42. In case r0 is equal to $42, the status flag is set and the conditional jump JC back to the LOOP label is executed. Otherwise, the flag is cleared/set to zero and program execution continues without the jump.

COMP NE (Compare Registers for Not Equal)

Operation:

The values of two registers are compared. In case the value of the first parameter register is different from the value of the second parameter register, the status flag is set - otherwise cleared. The source registers remains untouched and just the status flag is overwritten with the result. Any general-purpose register can be selected as source register. The execution of this instruction takes one clock cycle and the updated status flag is available for evaluation with the next instruction/during the next clock cycle.

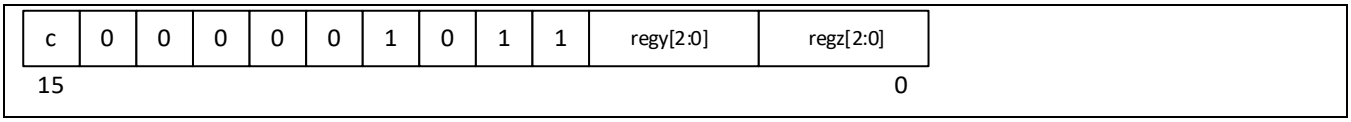
Assembler Syntax:

COMP NE <regy>, <regz>

CCOMP NE <regy>, <regz>

<regy>, <regz>: processor source registers 0…7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute COMP instruction in case flag is '1'/compare conditionally (CCOMP)

Example:

```
LOOP:
  ...
  LDI $42, r1
  COMP NE r0, r1
  JC LOOP
  ...
```

In this example, the register contents of r0 are compared to $42. As long as r0 is different from $42, the status flag is set and the conditional jump JC back to the LOOP label is executed. As soon as r0 is equal to $42, the flag is cleared/set to zero and program execution continues without the jump.

TEST0 (Test Bit for 0)

Operation:

Test specified bit of processor register. In case the bit is '0', the status flag is set to '1' - otherwise zero. Any general-purpose register can be selected as register. The contents of the register remain untouched. The execution of this instruction takes one clock cycle and the updated status flag is available for evaluation with the next instruction/during the next clock cycle.

Assembler Syntax:

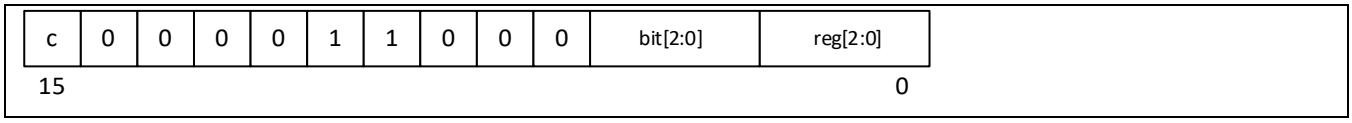
TEST0 <bit>, <reg>

CTEST0 <bit>, <reg>

<bit>: bit within processor register 0…7

<reg>: processor source register 0…7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute TEST instruction in case flag is '1'/test bit conditionally (CSET)

Example:

```
READ_LOOP:
  ...
  INC r5, r5
  TEST0 $3, r5
  JC READ_LOOP
  ...
```

In this example, the contents of register r5 is increased by one and then bit 3 of r5 tested. As long as this bit is still 0, the conditional jump to label READ_LOOP is executed and loop instruction execution repeated.

TEST1 (Test Bit for 1)

Operation:

Test specified bit of processor register. In case the bit is '1', the status flag is set to '1' - otherwise zero. Any general-purpose register can be selected as register. The contents of the register remain untouched. The execution of this instruction takes one clock cycle and the updated status flag is available for evaluation with the next instruction/during the next clock cycle.

Assembler Syntax:

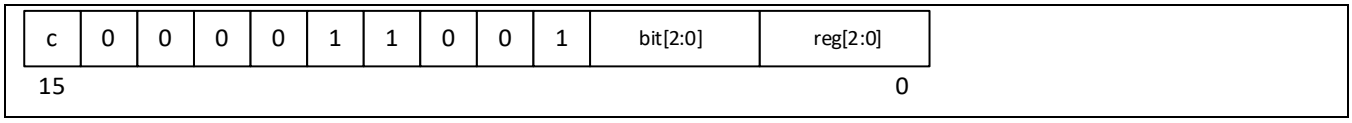
TEST1 <bit>, <reg>

CTEST1 <bit>, <reg>

<bit>: bit within processor register 0…7

<reg>: processor source register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute TEST instruction in case flag is '1'/test bit conditionally (CSET)

Example:

```
...
  TEST1 $3, r0
...
```

In this example, bit 3 of r0 is tested. In case this bit is ‘1’, the status flag is set.

SFTEST0 (Test System Register Bit for 0)

Operation:

Test specified bit of system flag register. In case the bit/flag is '0', the status flag is set to '1' - otherwise zero. The contents of the system flag register remain untouched. The execution of this instruction takes one clock cycle and the updated status flag is available for evaluation with the next instruction/during the next clock cycle.

Assembler Syntax:

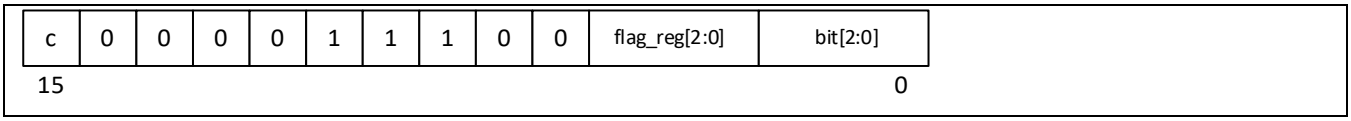
SFTEST0 <flag_reg>, <bit>

CSFTEST0 <flag_reg>, <bit>

<flag_reg>: system flag register 0...7

<bit>: bit/flag within system flag register 0...7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute SFTEST instruction in case flag is '1'/test bit/flag conditionally (CSFTEST0)

<flag_reg>	DESCRIPTION
0	Bit 0: DIRECT_IN[0] Bit 1: DIRECT_IN[1] Bit 2: DIRECT_IN[2] Bit 3: DIRECT_IN[3]
1	Bit 0 - clock generator output Bit 1 - pulse counter has reached limit value

Example:

```
...
SFTEST0 0, $1
 ...
```

In this example, the status flag is set in case DIRECT_IN[1] is currently zero.

SFTEST1 (Test System Register Bit for 1)

Operation:

Test specified bit of system flag register. In case the bit/flag is '1', the status flag is set to '1' - otherwise zero. The contents of the system flag register remain untouched. The execution of this instruction takes one clock cycle and the updated status flag is available for evaluation with the next instruction/during the next clock cycle.

Assembler Syntax:

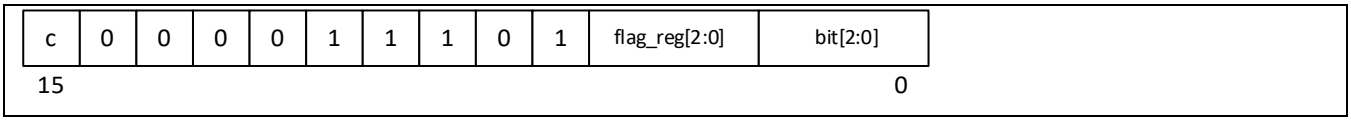
SFTEST1 <flag_reg>, <bit>

CSFTEST1 <flag_reg>, <bit>

<flag_reg>: system flag register 0...7

<bit>: bit/flag within system flag register 0…7

Instruction Format:



c: condition flag

- 0: Always execute instruction
- 1: Execute SFTEST instruction in case flag is '1'/test bit/flag conditionally (CSFTEST0)

<flag_reg>	DESCRIPTION
0	Bit 0: DIRECT_IN[0] Bit 1: DIRECT_IN[1] Bit 2: DIRECT_IN[2] Bit 3: DIRECT_IN[3]
1	Bit 0 - clock generator output Bit 1 - pulse counter has reached limit value

Example:

```
...
  SFTEST1 0, $1
...
```

In this example, the status flag is set in case DIRECT_IN[1] is currently one.

SHLO WAIT0SF/WAIT1SF (Wait and Shift Left Out)

Operation:

Wait with further program execution until specified system bit/flag (selected with parameter <wait_flag>) has changed to zero (WAIT0SF) or one (WAIT1SF). In case the specified bit/flag is already zero/one, execution of the instruction takes just one clock cycle. Otherwise, the specified bit/flag is read during each clock cycle and checked for the status change. As soon as the bit has changed, the specified processor register is shifted to the left by one, the most significant bit of the register (MSB) is shifted out to the specified system flag (<out_flag>), and program execution continues. At the same time, the system flag is shifted in as new LSB for the specified processor register. This instruction can be used to synchronize parallel-to-serial conversion and transmission of serial data to external signals, serial clock/data received, or internal timer events.

Assembler Syntax:

SHLO WAIT0SF <wait_flag>, <out_flag>, <reg>

CSHLO WAIT0SF <wait_flag>, <out_flag>, <reg>

SHLO WAIT1SF <wait_flag>, <out_flag>, <reg>

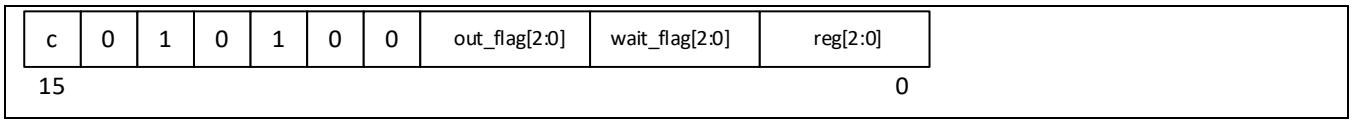
CSHLO WAIT1SF <wait_flag>, <out_flag>, <reg>

<wait_flag>: system wait flag

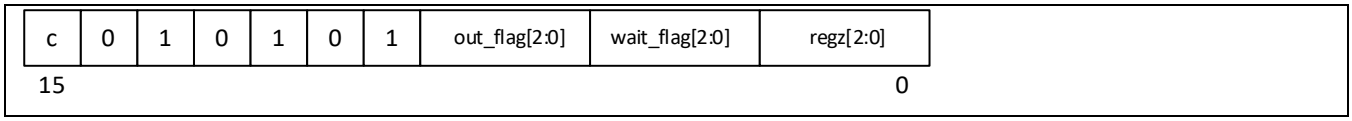
<out_flag>: output bit/flag

<reg>: processor register (0…7)

Instruction Format SHLO WAIT0SF:



Instruction Format SHLO WAIT1SF:



c: condition flag

- 0: Always execute instruction/wait
- 1: Execute instruction/shift left out in case flag is '1'/(CSHLO)

<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Overflow pulse counter
6	Overflow timer
7	No wait

<out_flag>	DESCRIPTION
0	DIRECT_OUT[0]
1	DIRECT_OUT[1]
2	DIRECT_OUT[2]
3	DIRECT_OUT[3]
4	DIRECT_OUT[0] and CRC unit in
5	DIRECT_OUT[1] and CRC unit in
6	DIRECT_OUT[2] and CRC unit in

7	DIRECT_OUT[3] and CRC unit in

Example:

```
...
WAIT_OVERFLOW_TIMER = 6
...
FLAG_OUT1 = 1
...
LDI %0101_0000, r0
REP 4, 1
SHLO WAIT1SF WAIT_OVERFLOW_TIMER, r0, FLAG_OUT1
...
```

In this example, the upper four bits of pattern %0101_0000 in register r0 are shifted out to DIRECT_OUT[1] bit-for-bit each time the system timer overflows and wraps around. The REP instruction initializes the hardware loop and makes sure the shift instruction SHLO is repeated four times. The shift instruction SHLO itself then synchronizes shifting to the system timer overflow.

SHLI WAIT0SF/WAIT1SF (Wait and Shift Left In)

Operation:

Wait with further program execution until specified system bit/flag (selected with parameter <wait_flag>) has changed to zero (WAIT0SF) or one (WAIT1SF). In case the specified bit/flag is already zero/one, execution of the instruction takes just one clock cycle. Otherwise, the specified bit/flag is read during each clock cycle and checked for the status change. As soon as the bit has changed, the specified processor register is shifted to the left by one, the least significant bit of the register (LSB) is shifted in from the specified system flag (<in_flag>), and program execution continues. The MSB of this register is dropped. This instruction can be used to synchronize serial-to-parallel conversion and capture incoming serial data to external signals, serial clock/data received, or internal timer events.

Assembler Syntax:

SHLI WAIT0SF <wait_flag>, <reg>, <in_flag>

CSHLI WAIT0SF <wait_flag>, <reg>, <in_flag>

SHLI WAIT1SF <wait_flag>, <reg>, <in_flag>

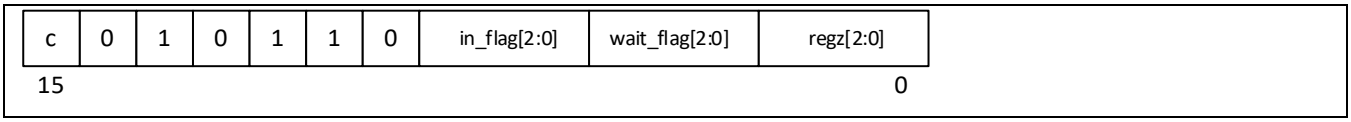
CSHLI WAIT1SF <wait_flag>, <reg>, <in_flag>

<wait_flag>: system wait flag

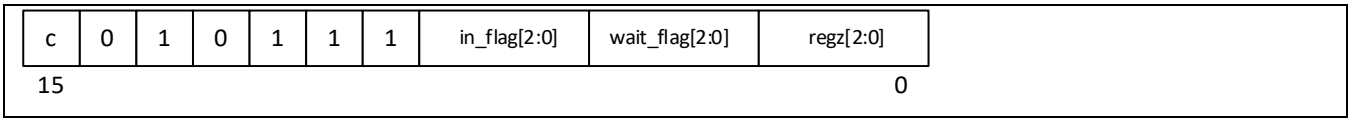
<reg>: processor register (0...7)

<in_flag>: input bit/flag

Instruction Format SHLI WAIT0SF:



Instruction Format SHLI WAIT1SF:



c: condition flag

- 0: Always execute instruction/wait
- 1: Execute instruction/shift left in in case flag is '1'/(CSHLI)

<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Overflow pulse counter
6	Overflow timer
7	No wait

<in_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	DIRECT_IN[0] and CRC unit in
5	DIRECT_IN[1] and CRC unit in
6	DIRECT_IN[2] and CRC unit in
7	DIRECT_IN[3] and CRC unit in

Example:

```
...
WAIT_OVERFLOW_TIMER = 6
...
FLAG_IN1 = 1
...

REP 8, 1
; wait for timer overflow and shift in D0..D7
SHLI WAIT1SF WAIT_OVERFLOW_TIMER, r6, FLAG_IN1
...
```

In this example, 8 bits from DIRECT_IN[1] are shifted into register r6 one after the other each time the system timer wraps around/overflows. The REP instruction initializes the hardware loop and makes sure the shift instruction SHLI is repeated eight times. The shift instruction SHLI itself then synchronizes shifting and serial-to-parallel conversion to the system timer overflow.

SHRO WAIT0SF/WAIT1SF (Wait and Shift Right Out)

Operation:

Wait with further program execution until specified system bit/flag (selected with parameter <wait_flag>) has changed to zero (WAIT0SF) or one (WAIT1SF). In case the specified bit/flag is already zero/one, execution of the instruction takes just one clock cycle. Otherwise, the specified bit/flag is read during each clock cycle and checked for the status change. As soon as the bit has changed, the specified processor register is shifted to the right by one, the least significant bit of the register (LSB) is shifted out to the specified system signal/flag (<out_flag>), and program execution continues. At the same time, the system flag is shifted in as new MSB for the specified processor register. This instruction can be used to synchronize parallel-to-serial conversion and transmission of serial data to external signals, serial clock/data received, or internal timer events.

Assembler Syntax:

SHRO WAIT0SF <wait_flag>, <reg>, <out_flag>

CSHRO WAIT0SF <wait_flag>, <reg>, <out_flag>

SHRO WAIT1SF <wait_flag>, <reg>, <out_flag>

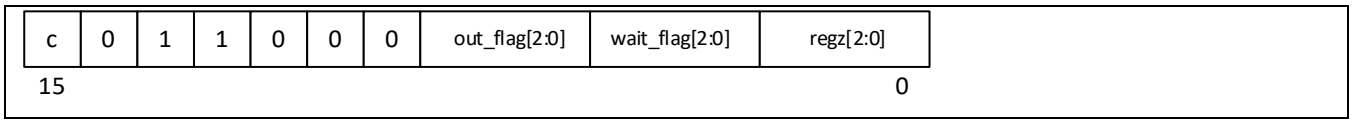
CSHRO WAIT1SF <wait_flag>, <reg>, <out_flag>

<wait_flag>: system wait flag

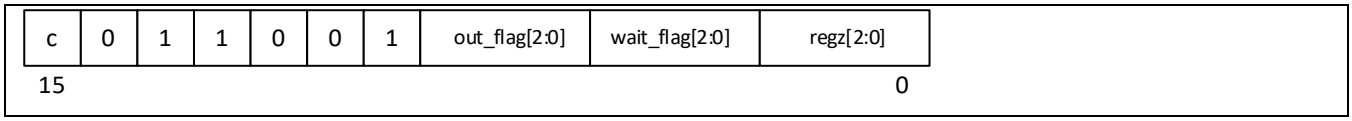
<reg>: processor register (0...7)

<out_flag>: output bit/flag

Instruction Format SHRO WAIT0SF:



Instruction Format SHRO WAIT1SF:



c: condition flag

- 0: Always execute instruction/wait
- 1: Execute instruction/shift left out in case flag is '1'/(CSHLO)

<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Overflow pulse counter
6	Overflow timer
7	No wait

<out_flag>	DESCRIPTION
0	DIRECT_OUT[0]
1	DIRECT_OUT[1]
2	DIRECT_OUT[2]
3	DIRECT_OUT[3]
4	DIRECT_OUT[0] and CRC unit in
5	DIRECT_OUT[1] and CRC unit in
6	DIRECT_OUT[2] and CRC unit in

7	DIRECT_OUT[3] and CRC unit in

Example:

```
...
WAIT_OVERFLOW_TIMER = 6
...
FLAG_OUT1 = 1
...
LDI %0000_0100, r0
REP 4, 1
SHRO WAIT1SF WAIT_OVERFLOW_TIMER, r0, FLAG_OUT1
...
```

In this example, the lower four bits of pattern %0000_0100 in register r0 are shifted out to DIRECT_OUT[1] bit-by-bit each time the system timer overflows and wraps around. The REP instruction initializes the hardware loop and makes sure the shift instruction SHRO is repeated four times. The shift instruction SHRO itself then synchronizes shifting to the system timer overflow.

SHRI WAIT0SF/WAIT1SF (Wait and Shift Right In)

Operation:

Wait with further program execution until specified system bit/flag (selected with parameter <wait_flag>) has changed to zero (WAIT0SF) or one (WAIT1SF). In case the specified bit/flag is already zero/one, execution of the instruction takes just one clock cycle. Otherwise, the specified bit/flag is read during each clock cycle and checked for the status change. As soon as the bit has changed, the specified processor register is shifted to the right by one, the most significant bit of the register (MSB) is shifted in from the specified system flag (<in_flag>), and program execution continues. The LSB of this register is dropped. This instruction can be used to synchronize serial-to-parallel conversion and capture incoming serial data to external signals, serial clock/data received, or internal timer events.

Assembler Syntax:

SHRI WAIT0SF <wait_flag>, <in_flag>, <reg>

CSHRI WAIT0SF <wait_flag>, <in_flag>, <reg>

SHRI WAIT1SF <wait_flag>, <in_flag>, <reg>

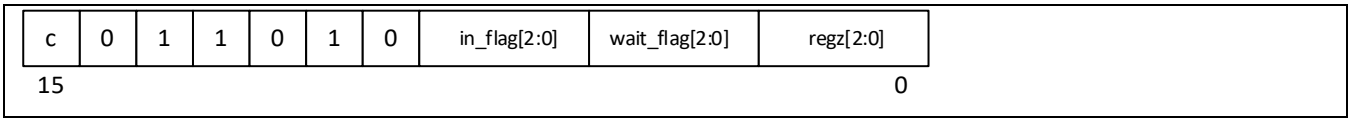
CSHRI WAIT1SF <wait_flag>, <in_flag>, <reg>

<wait_flag>: system wait flag

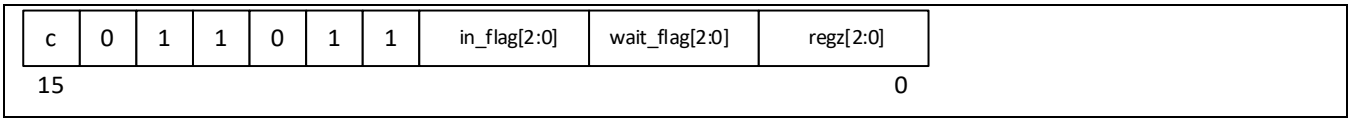
<in_flag>: input bit/flag

<reg>: processor register (0...7)

Instruction Format SHRI WAIT0SF:



Instruction Format SHRI WAIT1SF:



c: condition flag

- 0: Always execute instruction/wait
- 1: Execute instruction/shift left in in case flag is '1'/(CSHLI)

<wait_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	Overflow counter
5	Overflow pulse counter
6	Overflow timer
7	No wait

<in_flag>	DESCRIPTION
0	DIRECT_IN[0]
1	DIRECT_IN[1]
2	DIRECT_IN[2]
3	DIRECT_IN[3]
4	DIRECT_IN[0] and CRC unit in
5	DIRECT_IN[1] and CRC unit in
6	DIRECT_IN[2] and CRC unit in
7	DIRECT_IN[3] and CRC unit in

Example:

```
WAIT_OVERFLOW_TIMER = 6
FLAG_IN1_CRC = 5
...
REP 8, 1
SHRI WAIT1SF WAIT_OVERFLOW_TIMER, FLAG_IN1_CRC, r0
```

In this example, 8 bits from DIRECT_IN[1] are shifted into register r0 and the hardware CRC unit one after the other each time the system timer wraps around/overflows. The REP instruction initializes the hardware loop and makes sure the shift instruction SHRI is repeated eight times. The shift instruction SHRI itself then synchronizes shifting and serial-to-parallel conversion to the system timer overflow.

Register Map

Peripherals

ADDRESS	NAME	MSB							LSB
UART0									
0x08	UART0_BUFFER[7:0]	TX_DATA[7:0]							
0x08	UART0_BUFFER[7:0]	RX_DATA[7:0]							
0x09	UART0_BAUD_L[7:0]	BAUD_RATE_LIMIT_L[7:0]							
0x0A	UART0_BAUD_H[7:0]	–	–	–	–	BAUD_RATE_LIMIT_H[3:0]			
0x0B	UART0_CTRL[7:0]	–	RX_BUFFER_LENGTH[2:0]			RX_RESET	AUTOBAUD	NO_FILTER	x8
0x0B	UART0_STATUS[7:0]	–	–	–	TIMEOUT	AUTOBAUD_ACTIVE	TX_EMPTY	TX_FULL	RX_FULL
0x0C	UART0_TIMEOUT_L[7:0]	TIMEOUT_COUNTER_LIMIT_L[7:0]							
0x0D	UART0_TIMEOUT_H[7:0]	TIMEOUT_COUNTER_LIMIT_H[7:0]							
UART1									
0x10	UART1_BUFFER[7:0]	TX_DATA[7:0]							
0x10	UART1_BUFFER[7:0]	RX_DATA[7:0]							
0x11	UART1_BAUD_L[7:0]	BAUD_RATE_LIMIT_L[7:0]							
0x12	UART1_BAUD_H[7:0]	–	–	–	–	BAUD_RATE_LIMIT_H[3:0]			
0x13	UART1_CTRL[7:0]	–	RX_BUFFER_LENGTH[2:0]			RX_RESET	AUTOBAUD	NO_FILTER	x8

ADDRESS	NAME	MSB							LSB
0x13	UART1_STATUS[7:0]	–	–	–	TIMEOUT	AUTOBAUD_ACTIVE	TX_EMPTY	TX_FULL	RX_FULL
0x14	UART1_TIMEOUT_L[7:0]	TIMEOUT_COUNTER_LIMIT_L[7:0]							
0x15	UART1_TIMEOUT_H[7:0]	TIMEOUT_COUNTER_LIMIT_H[7:0]							
MEM									
0x18	MEM_CTRL[7:0]	–	–	–	–	ACCESS	WRITE	ADDR_MOD[1:0]	
0x19	MEM_DATA_L[7:0]	DATA_L[7:0]							
0x1A	MEM_DATA_H[7:0]	DATA_H[7:0]							
0x1B	MEM_ADDR_L[7:0]	ADDR_L[7:0]							
0x1C	MEM_ADDR_H[7:0]	–	–	–	–	–	–	ADDR_H[1:0]	
DIRECT									
0x20	DIRECT_POLARITY[7:0]	OUT3	OUT2	OUT1	OUT0	IN3	IN2	IN1	IN0
0x21	DIRECT_OUT_ALT[7:0]	OUT3_ALT[1:0]		OUT2_ALT[1:0]		OUT1_ALT[1:0]		OUT0_ALT[1:0]	
0x22	DIRECT_IN_PU[7:0]	HOME	ENC_Z	–	–	IN3	IN2	IN1	IN0
0x23	DIRECT_IN_PD[7:0]	HOME	ENC_Z	–	–	IN3	IN2	IN1	IN0
I2C									
0x28	I2C_BUFFER[7:0]	TX_DATA[7:0]							
0x28	I2C_BUFFER[7:0]	RX_DATA[7:0]							
0x29	I2C_BAUD_L[7:0]	BAUD_RATE_LIMIT_L[7:0]							
0x2A	I2C_BAUD_H[7:0]	BAUD_RATE_LIMIT_H[7:0]							
0x2B	I2C_CMD[7:0]	–	–	–	–	–	COMMAND[2:0]		
0x2B	I2C_STATUS[7:0]	–	–	–	–	–	RCV_ACK	RCV_ACK_VALUE	CMD_RDY
SPI									

ADDRESS	NAME	MSB							LSB
0x30	SPI_BUFFER0[7:0]	TX_DATA_BYTE0[7:0]							
0x30	SPI_BUFFER0[7:0]	RX_DATA_BYTE0[7:0]							
0x31	SPI_BUFFER1[7:0]	TX_DATA_BYTE1[7:0]							
0x31	SPI_BUFFER1[7:0]	RX_DATA_BYTE1[7:0]							
0x32	SPI_BUFFER2[7:0]	RX_DATA_BYTE2[7:0]							
0x32	SPI_BUFFER2[7:0]	TX_DATA_BYTE2[7:0]							
0x33	SPI_BUFFER3[7:0]	TX_DATA_BYTE3[7:0]							
0x33	SPI_BUFFER3[7:0]	RX_DATA_BYTE3[7:0]							
0x34	SPI_CTRL[7:0]	–	–	–	–	–	–	TX_RESET	TX_SKIP
0x34	SPI_STATUS[7:0]	–	–	–	–	–	TX_FULL	NO_TRANSFER	EOT
GPIO									
0x40	GPIO_IN[7:0]	–	GPIO6_IN	GPIO5_IN	GPIO4_IN	GPIO3_IN	GPIO2_IN	GPIO1_IN	GPIO0_IN
0x40	GPIO_OUT[7:0]	–	GPIO6_OUT	GPIO5_OUT	GPIO4_OUT	GPIO3_OUT	GPIO2_OUT	GPIO1_OUT	GPIO0_OUT
0x41	GPIO_POLARITY[7:0]	–	GPIO6_POLARITY	GPIO5_POLARITY	GPIO4_POLARITY	GPIO3_POLARITY	GPIO2_POLARITY	GPIO1_POLARITY	GPIO0_POLARITY
0x42	GPIO_OUT_OD[7:0]	–	–	–	–	–	GPIO2_OD	–	–
0x43	GPIO_ALT0[7:0]	GPIO3_ALT[1:0]		GPIO2_ALT[1:0]		GPIO1_ALT[1:0]		GPIO0_ALT[1:0]	
0x44	GPIO_ALT1[7:0]	–	–	GPIO6_ALT[1:0]		GPIO5_ALT[1:0]		GPIO4_ALT[1:0]	
0x45	GPIO_OUT_EN[7:0]	–	GPIO6_OUT_EN	GPIO5_OUT_EN	GPIO4_OUT_EN	GPIO3_OUT_EN	GPIO2_OUT_EN	GPIO1_OUT_EN	GPIO0_OUT_EN
0x46	GPIO_PU[7:0]	–	GPIO6_PU	GPIO5_PU	GPIO4_PU	GPIO3_PU	GPIO2_PU	GPIO1_PU	GPIO0_PU
0x47	GPIO_PD[7:0]	–	GPIO6_PD	GPIO5_PD	GPIO4_PD	GPIO3_PD	GPIO2_PD	GPIO1_PD	GPIO0_PD
0x48	SPI_PU_PD[7:0]	CSN_PD	SCLK_PD	SDO_PD	SDI_PD	CSN_PU	SCLK_PU	SDO_PU	SDI_PU
0x49	CLK_ADDR[7:0]	CLK_ADDR[7:0]							
0x4A	CLK_DATA[7:0]	CLK_DATA_WRITE[7:0]							

ADDRESS	NAME	MSB							LSB
0x4A	CLK_DATA[7:0]	CLK_DATA_READ[7:0]							
0x4C	GPIO_IN_EN[7:0]	–	–	–	–	–	–	GPIO1_IN_EN	GPIO0_IN_EN
0x4E	SILICON_REV[7:0]	SILICON_REV_DIGITAL[3:0]				SILICON_REV_ANALOG[3:0]			
TIMER									
0x60	TIMER_LIMIT0[7:0]	COUNTER_LIMIT_BYTE0[7:0]							
0x60	TIMER_COUNTER0[7:0]	COUNTER_VALUE_BYTE0[7:0]							
0x61	TIMER_LIMIT1[7:0]	COUNTER_LIMIT_BYTE1[7:0]							
0x61	TIMER_COUNTER1[7:0]	COUNTER_VALUE_BYTE1[7:0]							
0x62	TIMER_LIMIT2[7:0]	COUNTER_LIMIT_BYTE2[7:0]							
0x62	TIMER_COUNTER2[7:0]	COUNTER_VALUE_BYTE2[7:0]							
0x63	TIMER_COUNTER3[7:0]	COUNTER_VALUE_BYTE3[7:0]							
0x64	TIMER_START0[7:0]	COUNTER_START_BYTE0[7:0]							
0x64	TIMER_CAPTURE0[7:0]	COUNTER_CAPTURE_BYTE0[7:0]							
0x65	TIMER_CAPTURE1[7:0]	COUNTER_CAPTURE_BYTE1[7:0]							
0x65	TIMER_START1[7:0]	COUNTER_START_BYTE1[7:0]							
0x66	TIMER_CAPTURE2[7:0]	COUNTER_CAPTURE_BYTE2[7:0]							
0x66	TIMER_START2[7:0]	COUNTER_START_BYTE2[7:0]							
0x67	TIMER_CAPTURE3[7:0]	COUNTER_CAPTURE_BYTE3[7:0]							
0x67	TIMER_START3[7:0]	COUNTER_START_BYTE3[7:0]							
0x68	TIMER_ABZ_DIV[7:0]	ABZ_SAMPLE_DIVIDER[7:0]							
0x69	TIMER_HOME_DIV[7:0]	HOME_SAMPLE_DIVIDER[7:0]							

ADDRESS	NAME	MSB							LSB
0x6A	TIMER_AB_EVENT_CFG[7:0]	–	–	ENC_B_CONFIG[2:0]			ENC_A_CONFIG[2:0]		
0x6B	TIMER_HZ_EVENT_CFG[7:0]	–	–	HOME_CONFIG[2:0]			ENC_Z_CONFIG[2:0]		
0x6C	TIMER_CTRL[7:0]	–	CAPTURE_ONCE	CAPTURE_Z	RESET_ONCE	RESET_Z	DEC_MODE[2:0]		
0x6C	TIMER_STATUS[7:0]	–	–	–	–	–	–	OVFL	Z_EVENT
0x6D	TIMER_COMP0_0[7:0]	COMPARE0_BYTE0[7:0]							
0x6E	TIMER_COMP0_1[7:0]	COMPARE0_BYTE1[7:0]							
0x6F	TIMER_COMP0_2[7:0]	COMPARE0_BYTE2[7:0]							
0x70	TIMER_COMP0_3[7:0]	COMPARE0_BYTE3[7:0]							
0x71	TIMER_COMP1_0[7:0]	COMPARE1_BYTE0[7:0]							
0x72	TIMER_COMP1_1[7:0]	COMPARE1_BYTE1[7:0]							
0x73	TIMER_COMP1_2[7:0]	COMPARE1_BYTE2[7:0]							
0x74	TIMER_COMP1_3[7:0]	COMPARE1_BYTE3[7:0]							
0x75	TIMER_COMP_PULSE_LIMIT0[7:0]	COMP_PULSE_LIMIT_BYTE0[7:0]							
0x76	TIMER_COMP_PULSE_LIMIT1[7:0]	COMP_PULSE_LIMIT_BYTE1[7:0]							
0x77	TIMER_COMP_PULSE_CFG[7:0]	–	–	–	–	–	–	COMP1_LE	COMP0_LE
0x78	TIMER_DEC_PULSE_CFG[7:0]	DECODER_PULSE_LIMIT[7:0]							

Register Details

UART0_BUFFER (0x8)

BIT	7	6	5	4	3	2	1	0
Field	TX_DATA[7:0]							
Reset	0x0							

Access Type	Write Only

BITFIELD	BITS	DESCRIPTION
TX_DATA	7:0	Transmit fifo buffer with 8 entries

UART0_BUFFER (0x8)

BIT	7	6	5	4	3	2	1	0
Field	RX_DATA[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
RX_DATA	7:0	Receive buffer with up-to 8 entries

UART0_BAUD_L (0x9)

BIT	7	6	5	4	3	2	1	0
Field	BAUD_RATE_LIMIT_L[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
BAUD_RATE_LIMIT_L	7:0	Baud rate divider limit value - lower byte

UART0_BAUD_H (0xA)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	BAUD_RATE_LIMIT_H[3:0]			
Reset	–	–	–	–	0x0			
Access Type	–	–	–	–	Write, Read			

BITFIELD	BITS	DESCRIPTION
BAUD_RATE_LIMIT_H	3:0	Baud rate divider limit value - upper 4 bit

UART0_CTRL (0xB)

BIT	7	6	5	4	3	2	1	0
Field	–	RX_BUFFER_LENGTH[2:0]			RX_RESET	AUTOBAUD	NO_FILTER	x8
Reset	–	0x0			0x0	0x0	0x0	0x0
Access Type	–	Write Only			Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
RX_BUFFER_LENGTH	6:4	Set receive buffer length.	
RX_RESET	3	Reset receive buffer contents	
AUTOBAUD	2	Enable autobaud	
NO_FILTER	1	Disable receiver Input Filter	
x8	0	Enable x8 oversampling instead of x16 for receiver and transmitter	0x0: x16 oversampling 0x1: x8 oversampling

UART0_STATUS (0xB)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	TIMEOUT	AUTOBAUD_ACTIVE	TX_EMPTY	TX_FULL	RX_FULL
Reset	–	–	–	0x0	0x0	0x1	0x0	0x0
Access Type	–	–	–	Read Only	Read Only	Read Only	Read Only	Read Only

BITFIELD	BITS	DESCRIPTION
TIMEOUT	4	Receiver timeout counter limit value reached
AUTOBAUD_ACTIVE	3	Autobaud is active
TX_EMPTY	2	Transmit buffer is empty
TX_FULL	1	Transmit buffer is full
RX_FULL	0	Number of entries in receive buffer reached RX_BUFFER_LENGTH

UART0_TIMEOUT_L (0xC)

BIT	7	6	5	4	3	2	1	0
Field	TIMEOUT_COUNTER_LIMIT_L[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
TIMEOUT_COUNTER_LIMIT_L	7:0	Timeout counter limit value - lower 8-bit.

UART0_TIMEOUT_H (0xD)

BIT	7	6	5	4	3	2	1	0
Field	TIMEOUT_COUNTER_LIMIT_H[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
TIMEOUT_COUNTER_LIMIT_H	7:0	Timeout counter limit value - upper 8-bit.

UART1_BUFFER (0x10)

BIT	7	6	5	4	3	2	1	0
Field	TX_DATA[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
TX_DATA	7:0	Transmit fifo buffer with 8 entries

UART1_BUFFER (0x10)

BIT	7	6	5	4	3	2	1	0
Field	RX_DATA[7:0]							

Reset	0x0
Access Type	Read Only

BITFIELD	BITS	DESCRIPTION
RX_DATA	7:0	Receive buffer with up-to 8 entries

UART1_BAUD_L (0x11)

BIT	7	6	5	4	3	2	1	0
Field	BAUD_RATE_LIMIT_L[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
BAUD_RATE_LIMIT_L	7:0	Baud rate divider limit value - lower byte

UART1_BAUD_H (0x12)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	BAUD_RATE_LIMIT_H[3:0]			
Reset	–	–	–	–	0x0			
Access Type	–	–	–	–	Write, Read			

BITFIELD	BITS	DESCRIPTION
BAUD_RATE_LIMIT_H	3:0	Baud rate divider limit value - upper 4 bit

UART1_CTRL (0x13)

BIT	7	6	5	4	3	2	1	0
Field	–	RX_BUFFER_LENGTH[2:0]			RX_RESET	AUTOBAUD	NO_FILTER	x8
Reset	–	0x0			0x0	0x0	0x0	0x0
Access Type	–	Write Only			Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
RX_BUFFER_LENGTH	6:4	Set receive buffer length.	
RX_RESET	3	Reset receive buffer contents	
AUTOBAUD	2	Enable autobaud	
NO_FILTER	1	Disable Receiver Input Filter	
x8	0	Switch to x8 oversampling for receiver and transmitter	0x0: x16 oversampling 0x1: x8 oversampling

UART1_STATUS (0x13)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	TIMEOUT	AUTOBAUD_ACTIVE	TX_EMPTY	TX_FULL	RX_FULL
Reset	–	–	–	0x0	0x0	0x1	0x0	0x0
Access Type	–	–	–	Read Only	Read Only	Read Only	Read Only	Read Only

BITFIELD	BITS	DESCRIPTION
TIMEOUT	4	Receiver timeout counter limit value reached
AUTOBAUD_ACTIVE	3	Autobaud is active
TX_EMPTY	2	Transmit buffer is empty
TX_FULL	1	Transmit buffer is full
RX_FULL	0	Number of entries in receive buffer reached RX_BUFFER_LENGTH

UART1_TIMEOUT_L (0x14)

BIT	7	6	5	4	3	2	1	0
Field	TIMEOUT_COUNTER_LIMIT_L[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
TIMEOUT_COUNTER_LIMIT_L	7:0	Timeout counter limit value - lower 8-bit.

UART1_TIMEOUT_H (0x15)

BIT	7	6	5	4	3	2	1	0
Field	TIMEOUT_COUNTER_LIMIT_H[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
TIMEOUT_COUNTER_LIMIT_H	7:0	Timeout counter limit value - upper 8-bit.

MEM_CTRL (0x18)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	ACCESS	WRITE	ADDR_MOD[1:0]	
Reset	–	–	–	–	0x0	0x0	0x0	
Access Type	–	–	–	–	Write Only	Write Only	Write Only	

BITFIELD	BITS	DESCRIPTION	DECODE
ACCESS	3	Program memory read or write access	
WRITE	2	Write/not read to program memory	
ADDR_MOD	1:0	Modify address counter (after program memory access)	0x0 0x1: (Post) Increment Address Counter 0x2: (Post) Decrement Address Counter 0x3: (Post) Reset Address Counter

MEM_DATA_L (0x19)

BIT	7	6	5	4	3	2	1	0
Field	DATA_L[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
DATA_L	7:0	Program memory read/write data (lower byte)

MEM_DATA_H (0x1A)

BIT	7	6	5	4	3	2	1	0
Field	DATA_H[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
DATA_H	7:0	Program memory read/write data (upper byte)

MEM_ADDR_L (0x1B)

BIT	7	6	5	4	3	2	1	0
Field	ADDR_L[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
ADDR_L	7:0	Program memory address (lower byte)

MEM_ADDR_H (0x1C)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	–	ADDR_H[1:0]	
Reset	–	–	–	–	–	–	0x0	
Access Type	–	–	–	–	–	–	Write Only	

BITFIELD	BITS	DESCRIPTION
ADDR_H	1:0	Program memory address (upper bits)

DIRECT_POLARITY (0x20)

BIT	7	6	5	4	3	2	1	0
Field	OUT3	OUT2	OUT1	OUT0	IN3	IN2	IN1	IN0

Reset	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access Type	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
OUT3	7	DIRECT_OUT3 polarity	0x0: non-inverted 0x1: inverted
OUT2	6	DIRECT_OUT2 polarity	0x0: non-inverted 0x1: inverted
OUT1	5	DIRECT_OUT1 polarity	0x0: non-inverted 0x1: inverted
OUT0	4	DIRECT_OUT0 polarity	0x0: non-inverted 0x1: inverted
IN3	3	DIRECT_IN3 polarity	0x0: non-inverted 0x1: inverted
IN2	2	DIRECT_IN2 polarity	0x0: non-inverted 0x1: inverted
IN1	1	DIRECT_IN1 polarity	0x0: non-inverted 0x1: inverted
IN0	0	DIRECT_IN0 polarity	0x0: non-inverted 0x1: inverted

DIRECT_OUT_ALT (0x21)

BIT	7	6	5	4	3	2	1	0
Field	OUT3_ALT[1:0]		OUT2_ALT[1:0]		OUT1_ALT[1:0]		OUT0_ALT[1:0]	
Reset	0x2		0x2		0x0		0x0	
Access Type	Write Only		Write Only		Write Only		Write Only	

BITFIELD	BITS	DESCRIPTION	DECODE
OUT3_ALT	7:6		0x0: core DIRECT_OUT3 0x1: core clock output 0x2: disable output 0x3
OUT2_ALT	5:4		0x0: core DIRECT_OUT2 0x1: core clock output 0x2: disable output 0x3
OUT1_ALT	3:2		0x0: core DIRECT_OUT1 0x1: core clock output 0x2 0x3
OUT0_ALT	1:0		0x0: core DIRECT_OUT0 0x1: core clock output 0x2 0x3

DIRECT_IN_PU (0x22)

BIT	7	6	5	4	3	2	1	0
Field	HOME	ENC_Z	–	–	IN3	IN2	IN1	IN0
Reset	0x0	0x0	–	–	0x0	0x0	0x0	0x0
Access Type	Write Only	Write Only	–	–	Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
HOME	7	Enable or disable pull-up resistor for HOME input	0x0: Enable pull-up 0x1: Disable pull-up
ENC_Z	6	Enable or disable pull-up resistor for ENC_Z input	0x0: Enable pull-up 0x1: Disable pull-up
IN3	3	Enable or disable pull-up resistor for DIRECT_IN3 input	0x0: Enable pull-up 0x1: Disable pull-up
IN2	2	Enable or disable pull-up resistor for DIRECT_IN2 input	0x0: Enable pull-up 0x1: Disable pull-up
IN1	1	Enable or disable pull-up resistor for DIRECT_IN1 input	0x0: Enable pull-up 0x1: Disable pull-up
IN0	0	Enable or disable pull-up resistor for DIRECT_IN0 input	0x0: Enable pull-up 0x1: Disable pull-up

DIRECT_IN_PD (0x23)

BIT	7	6	5	4	3	2	1	0
Field	HOME	ENC_Z	–	–	IN3	IN2	IN1	IN0
Reset	0x0	0x0	–	–	0x0	0x0	0x0	0x0
Access Type	Write Only	Write Only	–	–	Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
HOME	7	Enable or disable pull-down resistor for HOME input	0x0: Disable pull-down 0x1: Enable pull-down
ENC_Z	6	Enable or disable pull-down resistor for ENC_Z input	0x0: Disable pull-down 0x1: Enable pull-down
IN3	3	Enable or disable pull-down resistor for DIRECT_IN3 input	0x0: Disable pull-down 0x1: Enable pull-down

BITFIELD	BITS	DESCRIPTION	DECODE
IN2	2	Enable or disable pull-down resistor for DIRECT_IN2 input	0x0: Disable pull-down 0x1: Enable pull-down
IN1	1	Enable or disable pull-down resistor for DIRECT_IN1 input	0x0: Disable pull-down 0x1: Enable pull-down
IN0	0	Enable or disable pull-down resistor for DIRECT_IN0 input	0x0: Disable pull-down 0x1: Enable pull-down

I2C_BUFFER (0x28)

BIT	7	6	5	4	3	2	1	0
Field	TX_DATA[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
TX_DATA	7:0	Transmit data buffer

I2C_BUFFER (0x28)

BIT	7	6	5	4	3	2	1	0
Field	RX_DATA[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
RX_DATA	7:0	Receive data buffer

I2C_BAUD_L (0x29)

BIT	7	6	5	4	3	2	1	0
Field	BAUD_RATE_LIMIT_L[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
BAUD_RATE_LIMIT_L	7:0	Baud rate divider limit value - lower byte

I2C_BAUD_H (0x2A)

BIT	7	6	5	4	3	2	1	0
Field	BAUD_RATE_LIMIT_H[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
BAUD_RATE_LIMIT_H	7:0	Baud rate divider limit value - upper byte

I2C_CMD (0x2B)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	COMMAND[2:0]		
Reset	–	–	–	–	–	0x0		
Access Type	–	–	–	–	–	Write Only		

BITFIELD	BITS	DESCRIPTION	DECODE
COMMAND	2:0	I2C comand	0x0: I2C_CMD_STOP 0x1: I2C_CMD_START_TXD_ACK 0x2: I2C_CMD_TXD_ACK 0x3: I2C_CMD_RXD_ACK 0x4: I2C_CMD_RXD_NO_ACK 0x5 0x6 0x7

I2C_STATUS (0x2B)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	RCV_ACK	RCV_ACK_VALUE	CMD_RDY
Reset	–	–	–	–	–	0x0	0x0	0x0
Access Type	–	–	–	–	–	Read Only	Read Only	Read Only

BITFIELD	BITS	DESCRIPTION	DECODE
RCV_ACK	2	Either ACK or NACK received	
RCV_ACK_VALUE	1	Value of acknowledge received - either ACK or NACK	0x0: ACK received 0x1: NACK received
CMD_RDY	0	Command processed flag	

SPI_BUFFER0 (0x30)

BIT	7	6	5	4	3	2	1	0
Field	TX_DATA_BYTE0[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
TX_DATA_BYTE0	7:0	Transmit buffer LSB [7:0]

SPI_BUFFER0 (0x30)

BIT	7	6	5	4	3	2	1	0
Field	RX_DATA_BYTE0[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
RX_DATA_BYTE0	7:0	Receive buffer LSB [7:0]

SPI_BUFFER1 (0x31)

BIT	7	6	5	4	3	2	1	0
Field	TX_DATA_BYTE1[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
TX_DATA_BYTE1	7:0	Transmit buffer [15:8]

SPI_BUFFER1 (0x31)

BIT	7	6	5	4	3	2	1	0
Field	RX_DATA_BYTE1[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
RX_DATA_BYTE1	7:0	Receive buffer [15:8]

SPI_BUFFER2 (0x32)

BIT	7	6	5	4	3	2	1	0
Field	RX_DATA_BYTE2[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
RX_DATA_BYTE2	7:0	Receive buffer [23:16]

SPI_BUFFER2 (0x32)

BIT	7	6	5	4	3	2	1	0
Field	TX_DATA_BYTE2[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
TX_DATA_BYTE2	7:0	Transmit buffer [23:16]

SPI_BUFFER3 (0x33)

BIT	7	6	5	4	3	2	1	0
Field	TX_DATA_BYTE3[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
TX_DATA_BYTE3	7:0	Transmit buffer MSB [31:24]

SPI_BUFFER3 (0x33)

BIT	7	6	5	4	3	2	1	0
Field	RX_DATA_BYTE3[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
RX_DATA_BYTE3	7:0	Receive buffer MSB [31:24]

SPI_CTRL (0x34)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	–	TX_RESET	TX_SKIP
Reset	–	–	–	–	–	–	0x0	0x0
Access Type	–	–	–	–	–	–	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
TX_RESET	1	Remove all entries from transmit buffer	
TX_SKIP	0	Drop oldest entry in transmit buffer and allow adding data instead of suppressing any write operation in case there is an overflow of the transmit buffer.	0x0: Suppress write operation in case the transmit buffer is full 0x1: Allow write operation but drop oldest value in case write buffer is full

SPI_STATUS (0x34)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	TX_FULL	NO_TRANSFER	EOT
Reset	–	–	–	–	–	0x0	0x0	0x0
Access Type	–	–	–	–	–	Read Only	Read Only	Read Only

BITFIELD	BITS	DESCRIPTION
TX_FULL	2	Transmit buffer is full
NO_TRANSFER	1	No SPI data transfer (chip select high)
EOT	0	End of SPI data transmission

GPIO_IN (0x40)

BIT	7	6	5	4	3	2	1	0
Field	–	GPIO6_IN	GPIO5_IN	GPIO4_IN	GPIO3_IN	GPIO2_IN	GPIO1_IN	GPIO0_IN
Reset	–	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access Type	–	Read Only	Read Only	Read Only	Read Only	Read Only	Read Only	Read Only

BITFIELD	BITS	DESCRIPTION
GPIO6_IN	6	GPIO6 input pin value
GPIO5_IN	5	GPIO5 input pin value
GPIO4_IN	4	GPIO4 input pin value
GPIO3_IN	3	GPIO3 input pin value
GPIO2_IN	2	GPIO2 input pin value
GPIO1_IN	1	GPIO1 input pin value
GPIO0_IN	0	GPIO0 input pin value

GPIO_OUT (0x40)

BIT	7	6	5	4	3	2	1	0
Field	–	GPIO6_OUT	GPIO5_OUT	GPIO4_OUT	GPIO3_OUT	GPIO2_OUT	GPIO1_OUT	GPIO0_OUT

Reset	–	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access Type	–	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION
GPIO6_OUT	6	GPIO6 output value
GPIO5_OUT	5	GPIO5 output value
GPIO4_OUT	4	GPIO4 output value
GPIO3_OUT	3	GPIO3 output value
GPIO2_OUT	2	GPIO2 output value
GPIO1_OUT	1	GPIO1 output value
GPIO0_OUT	0	GPIO0 output pin value

GPIO_POLARITY (0x41)

BIT	7	6	5	4	3	2	1	0
Field	–	GPIO6_POLARITY	GPIO5_POLARITY	GPIO4_POLARITY	GPIO3_POLARITY	GPIO2_POLARITY	GPIO1_POLARITY	GPIO0_POLARITY
Reset	–	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access Type	–	Write, Read	Write, Read	Write, Read	Write, Read	Write, Read	Write, Read	Write, Read

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO6_POLARITY	6	GPIO6 input/output polarity	
GPIO5_POLARITY	5	GPIO5 input/output polarity	
GPIO4_POLARITY	4	GPIO4 input/output polarity	
GPIO3_POLARITY	3	GPIO3 input/output polarity	
GPIO2_POLARITY	2	GPIO2 input/output polarity	
GPIO1_POLARITY	1	GPIO1 input/output polarity	
GPIO0_POLARITY	0	GPIO0 input/output polarity	0x0: non-inverted 0x1: inverted

GPIO_OUT_OD (0x42)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	GPIO2_OD	–	–
Reset	–	–	–	–	–	0x0	–	–
Access Type	–	–	–	–	–	Write Only	–	–

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO2_OD	2	GPIO2 output buffer type	0x0: push-pull 0x1: open-drain

GPIO_ALT0 (0x43)

BIT	7	6	5	4	3	2	1	0
Field	GPIO3_ALT[1:0]		GPIO2_ALT[1:0]		GPIO1_ALT[1:0]		GPIO0_ALT[1:0]	
Reset	0x0		0x0		0x0		0x0	
Access Type	Write, Read		Write, Read		Write, Read		Write, Read	

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO3_ALT	7:6	GPIO3 alternate function selection	0x0: GPIO3 0x1: I2C_SCL 0x2: UART1_RXD 0x3: DECODER_OUT
GPIO2_ALT	5:4	GPIO2 alternate function selection	0x0: GPIO2 0x1: I2C_SDA 0x2: UART1_TXD 0x3: HOME
GPIO1_ALT	3:2	GPIO1 alternate function selection	0x0: GPIO1 0x1: XTAL_OUT 0x2 0x3
GPIO0_ALT	1:0	GPIO0 alternate function selection	0x0: GPIO0 0x1: XTAL_IN 0x2: EXT_CLK 0x3

GPIO_ALT1 (0x44)

BIT	7	6	5	4	3	2	1	0
Field	–	–	GPIO6_ALT[1:0]		GPIO5_ALT[1:0]		GPIO4_ALT[1:0]	

Reset	–	–	0x0	0x0	0x0
Access Type	–	–	Write, Read	Write, Read	Write, Read

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO6_ALT	5:4	GPIO6 alternate function selection	0x0: GPIO6 0x1: SPI_DATA_AVAILABLE 0x2: COMPARE_OUT 0x3: DECODER_OUT
GPIO5_ALT	3:2	GPIO5 alternate function selection	0x0: GPIO5 0x1: UART0_RXD 0x2: COMPARE_OUT 0x3: DECODER_OUT
GPIO4_ALT	1:0	GPIO4 alternate function selection	0x0: GPIO4 0x1: UART0_TXD 0x2: SPI_DATA_AVAILABLE 0x3: HOME

GPIO_OUT_EN (0x45)

BIT	7	6	5	4	3	2	1	0
Field	–	GPIO6_OUT_EN	GPIO5_OUT_EN	GPIO4_OUT_EN	GPIO3_OUT_EN	GPIO2_OUT_EN	GPIO1_OUT_EN	GPIO0_OUT_EN
Reset	–	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access Type	–	Write, Read	Write, Read	Write, Read	Write, Read	Write, Read	Write, Read	Write, Read

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO6_OUT_EN	6	GPIO6 output enable	0x0: disable 0x1: enable
GPIO5_OUT_EN	5	GPIO5 output enable	0x0: disable 0x1: enable
GPIO4_OUT_EN	4	GPIO4 output enable	0x0: disable 0x1: enable
GPIO3_OUT_EN	3	GPIO3 output enable	0x0: disable 0x1: enable
GPIO2_OUT_EN	2	GPIO2 output enable	0x0: disable 0x1: enable
GPIO1_OUT_EN	1	GPIO1 output enable	0x0: disable 0x1: enable
GPIO0_OUT_EN	0	GPIO0 output enable	0x0: output disable 0x1: output enable

GPIO_PU (0x46)

BIT	7	6	5	4	3	2	1	0
Field	–	GPIO6_PU	GPIO5_PU	GPIO4_PU	GPIO3_PU	GPIO2_PU	GPIO1_PU	GPIO0_PU

Reset	–	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access Type	–	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO6_PU	6	GPIO6 internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
GPIO5_PU	5	GPIO5 internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
GPIO4_PU	4	GPIO4 internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
GPIO3_PU	3	GPIO3 internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
GPIO2_PU	2	GPIO2 internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
GPIO1_PU	1	GPIO1 internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
GPIO0_PU	0	GPIO0 internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable

GPIO_PD (0x47)

BIT	7	6	5	4	3	2	1	0
Field	–	GPIO6_PD	GPIO5_PD	GPIO4_PD	GPIO3_PD	GPIO2_PD	GPIO1_PD	GPIO0_PD
Reset	–	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access Type	–	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO6_PD	6	GPIO6 internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
GPIO5_PD	5	GPIO5 internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
GPIO4_PD	4	GPIO4 internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
GPIO3_PD	3	GPIO3 internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
GPIO2_PD	2	GPIO2 internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
GPIO1_PD	1	GPIO1 internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO0_PD	0	GPIO0 internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable

SPI_PU_PD (0x48)

BIT	7	6	5	4	3	2	1	0
Field	CSN_PD	SCLK_PD	SDO_PD	SDI_PD	CSN_PU	SCLK_PU	SDO_PU	SDI_PU
Reset	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access Type	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
CSN_PD	7	SPI chip select (CSN) internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
SCLK_PD	6	SPI serial clock (SCLK) internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
SDO_PD	5	SPI serial data out (SDO) internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
SDI_PD	4	SPI serial data in (SDI) internal pull-down resistor enable	0x0: Pull-down disable 0x1: Pull-down enable
CSN_PU	3	SPI chip select (CSN) internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
SCLK_PU	2	SPI serial clock internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
SDO_PU	1	SPI serial data out (SDO) internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable
SDI_PU	0	SPI serial data in (SDI) internal pull-up resistor disable	0x0: Pull-up enable 0x1: Pull-up disable

CLK_ADDR (0x49)

BIT	7	6	5	4	3	2	1	0
Field	CLK_ADDR[7:0]							
Reset	0x0							
Access Type	Write, Read							

BITFIELD	BITS	DESCRIPTION
CLK_ADDR	7:0	Register address for clock block access

CLK_DATA (0x4A)

BIT	7	6	5	4	3	2	1	0
Field	CLK_DATA_WRITE[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
CLK_DATA_WRITE	7:0	Register data for clock block write access. Writing to this register also triggers clock block write access.

CLK_DATA (0x4A)

BIT	7	6	5	4	3	2	1	0
Field	CLK_DATA_READ[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
CLK_DATA_READ	7:0	Register data from clock block read access

GPIO_IN_EN (0x4C)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	–	GPIO1_IN_EN	GPIO0_IN_EN
Reset	–	–	–	–	–	–	0x1	0x1
Access Type	–	–	–	–	–	–	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
GPIO1_IN_EN	1	Enable GPIO1 digital input. Disable for external XTAL connection.	0x0: Disable digital input 0x1: Enable digital input
GPIO0_IN_EN	0	Enable GPIO0 digital input. Disable for external XTAL connection.	0x0: Disable digital input 0x1: Enable digital input

SILICON_REV (0x4E)

BIT	7	6	5	4	3	2	1	0
Field	SILICON_REV_DIGITAL[3:0]				SILICON_REV_ANALOG[3:0]			
Reset	0x1				0x1			
Access Type	Read Only				Read Only			

BITFIELD	BITS	DESCRIPTION
SILICON_REV_DIGITAL	7:4	Silicon mask revision (digital part)
SILICON_REV_ANALOG	3:0	Silicon mask revision (analog part)

TIMER_LIMIT0 (0x60)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_LIMIT_BYTE0[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_LIMIT_BYTE0	7:0	Encoder counter upper wrap-around limit value LSB [7:0]

TIMER_COUNTER0 (0x60)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_VALUE_BYTE0[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_VALUE_BYTE0	7:0	Encoder counter value LSB [7:0]

TIMER_LIMIT1 (0x61)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_LIMIT_BYTE1[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_LIMIT_BYTE1	7:0	Encoder counter upper wrap-around limit value [15:8]

TIMER_COUNTER1 (0x61)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_VALUE_BYTE1[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_VALUE_BYTE1	7:0	Encoder counter value [15:8]

TIMER_LIMIT2 (0x62)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_LIMIT_BYTE2[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_LIMIT_BYTE2	7:0	Encoder counter upper wrap-around limit value [23:16]

TIMER_COUNTER2 (0x62)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_VALUE_BYTE2[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_VALUE_BYTE2	7:0	Encoder counter value [23:16]

TIMER_COUNTER3 (0x63)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_VALUE_BYTE3[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_VALUE_BYTE3	7:0	Encoder counter value MSB [31:24]

TIMER_START0 (0x64)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_START_BYTE0[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_START_BYTE0	7:0	Encoder counter start value after reset or overflow LSB [7:0]

TIMER_CAPTURE0 (0x64)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_CAPTURE_BYTE0[7:0]							

Reset	0x0
Access Type	Read Only

BITFIELD	BITS	DESCRIPTION
COUNTER_CAPTURE_BYTE0	7:0	Captured encoder counter value LSB [7:0]

TIMER_CAPTURE1 (0x65)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_CAPTURE_BYTE1[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_CAPTURE_BYTE1	7:0	Captured encoder counter value [15:8]

TIMER_START1 (0x65)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_START_BYTE1[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_START_BYTE1	7:0	Encoder counter start value after reset or overflow [15:8]

TIMER_CAPTURE2 (0x66)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_CAPTURE_BYTE2[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_CAPTURE_BYTE2	7:0	Captured encoder counter value [23:16]

TIMER_START2 (0x66)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_START_BYTE2[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_START_BYTE2	7:0	Encoder counter start value after reset or overflow [23:16]

TIMER_CAPTURE3 (0x67)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_CAPTURE_BYTE3[7:0]							
Reset	0x0							
Access Type	Read Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_CAPTURE_BYTE3	7:0	Captured encoder counter value MSB [31:24]

TIMER_START3 (0x67)

BIT	7	6	5	4	3	2	1	0
Field	COUNTER_START_BYTE3[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COUNTER_START_BYTE3	7:0	Encoder counter start value after reset or overflow MSB [31:24]

TIMER_ABZ_DIV (0x68)

BIT	7	6	5	4	3	2	1	0
Field	ABZ_SAMPLE_DIVIDER[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
ABZ_SAMPLE_DIVIDER	7:0	Sample clock divider for ENC_A/B/Z input signals

TIMER_HOME_DIV (0x69)

BIT	7	6	5	4	3	2	1	0
Field	HOME_SAMPLE_DIVIDER[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
HOME_SAMPLE_DIVIDER	7:0	Sample clock divider for HOME input signal

TIMER_AB_EVENT_CFG (0x6A)

BIT	7	6	5	4	3	2	1	0
Field	–	–	ENC_B_CONFIG[2:0]			ENC_A_CONFIG[2:0]		
Reset	–	–	0x0			0x0		
Access Type	–	–	Write Only			Write Only		

BITFIELD	BITS	DESCRIPTION	DECODE
ENC_B_CONFIG	5:3	Select encoder B channel contribution to Z event	0x0: Encoder B low 0x1: Encoder B high 0x2: Encoder B rising edge 0x3: Encoder B falling edge 0x4: Encoder B rising and falling edge 0x5: Disable event generation 0x6: Disable event generation 0x7: Ignore encoder B input
ENC_A_CONFIG	2:0	Select encoder A channel contribution to Z event	0x0: Encoder A low 0x1: Encoder A high 0x2: Encoder A rising edge 0x3: Encoder A falling edge

BITFIELD	BITS	DESCRIPTION	DECODE
			0x4: Encoder A rising and falling edge 0x5: Disable event generation 0x6: Disable event generation 0x7: Ignore encoder A input

TIMER_HZ_EVENT_CFG (0x6B)

BIT	7	6	5	4	3	2	1	0
Field	–	–	HOME_CONFIG[2:0]			ENC_Z_CONFIG[2:0]		
Reset	–	–	0x0			0x0		
Access Type	–	–	Write Only			Write Only		

BITFIELD	BITS	DESCRIPTION	DECODE
HOME_CONFIG	5:3	Select HOME input contribution to Z event	0x0: HOME low 0x1: HOME high 0x2: HOME rising edge 0x3: HOME falling edge 0x4: HOME rising and falling edge 0x5: Disable event generation 0x6: Disable event generation 0x7: Ignore HOME input
ENC_Z_CONFIG	2:0	Select encoder Z channel contribution to Z event	0x0: Encoder Z low 0x1: Encoder Z high 0x2: Encoder Z rising edge 0x3: Encoder Z falling edge 0x4: Encoder Z rising and falling edge 0x5: Disable event generation 0x6: Disable event generation 0x7: Ignore encoder Z input

TIMER_CTRL (0x6C)

BIT	7	6	5	4	3	2	1	0
Field	–	CAPTURE_ONCE	CAPTURE_Z	RESET_ONCE	RESET_Z	DEC_MODE[2:0]		
Reset	–	0x0	0x0	0x0	0x0	0x0		
Access Type	–	Write Only	Write Only	Write Only	Write Only	Write Only		

BITFIELD	BITS	DESCRIPTION	DECODE
CAPTURE_ONCE	6	Capture encoder counter value on Z event once	
CAPTURE_Z	5	Capture encoder counter value on Z event	
RESET_ONCE	4	Reset encoder counter on Z event once	
RESET_Z	3	Reset encoder counter on Z event	

BITFIELD	BITS	DESCRIPTION	DECODE
DEC_MODE	2:0	Select input decoder operation mode	0x0: x1 code 0x1: x2 code 0x2: x4 code 0x3: cw/ccw 0x4: STEP (rising edge)/DIR 0x5: STEP (both edges)/DIR 0x6 0x7

TIMER_STATUS (0x6C)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	–	OVFL	Z_EVENT
Reset	–	–	–	–	–	–	0x0	0x0
Access Type	–	–	–	–	–	–	Read Only	Read Only

BITFIELD	BITS	DESCRIPTION
OVFL	1	Encoder counter overflow flag
Z_EVENT	0	Zero channel event channel

TIMER_COMP0_0 (0x6D)

BIT	7	6	5	4	3	2	1	0
Field	COMPARE0_BYTE0[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMPARE0_BYTE0	7:0	Encoder counter compare value 0 LSB [7:0]

TIMER_COMP0_1 (0x6E)

BIT	7	6	5	4	3	2	1	0
Field	COMPARE0_BYTE1[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMPARE0_BYTE1	7:0	Encoder counter compare value 0 [15:8]

TIMER_COMP0_2 (0x6F)

BIT	7	6	5	4	3	2	1	0
Field	COMPARE0_BYTE2[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMPARE0_BYTE2	7:0	Encoder counter compare value 0 [23:16]

TIMER_COMP0_3 (0x70)

BIT	7	6	5	4	3	2	1	0
Field	COMPARE0_BYTE3[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMPARE0_BYTE3	7:0	Encoder counter compare value 0 MSB [31:24]

TIMER_COMP1_0 (0x71)

BIT	7	6	5	4	3	2	1	0
Field	COMPARE1_BYTE0[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMPARE1_BYTE0	7:0	Encoder counter compare value 1 LSB [7:0]

TIMER_COMP1_1 (0x72)

BIT	7	6	5	4	3	2	1	0
Field	COMPARE1_BYTE1[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMPARE1_BYTE1	7:0	Encoder counter compare value 1 [15:8]

TIMER_COMP1_2 (0x73)

BIT	7	6	5	4	3	2	1	0
Field	COMPARE1_BYTE2[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMPARE1_BYTE2	7:0	Encoder counter compare value 1 [23:16]

TIMER_COMP1_3 (0x74)

BIT	7	6	5	4	3	2	1	0
Field	COMPARE1_BYTE3[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMPARE1_BYTE3	7:0	Encoder counter compare value 1 [31:24]

TIMER_COMP_PULSE_LIMIT0 (0x75)

BIT	7	6	5	4	3	2	1	0
Field	COMP_PULSE_LIMIT_BYTE0[7:0]							

Reset	0x0
Access Type	Write Only

BITFIELD	BITS	DESCRIPTION
COMP_PULSE_LIMIT_BYTE0	7:0	Length of COMPARE_OUT signal in number of system clock cycles + 1 (lower byte)

TIMER_COMP_PULSE_LIMIT1 (0x76)

BIT	7	6	5	4	3	2	1	0
Field	COMP_PULSE_LIMIT_BYTE1[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
COMP_PULSE_LIMIT_BYTE1	7:0	Length of COMPARE_OUT signal in number of system clock cycles + 1 (upper byte)

TIMER_COMP_PULSE_CFG (0x77)

BIT	7	6	5	4	3	2	1	0
Field	–	–	–	–	–	–	COMP1_LE	COMP0_LE
Reset	–	–	–	–	–	–	0x0	0x0
Access Type	–	–	–	–	–	–	Write Only	Write Only

BITFIELD	BITS	DESCRIPTION	DECODE
COMP1_LE	1	Select compare operation between compare1 and encoder counter value register. In case the compare operations with compare0 and compare1 registers both get valid, the output signal COMPARE_OUT is activated.	0x0: Compare1 greater than 0x1: Compare1 less or equal
COMP0_LE	0	Select compare operation between compare0 and encoder counter value register. In case the compare operations with compare0 and compare1 registers both get valid, the output signal COMPARE_OUT is activated.	0x0: Compare0 value greater than 0x1: Compare0 less or equal

TIMER_DEC_PULSE_CFG (0x78)

BIT	7	6	5	4	3	2	1	0
Field	DECODER_PULSE_LIMIT[7:0]							
Reset	0x0							
Access Type	Write Only							

BITFIELD	BITS	DESCRIPTION
DECODER_PULSE_LIMIT	7:0	Length of DECODER_OUT signal in number of system clock cycles + 1

Typical Application Circuits

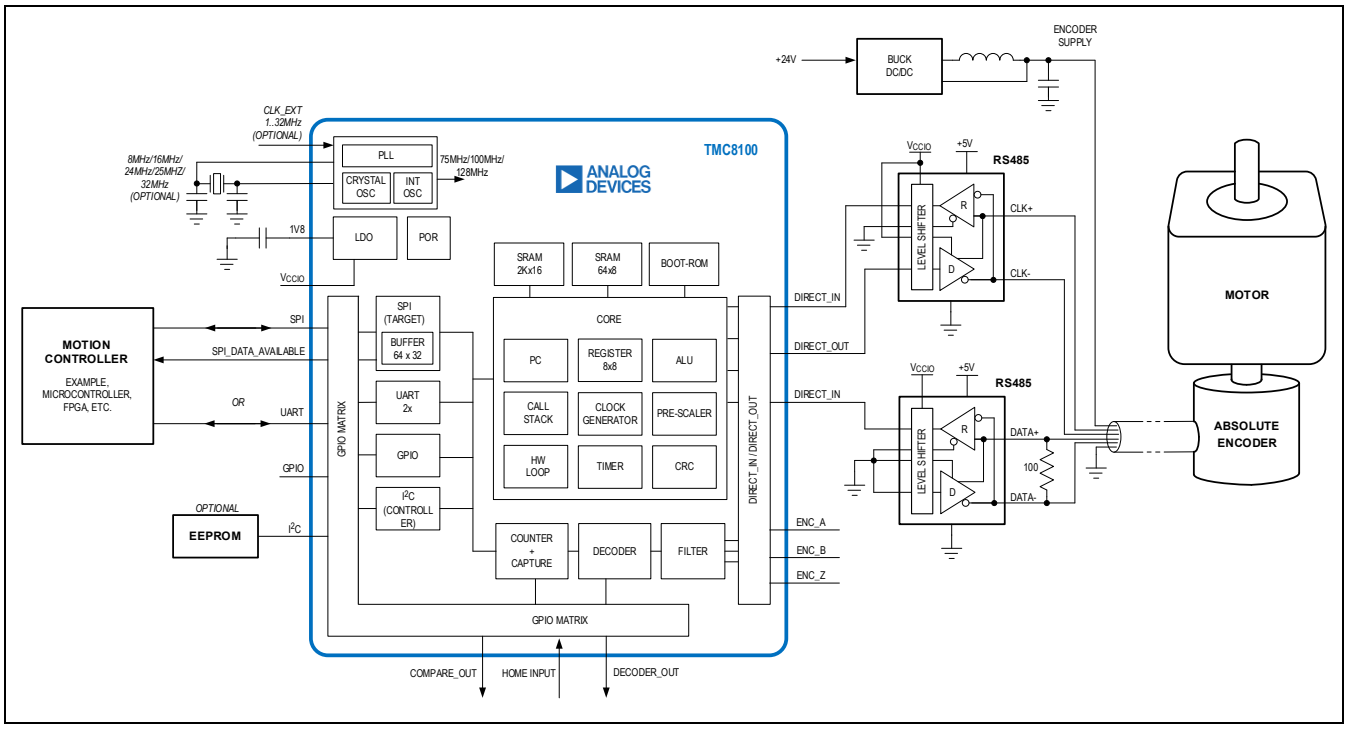


Figure 17. SSI Encoder Application Circuit Example

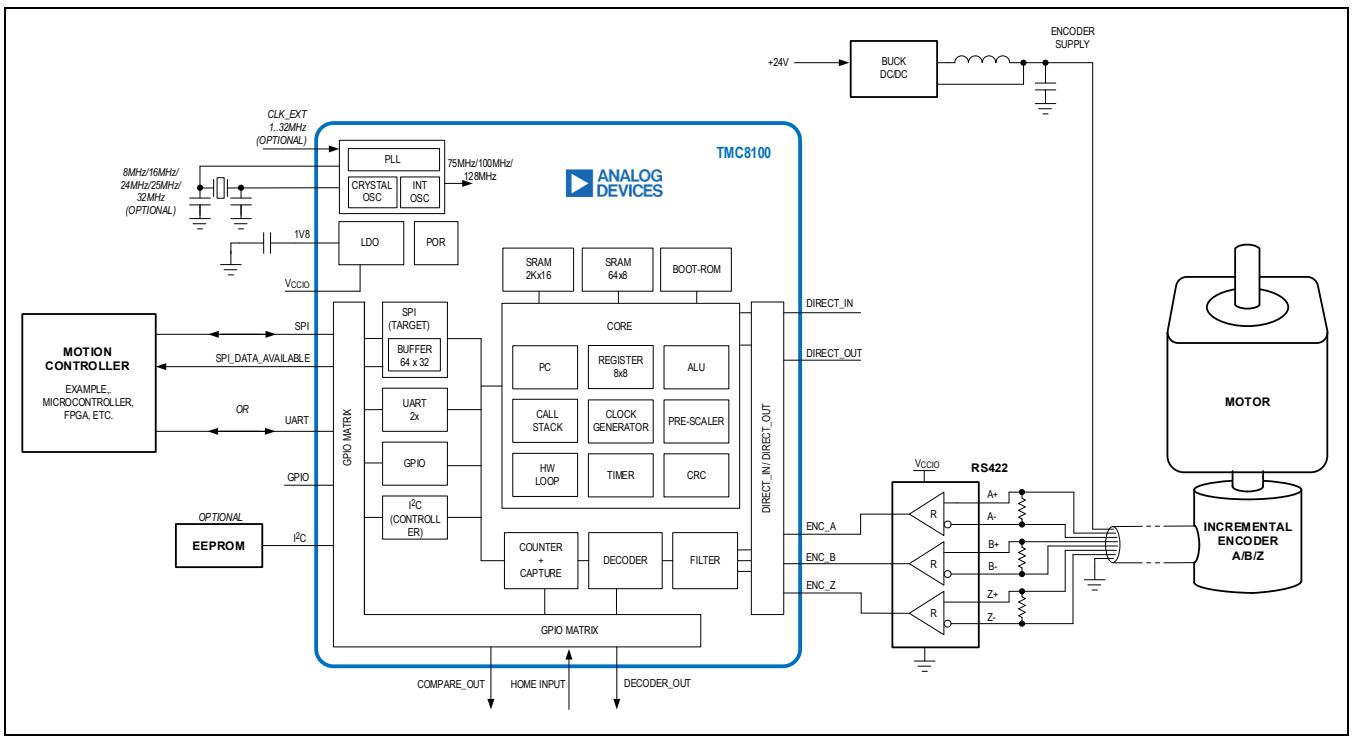


Figure 18. A/B/Z Incremental Encoder Application Example

Ordering Information

PART NUMBER	TEMP RANGE	PIN-PACKAGE
TMC8100ATG+	-40°C to +125°C	24 TQFN 4mm x 4mm

+Denotes lead(Pb)-free/RoHS-compliance.

#Denotes a RoHS-compliant device that may include lead(Pb) that is exempt under the RoHS requirements.

T = Tape and reel.

Y = Side-wettable package.

Revision History

REVISION NUMBER	REVISION DATE	DESCRIPTION
0	04/24	Release for market intro