PoKeys PoIL processor manual

Version: 21/1/2017

Please read the following notes

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice.
2. PoLabs does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of PoLabs products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of PoLabs or others. PoLabs claims the copyright of, and retains the rights to, all material (software, documents, etc.) contained in this release. You may copy and distribute the entire release in its original state, but must not copy individual items within the release other than for backup purposes.
3. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of the products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. PoLabs assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
4. PoLabs has used reasonable care in preparing the information included in this document, but PoLabs does not warrant that such information is error free. PoLabs assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
5. PoLabs devices may be used in equipment that does not impose a threat to human life in case of the malfunctioning, such as: computer interfaces, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment, and industrial robots.
6. Measures such as fail-safe function and redundant design should be taken to ensure reliability and safety when PoLabs devices are used for or in connection with equipment that requires higher reliability, for example: traffic control systems, anti-disaster systems, anticrime systems, safety equipment, medical equipment not specifically designed for life support, and other similar applications.
7. PoLabs devices shall not be used for or in connection with equipment that requires an extremely high level of reliability and safety, as for example: aircraft systems, aerospace equipment, nuclear reactor control systems, medical equipment or systems for life support (e.g. artificial life support devices or systems), and any other applications or purposes that pose a direct threat to human life.
8. You should use the PoLabs products described in this document within the range specified by PoLabs, especially with respect to the maximum rating, operating supply voltage range and other product characteristics. PoLabs shall have no liability for malfunctions or damages arising out of the use of PoLabs products beyond such specified ranges.
9. Although PoLabs endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, PoLabs products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a PoLabs product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures.
10. Usage: the software in this release is for use only with PoLabs products or with data collected using PoLabs products.
11. Fitness for purpose: no two applications are the same, so PoLabs cannot guarantee that its equipment or software is suitable for a given application. It is therefore the user's responsibility to ensure that the product is suitable for the user's application.
12. Viruses: this software was continuously monitored for viruses during production, however the user is responsible for virus checking the software once it is installed.
13. Upgrades: we provide upgrades, free of charge, from our web site at www.poscope.com. We reserve the right to charge for updates or replacements sent out on physical media.
14. Please contact a PoLabs support for details as to environmental matters such as the environmental compatibility of each PoLabs product. Please use PoLabs products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. PoLabs assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
15. Please contact a PoLabs support at support@poscope.com if you have any questions regarding the information contained in this document or PoLabs products, or if you have any other inquiries.
16. The licensee agrees to allow access to this software only to persons who have been informed of and agree to abide by these conditions.
17. Trademarks: Windows is a registered trademark of Microsoft Corporation. PoIL, PoKeys, PoKeys55, PoKeys56U, PoKeys56E, PoScope, PoLabs and others are internationally registered trademarks.

Table of contents

PoKeys PoIL processor

PoKeys PoIL processor is a 16/32-bit software processor, which executes PoIL code, and is used in supported PoKeys devices. The processor employs Harvard architecture with reduced instruction set (RISC).

The processor has one 32-bit working register (named W) and advanced set of assembly commands that allow high compatibility with higher programming languages (oriented towards the languages defined by IEC 61131 standard). The PoIL processor targets application where the advanced PoKeys input-output interface card can be equipped with additional logic in order to allow the autonomous operation without a host PC. With the rich set of features, PoIL processor transforms a PoKeys device into a low-cost Programmable Logic Controller (PLC) with easy to learn syntax.

Architectural overview

The PoIL processor features a number of architectural properties commonly found in RISC microprocessors. As mentioned before, PoIL processor employs Harvard architecture in which code and data memory are accessed separately, giving the developer a better overview of the memory. The separated code and data memories both use 16-bit (byte-) addressing (both in LSB first configuration). Special regions in the data memory are assigned to specific functions, as are defined in Table 1. Similarly, special regions in code memory are assigned to specific system functions, as defined in Table 2. Additionally, code memory is 16-bit aligned (instructions must reside on even addresses only).

The PoIL processor supports four different addressing modes – direct, indirect, stack and literal – as is described later on. The majority of the commands in the instruction set can use any of the addressing mode, thus allowing the combination of fetch+execute or execute+store in one command. The instructions are 16-bit wide with optional additional operand's value or operand's address. The instruction set consists of commands for managing fetching and storing data, logical operations (AND, OR, XOR), bit operations (bit set, bit clear, bit toggle), arithmetic operations (addition, subtraction, multiplication, division, modulus, bit-shifting left and right), compare operations (if greater, if greater or equal, if equal, if not equal, if lower than or equal, if lower, bit test) and execution manipulation commands (jump, jump if true, jump if false, call, return, exit task). The 32-bit working register is used for different arithmetic and logic operations and is not directly addressable (just its read-only value is accessible at 0xFF0C).

The result of the executed instruction (mostly the compare operations) is saved in the bit 7 (L) of the status register.

Status register (8-bit)

7	6	5	4	3	2	1	0
							Logical result

Bit	Symbol	Description
7:1	-	Reserved
0	L	Logical operation result

Stack

PoIL processor uses two separated stacks – one is used for the operand storage (data stack), while the other is used for the address storage (function stack). Data stack contains 32-bit entries, while the function stack contains 16-bit entries.

The exact size of each stack is dependent on the processor implementation.

Device	Function stack size	Data stack size
PoKeys56 series	64	32
PoKeys57 series	64	32

Program counter

Program counter (PC) is a processor register that indicates where the execution is in the current program sequence. The program counter value is increased by 2, 4 or 6 after each instruction cycle, unless the instruction changes the counter's value. The exact amount by which the PC counter is incremented depends on the presence of the operand and its format.

For a 'JMP', 'JMPT', 'JMPF' instructions, the 16-bit operand value is copied to the program counter and the execution is effectively diverted to the address specified by the operand.

For a 'CALL' instruction, the current PC value is pushed onto the function stack and the 16-bit operand value is copied to the program counter.

For a 'RETURN' instruction, the value is popped from the function stack and copied to the PC value.

Multitasking

The PoIL processor supports priority-based pre-emptive scheduler that switches between two (or more on later versions) tasks. Task 0 has the lowest priority and is enabled by default. Other tasks are periodic tasks that have a fixed time-period between executions. Tasks 1 and on must be enabled first using the task configuration system function.

Task switching is done at 1 ms intervals or on task exit events.

Addressing modes

Addressing mode	Opcode format	Description
Direct	address	Direct addressing mode uses the value stored at the address, specified in the operand
Indirect	[address]	Indirect addressing mode uses the value stored at the address that is stored at the address, specified in the operand. This mode can be described as using pointers to access the data.
Stack	S	This operation uses stack to exchange the data. No data type is specified, as this mode always uses 32-bit data.
Literal	L[value]	The operand's value is specified as a constant value. For bit data types, the value is ignored to save memory space and bit inversion ! sign must be used to load bit 1.

Data types

The PoIL processor directly supports 4 data types. The data type defines the type of data that gets manipulated in memory.

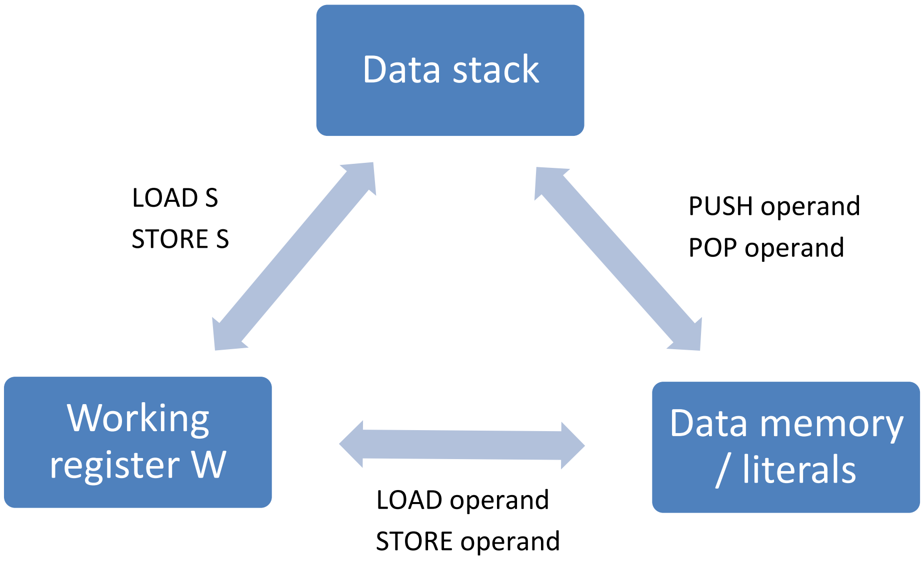
Data type	Symbol	Use
Bit (Boolean)	b	LOAD b!1234.3 # Load inverted bit 3 from 1234 to W ADD b20.5 # Add bit 5 from address 20 to W LOAD bL[0].0 # Load bit value 0 to W LOAD b!L[0].0 # Load bit value 1 to W
Byte	B	LOAD Bh10 # Load 0x10 (16) to W ADD B[50] # Add the byte value from the register with the address that is saved in register 50 to W
Word	W	ADD WL[1500] # Load constant value of 1500 to W
Double word	D	PUSH DL[hFF] # Push 0xFF (15) to stack

Data stack operations

There are 4 basic stack operations

Operation	Description	Example
PUSH operand	Push the operand's value to stack Equivalent of (with the W unaffected): LOAD operand STORE S	PUSH DL[10] # Push value of 10 to stack PUSH W5 # Push the 16-bit value of register at the address of 5 to stack
STORE S	Push the value of W to stack	STORE S # Store the W to stack
POP operand	Pop the value from stack and save it to operand Equivalent of (with the W unaffected): LOAD S STORE operand	POP D0 # Pop the 32-bit value from stack and save it to the register at 0
LOAD S	Pop the value of W from stack	LOAD S # Retrieve the W from stack

The following figure illustrates how data in data stack, working register and in data memory can be manipulated.


Data stack
LOAD S
STORE S
PUSH operand
POP operand
Working register W
Data memory / literals
LOAD operand
STORE operand

Memory organization

The PoIL processor memories are organized into code and data memory. Both memories are addressable by bytes. Also, both program counter and data addressing pointer are 16-bit, thus capable of addressing of up to 65536 bytes. However, only specific areas of the memory may be accessible to the processor, as is specified below.

Code memory with system functions calls

Code memory address	Function
0x0000 - 0x0FFF	Code memory (PoKeys56 series)
0x0000 - 0x7FFF	Code memory (PoKeys57 series)
0xFF00 - 0xFFFF	Special system calls
0xFF00	Task configuration function (parameters on stack) - Param 1: task ID - Param 2: task period (set to 0 to disable task) - Param 3: task start address
0xFF02	Encoder configuration - Param 1: encoder ID - Param 2: pin A - Param 3: pin B - Param 4: option
0xFF03 (deprecated in PoKeys57 series - see 0xFF17 for new function)	Configure Pulse engine axes speed/accelerations (float values) - Param 1: x max speed / ms - Param 2: x max acceleration / ms^2 - Param 3: x max deceleration / ms^2 - Param 4: y max speed / ms - Param 5: y max acceleration / ms^2 - Param 6: y max deceleration / ms^2 - Param 7: z max speed / ms - Param 8: z max acceleration / ms^2 - Param 9: z max deceleration / ms^2 - Param 10: reserved - 0

0xFF04 (deprecated in PoKeys57 series - see 0xFF17 for new function)	Pulse engine axes switches/directions configuration
	- Param 1: bit encoded switches configuration for x axis
	    o Bit 0: x home switch
	    o Bit 1: x limit- switch
	    o Bit 2: x limit+ switch
	    o Bit 3: home/limit combined switch
	    o Bit 4: invert x axis
	    o Bit 5: invert x homing direction
	- Param 2: bit encoded switches configuration for y axis
	- Param 3: bit encoded switches configuration for z axis
	- Param 4: number of axes - set to 3

0xFF05	Pulse engine - command a move
	- Param 1: x reference value
	- Param 2: y reference value
	- Param 3: z reference value
	- Param 4: bit encoded commands for position/speed
	    o Bit 0: go in position mode
	    o Bit 1: go in speed mode
	- Param 5: number of axes - set to 3

0xFF06	Put variable to LCD
	- Param 1: variable value
	- Param 2: format and position
	  o Bit 0: always show sign
	  o Bits 1-4: display digits count
	  o Bits 5-8: zero-padded digits count
	  o Bits 9-11: decimals count (float data)
	  o Bits 12-13: LCD row
	  o Bits 14-18: LCD column
	- Param 3: optional: multiplier (float)
	- Param 4: variable type - integer (0-9, equals to decimal places), float(10)

0xFF07	Initialize LCD
	- Param 1:
	  o Bits 0-7: LCD configuration (0 primary/1 secondary)
	  o Bits 8-15: LCD rows
	  o Bits 16-23: LCD columns

0xFF08	Configure counter on digital input pin
	- Param 1:
	  o Bits 0-7: counting pin
	  o Bits 8-15: direction pin + 1 (set to 0 to disable)
	  o Bit 16: count rising edges
	  o Bit 17: count falling edges

0xFF09	Configure multi-function analog input pin (on selected devices only)
	- Param 1:
	  o Bits 0-7: analog pin ID
	  o Bits 8-10: analog function
	  o Bits 11-13: conversion resolution
	  o Bits 14-18: additional parameters

0xFF0A	Configure and operate 1-wire devices
	- Param 1: Operation
	  o 0x00 - disable 1-wire
	  o 0x01 - enable 1-wire
	  o 0x10 - Start write and read
	  o 0x11 - Read status/result and data
	- Param 2: Data pointer (n x 8-bit)
	- Param 3: Data count to write (n)
	- Param 4: Data count to read
	Returns:
	- Param 1: Status

0xFF0B	Configure and operate I2C devices
	- Param 1: Operation
	  o 0x10 - Start write
	  o 0x11 - Get write result
	  o 0x20 - Start read
	  o 0x21 - Get read result
	- Param 2: Data pointer (n x 8-bit) - for operations 0x10, 0x21
	- Param 3: Device address
	- Param 4: Data count (bits 0-7: data count to write or read, bits 8-15: data count to read if combined transaction is required - bits 0-7 in this case equal to count of bytes to

	write) - up to 30 bytes per transaction Returns: - Param 1: Status (0 - error, 1 - OK, 0x10 - pending)
0xFF0C	Configure and operate SPI bus - Param 1: Operation o 0x10: Initialize (param 2 for prescaler and param 3 for format) o 0x20: Transfer data (number of data bytes in param 2) o 0x30: Get result - Param 2: Prescaler configuration (0x10) or number of bytes (0x20) - Param 3: Format (0x10) or data pointer (n x 8-bit) - Param 4: Pin select pin index Returns: - Param 1: Status (0 - ready, 1 - busy, 10 - error)
0xFF10	Timer block functionality - Param 1: Timer type (0-pulse, 10-ON, 20-OFF) - Param 2: Timer period in ms - Param 3: Previous input - Param 4: Current input - Param 5: Previous output - Param 6: 32-bit temporary variable Returns: - Param 1: new output value - Param 2: timer time (ET) value in ms - Param 3: 32-bit temporary variable
0xFF11	Counter block functionality - Param 1: counter type (0 – up, 1 – down, 2 – up/down) - Param 2: counter PV value (preset value) - Param 3: counter CV value (current) - Param 4: input 1 - Param 5: input 1 previous value - Param 6: input 2 - Param 7: input 2 previous value - Param 8: reset - Param 9: load PV to CV Returns: - Param 1: output UP - Param 2: output DOWN - Param 3: CV value
0xFF12	Look-up table functionality - Params 1-10: look-up table entries (LSB first) - Param next: look-up table index (0 to 39) in LSB byte, number of entries in MSB byte Returns: - Param 1: look-up table data (8-bit)
0xFF13	Time-scheduling functionality - Params 1-10: schedule entries (1 to 10 entries) o Bits 0-5: onMinute o Bits 6-10: onHour

	○ Bits 11-16: offMinute ○ Bits 17-21: offHour ○ Bits 22-28: bit-encoded week days ○ Bits 29-31: unused - Param next: number of schedules (1-10) Returns: - Param 1: On/Off value
0xFF14	PID controller - Param 1: PV - Param 2: SP - Param 2: Pointer to parameters memory (7x 32-bit) • Parameter 1: Kp • Parameter 2: Ki • Parameter 3: Kd • Parameter 4: Kf1 • Parameter 5: Kf2 • Parameter 6: low limit • Parameter 7: high limit - Param 3: Pointer to PID memory (2x 32-bit)
0xFF15	Look-up table functionality - 32-bit - Params 1-10: look-up table entries (LSB first) - Param next: look-up table index (0 to 9) in LSB byte, number of entries in MSB byte Returns: - Param 1: look-up table data (32-bit)
0xFF16	Additional functions (PoKeys57 series only) - 0-10 parameters - Function selection Returns: - 0-10 parameters Float number type is saved in 32-bit integer number memory slot. The float number type should only be used with functions that accept or return this type. Functions: 0x1000 - Calculate power of number - params: exponent, base (float) - returns: base ^ exponent (float) 0x1010 - Operation over one float - no checking - params: number (float), opNr - returns: operation over number (float) - opNr: 0=exp, 1=sin, 2=cos, 3=tan, 4=asin, 5=acos, 6=fabs 0x1011 - Operation over one float - zero checking - params: number (float), opNr - returns: operation over number (float) - opNr: 0=log, 1=log10, 2=atan 0x1020 - Operation over two floats

	- params: numbers y, x (float), opNr - returns: operation over x and y (float) - opNr: 0=pow(x,y), 1=atan2(x,y) 0x1100 - Sum two floats - params: numbers y, x (float) - returns: x+y (float) 0x1101 - Subtract two floats - params: numbers y, x (float) - returns: x-y (float) 0x1102 - Multiply two floats - params: numbers y, x (float) - returns: x*y (float) 0x1103 - Divide two floats - params: numbers y, x (float) - returns: x/y (float) 0x2000 - Convert from float to integer - params: x (float) - returns: x (int) 0x2001 - Convert from integer to float - params: x (int) - returns: x (float) 0x3000 - Convert float to scientific notation - params: x (float), l (int) - returns: a, b (int), where x=a*10^b, a is multiplied by 10^l

0xFF17	Pulse engine axis configuration function **(PoKeys57 series only)**
	- Param 1: Axis index (0 to 7)
	- Param 2: Configuration selection
	  ○ 0 - set maximum speed (steps/s)
	  ○ 1 - set maximum acceleration (steps/s^2)
	  ○ 2 - set maximum deceleration (steps/s^2)
	  ○ 10 - axis options (see 0x85/0x11 command in protocol specifications)
	  ○ 11 - axis switch options (see comment above)
	  ○ 12 - home input setting (see comment above)
	  ○ 13 - limit- input setting (see comment above)
	  ○ 14 - limit+ input setting (see comment above)
	- Param 3: parameter value
	Returns:
	- nothing

0xFF18	Pulse engine commands
	- last Param: Command ID (this parameter is put on stack on the last position)
	  ○ 0 - Execute home on selected axes
	    ▪ Param 1: bit-mapped axes to home

	Returns: - Parameter values (0-N) - Last parameter: N - number of parameters

0xFF19	UDP sender functionality
	- Last param: Command ID (this parameter is put on stack on the last position)
	  o 0 - send UDP packet
	    ▪ Param 1: 32-bit target IP address
	    ▪ Param 2: 16-bit target port number
	  o 1 - clear UDP packet buffer
	  o 10 - append text/binary data to UDP packet
	    ▪ Params 1-10: up to 40 bytes (stored in up to 10x 32-bit stack entries)
	    ▪ Second to last param: number of characters/bytes
	  o 11 - append number to UDP packet
	    ▪ Param 1: number
	    ▪ Param 2: format
	      • Bit 0: always show sign
	      • Bits 1-4: display digits count
	      • Bits 5-8: zero-padded digits count
	      • Bits 9-11: decimals count

0xFF1A	InterCom functionality
	- Last param: Command ID in lower 8 bits and target serial number (upper 24 bits)
	  o 0 - Send single InterCom data packet
	    ▪ Param 1: Data ID (lower 16 bits)
	    ▪ Param 2: data value

Table 1: Code memory

Data memory

Data memory address	Data type	Access	Function
0x0000 - 0x0FFF			Peripheral access (volatile memory)
0x0000	b,B,W,D	R/W	Digital pins (single input/output status per memory slot)
0x0064	b,B,W,D	R/W	Digital pin functions if bit 2 is set, the following rules are used (for writing operation): - Bit 0: if 1, set the output to the value, specified by bit 1 - Bit 1: digital output value
0x00F0	b,B,W,D	R/W	Digital pins (8 bit-mapped input/output statuses per memory slot)
0x0100	b,B	R	Matrix keyboard inputs (1 byte/input)
0x0200	D	R/W	Encoders – 4 bytes/encoder
0x0300	W	R	Analog inputs – 2 bytes/input
0x0380	D	R	Analog inputs - 4 bytes/input
0x0400	D	R/W	Digital input counters – 4 bytes/counter
0x04E0	D	R	Digital input capture counter values (on times) - 4 bytes / counter
0x04F0	D	R	Digital input capture counter values (off times) - 4 bytes / counter
0x0500	D	R/W	Sensor values (27 sensors on PoKeys56) – 4 bytes/sensor
0x05F8	b,D	R	Sensor OK values (bit encoded)
0x0600	W	R	RTC values 0 – Second, 2 – Minute, 4 – Hour, 6 – Day of week, 8 – Day of month, 10 – Month, 12 – Year
0x0610	W	R	PPM decoder values (8 channels) - 2 bytes/channel
0x0620	W	R	PPM decoder data age value (in 0.1 ms, max. 1000 ms)
0x0630	b,B	R/W	Joystick override flags: 0: bit-mapped joystick axes override 1-4: bit-mapped joystick buttons override 5: HAT switch override
0x0640	W	R/W	Joystick axis override values (6 values, 2 bytes/axis)
0x0650	b,B	R/W	Joystick buttons (4 bytes, bit-mapped), HAT switch (1 byte)
0x0700	D	R/W	PWM period - 4 bytes
0x0704	D	W	PWM period - 4 bytes -> writing to this address will not immediately reset the PWM counters
0x0710	B	R/W	PWM config registers – 1 byte per output
0x0720	D	R/W	PWM outputs duty cycles – 4 bytes per output
0x0800	b,B	R/W	PoExtBus outputs
0x0900	D	R/W	Pulse engine positions (8 axes)
0x0920	D	R/W	Reference position (8 axes)
0x0940	D	R/W	Reference speed (8 axes) (in steps/second)
0x0960	D	R	Probe position (8 axes)
0x0980	B	R/W	Pulse engine state
0x0981	B,b	R/W	Invert axis enable
0x0982	B,b	R	Limit+ status
0x0983	B,b	R	Limit- status
0x0984	B,b	R	Home switch status
0x0985	B,b	R/W	Limit override
0x0986	B,b	R	Error inputs status

0x0987	B,b	R/W	Axis enabled mask
0x0988	B	R	Axes states (see protocol specifications)
0x0990	B	R/W	Axes configuration (see protocol specifications)
0x0998	B	R/W	Axes switch configuration
0x09A0	B,b	R	Misc input status
0x09A1	B,b	R/W	External relay outputs
0x09A2	B,b	R/W	External OC outputs
0x09A3-0x09A7			reserved
0x09A8	B	R/W	MPG jog encoder setup
0x09B0	W	R/W	MPG jog multiplier (8 axes)
0x0A00	b,B,W,D	R/W	Battery-backed RAM (20-bytes)
0x0B00	B	W	LCD buffer (80-bytes)
0x0BFF	B	W	Writing any data to address 0x0BFF triggers the refresh of the LCD
0x0C00	B	R	Current device's IP address (4 successive bytes)
0x0C06	B	R	Current gateway IP address (4 successive bytes)
0x0C0C	B	R	Current network mask (4 successive bytes)
0xC012	B	R	DHCP status
0x0D00	D	R/W	Sensor values (100 sensors on PoKeys57) – 4 bytes/sensor
0x0EF0	b,D	R	Sensor OK values (bit encoded)
0x0F00	D	R/W	InterCom data values (total of 64)
0x1000 - 0x10FF			Shared data memory
0x1100 - 0x13FF			General purpose memory (PoKeys56 series)
0x1100 - 0x1FFF			General purpose memory (PoKeys57 series)
0xFF00 - 0xFFFF			Special system registers
0xFF00	W	R	PC (program counter)
0xFF04	B	R	Status register
0xFF08	D	R	System timer value (In milliseconds)
0xFF0C	D	R	Working register

Table 2: Data memory

PoIL instruction set

Unlike some specific instruction set implementations, PoIL does not differentiate between bit, 8-bit, 16-bit, 32-bit and literal oriented operations. The instruction set can be separated into the following categories:

- Processing operations
- Stack operations
- Control operations

PoIL instruction format

PoIL instructions consist of 2 byte opcode and additional operands. The presence and type of the operand is defined by the addressing mode and operand data type.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Addressing mode		Operand data type		Bit index			Bit invert	Opcode							

Bit	Symbol	Description
15:14	-	Addressing mode 00 – the operand contains the address 01 – the operand contains the address of a pointer to an address 10 – the operand is pushed to or popped from stack 11 – the operand is a literal value
13:12	-	Operand data type 00 – b: bit (1-bit) – no additional operand if addressing mode is literal or stack, otherwise 2-byte operand 01 – B: byte (8-bit) – 2-byte operand if addressing mode is not stack 10 – W: word (16-bit) – 2-byte operand if addressing mode is not stack 11 – D: double word (32-bit) – 4-byte operand if addressing mode is literal value, no operand if addressing mode is stack, 2-byte operand otherwise
11:9	-	Bit address (for bit data type)
8	-	Bit invert
7:0	OP	8-bit instruction opcode

Operand data types (sizes)

The following table shows the operand size (in bytes) for all combinations of addressing modes and operand data types.

		Addressing mode			
		Direct	Indirect	Stack	Literal
Operand data type	Bit	2	2	0	0
	Byte (8-bit)	2	2	0	2
	Word (16-bit)	2	2	0	2
	DWord (32-bit)	2	2	0	4

Instruction set summary

Instruction, operands		Description	Instruction opcode	Notes
Special instructions				
NOP		No operation	0x00	
Stack operations				
LOAD	o	Load W from o	0x01	
PUSH	o	Push o to stack	0x02	
STORE	o	Store W to o	0x80	
POP	o	Pop from stack and save to o	0x81	
Logical operations				
AND	o	AND W and o, store to W	0x10	
OR	o	OR W and o, store to W	0x11	
XOR	o	XOR W and o, store to W	0x12	
BITSET	o	Set bit, specified by o, store back to o	0x13	
BITCLR	o	Clear bit, specified by o, store back to o	0x14	
BITTGL	o	Toggle bit, specified by o, store back to o	0x15	
Arithmetic operations				
ADD	o	Add W and o, store to W	0x20	
SUB	o	Subtract o from W, store to W	0x21	
MUL	o	Multiply W and o, store to W	0x22	
DIV	o	Divide W by o, store to W	0x23	
MOD	o	Calculate modus of W by o, store to W	0x24	
DECT	o	Decrease value in o, compare with zero	0x25	Result saved in STATUS
ABS	o	Absolute value of o, store to W	0x26	
SHIFTL	o	Shift W by value of o to the left	0x40	
SHIFTR	o	Shift W by value of o to the right	0x41	
Compare operations				
CMPGT	o	Compare – W greater than o ?	0x30	Result saved in STATUS
CMPGTE	o	Compare – W greater than or equal o ?	0x31	Result saved in STATUS
CMPEQ	o	Compare – W equal to o ?	0x32	Result saved in STATUS
CMPNEQ	o	Compare – W not qual to o ?	0x33	Result saved in STATUS
CMPLTE	o	Compare – W lower than or equal o ?	0x34	Result saved in STATUS
CMPLT	o	Compare – W lower than o ?	0x35	Result saved in STATUS
BITTST	o		0x36	Result saved in STATUS
Control operations				
JMP	o	Unconditional branch to o	0x50	
JMPT	o	Conditional branch to o if L in STATUS is 1	0x51	
JMPF	o	Conditional branch to o if L in STATUS is 0	0x52	
CALL	o	Subroutine call, put PC to stack, jump to o	0x60	
RETURN		Return from subroutine, pop PC from stack	0x82	
EXIT		Exit current task	0x83	
COPY_V	o	Copy a vector to o	0x90	

Core states

State	Description
0	stopped (reset)
10	running
20	debug
100	exception
101	call stack overflow
101	call stack underflow
102	data stack overflow
103	data stack underflow
110	memory exception
111	save to literal not possible
112	vector operation to literal not possible
113	vector operation to pointer not possible
120	PC out of memory space
121	jump to odd address
122	jump instruction expects word operand
130	unknown instruction
140	unknown system function
141	wrong parameter for system function
150	operand not of type 0 for bit instruction
160	division by zero
161	mod by zero

Comments

PoIL compiler supports one-line comments that start with # sign.

```
# This is a one line comment

LOAD S   # This is an example of a code line comment that can describe the code at the left
```

Vector commands

Vector commands are used to specify a larger set of data. The basic commands are identical to other commands with the addition of additional data items, separated by commas, as shown below:

```
# Copy the set of bytes to location 0x1000
COPY_V Bh1000 5,10,23,124,255,0,5

# Copy the set of 32-bit integers to location 0x1000
COPY_V Dh1000 5,10,23,124,255,0,5,-235234,41234123
```

The length of data bytes must be a multiple of 2.

The data is saved as a set of binary data, following the opcode. First byte specifies the data length (number of bytes).

Address labels

To assign a label to a specific code section, a text label without spaces and a semicolon (:) sign must be put before the code section

Example 1

```
# This commands moves the program execution to the next code line under the 'Section_label'
JMP Section_label
…
# Code that does not gets executed
…

# This section is labelled with the 'Section_label'
Section_label:
# Section code below that gets executed

# The section label can also be in the same line as the code – simple for loop example:
LOAD BL[10]
STORE Bh1000
forLabel: DECT Dh1000
          JMPF forLabel
```

Example 2

```
label:
 LOAD  b!1234.0  # load inverted bit from 1234.0
 LOAD  Bh100
 ADD   WL[1500]
 ADD   b!L[0].0     # Add 1
 CMPGT WL[5000]
 JMPT  label      # default data type W
```

Syntax

Different addressing modes and data types

direct:	ADD	B100	# Add W and byte (8-bit) operand from address 100
index:	MUL	D[0]	# Multiply W and double word operand (32-bit) from # address stored at the memory location 0
stack:	STORE	S	# Store (push) W to stack
literal:	SUB	WL[5]	# Subtract word (16-bit) literal (with value 5) from W
bit:	BITSET	b0.5	# Set bit 5 at the memory address of 0
stackload:	PUSH	DL[100]	# Store (push) literal value 100 to stack
stackstore:	POP	D10	# Pop the double word (32-bit) value from the stack and # store it to memory address 10

These examples show that each operand (except for the stack operations) must be provided with the data type specification:

- b Bit data type – operand must end with .[bit number], where bit number is in the range 0-7
- B Byte data type
- W Word data type
- D Double word data type

Direct addressing mode

Label: INSTRUCTION {data type}{address}/.{bit}/

{data type} one of the data type specifiers described above – b, B, W or D. If 'b' is specified, the additional parameter {bit} must also be specified.

{address} address of the memory location

/.{bit}/ optional bit parameter

Examples:

LOAD D100 # Load double word (32-bit) from memory address 100

BITTST b100.5 # Test bit 5 of the memory address 100

Index addressing mode

Label: INSTRUCTION {data type}[{address}]/.{bit}/

{data type} one of the data type specifiers described above – b, B, W or D. If 'b' is specified, the additional parameter {bit} must also be specified.

[{address}] address of the memory location (of word type) that holds the address of the operand

[.{bit}] optional bit parameter

Examples:

LOAD D[100] # Load double word (32-bit) from memory address that is saved in address 100

BITTST b[100].5 # Test bit 5 of the memory address that is saved in address 100

Stack push/pop

Label: INSTRUCTION S

Examples:

LOAD S # Load double word (32-bit) from stack

ADD S # Add a value from stack to W

Literals

Label: INSTRUCTION {data type}L[{value}]/.{bit}/

{data type} one of the data type specifiers described above – b, B, W or D. If 'b' is specified, the additional parameter {bit} must also be specified.

{value} literal value (in decimal format)

/.{bit}/ optional bit parameter

Examples:

```
LOAD    DL[100]    # Load literal value 100 to W
```

Detailed instruction description

Instruction	*NOP – No operation*
Syntax	label: NOP
Description	No register is affected by NOP instruction execution

Instruction	*LOAD – Load W*
Syntax	label: LOAD O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	W is loaded with the value of O

Instruction	*PUSH – Push O to stack*
Syntax	label: PUSH O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The value of O is pushed to stack. W is not affected by this command

Instruction	*STORE – Store W*
Syntax	label: STORE O
Operand	O can be any of the direct memory, indexed memory addressing or stack push
Description	The value of W is stored to O

Instruction	*POP – Pop from stack and store to O*
Syntax	label: POP O
Operand	O can be any of the direct memory or indexed memory addressing
Description	First stack element is pop-ed from stack and saved to O. W is not affected by this command

Instruction	*AND – Bitwise AND between W and O*
Syntax	label: AND O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	A bitwise AND operation is executed between W and O and the result is saved to W.

Instruction	*OR – Bitwise OR between W and O*
Syntax	label: OR O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	A bitwise OR operation is executed between W and O and the result is saved to

	W.

Instruction	*XOR – Bitwise XOR between W and O*
Syntax	label: XOR O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	A bitwise XOR operation is executed between W and O and the result is saved to W.

Instruction	*BITSET – Set bit*
Syntax	label: BITSET O
Operand	O can be any of the direct memory or indexed memory addressing
Description	This operation sets the specified bit of the operand and stores it back. W is not affected.
Example	*BITSET b100.5*

Instruction	*BITCLR – Clear bit*
Syntax	label: BITCLR O
Operand	O can be any of the direct memory or indexed memory addressing
Description	This operation clears the specified bit of the operand and stores it back. W is not affected.
Example	*BITCLR b100.5*

Instruction	*BITTGL – Toggle bit*
Syntax	label: BITTGL O
Operand	O can be any of the direct memory or indexed memory addressing
Description	This operation toggles the specified bit of the operand and stores it back. W is not affected.

Instruction	*ADD – Add O and W*
Syntax	label: ADD O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The operand O is added to W and the result is saved to W.

Instruction	*SUB – Subtract O from W*
Syntax	label: SUB O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The operand O is subtracted from W and the result is saved to W.

Instruction	*MUL – Multiply O and W*
Syntax	label: MUL O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	O and W are multiplied and saved to W.

Instruction	*DIV – Divide W by O*
Syntax	label: DIV O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The W is divided by O and the result is saved to W.

Instruction	*MOD – Remainder on division of W by O*
Syntax	label: MOD O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The remainder on division of W by O is saved to W.

Instruction	*DECT – Decrement O and test if zero*
Syntax	label: DECT O
Operand	O can be any of the direct memory, indexed memory addressing, stack.
Description	The value in O is decremented and saved back to O. If the new value of O is zero, the logical result bit of the status register is set.
Example	The following code is a simple for loop implementation LOAD DL[5] STORE S for2: # do something here DECT S JMPF for2 Which in pseudo code would be equal to For i = 1 to 5 [do something here]

Instruction	*CMPGT – Compare O to W – is O greater than W?*
Syntax	label: CMPGT O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The compare statement. The result of comparison O > W is saved to logical result

	bit of the status register.

Instruction	*CMPGTE – Compare O to W – is O greater or equal to W?*
Syntax	label: CMPGTE O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The compare statement. The result of comparison O >= W is saved to logical result bit of the status register.

Instruction	*CMPEQ – Compare O to W – are O and W equal?*
Syntax	label: CMPEQ O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The compare statement. The result of comparison O == W is saved to logical result bit of the status register.

Instruction	*CMPNEQ – Compare O to W – are O and W not equal?*
Syntax	label: CMPNEQ O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The compare statement. The result of comparison O != W is saved to logical result bit of the status register.

Instruction	*CMPLTE – Compare O to W – is O lower than or equal to W?*
Syntax	label: CMPLTE O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The compare statement. The result of comparison O <= W is saved to logical result bit of the status register.

Instruction	*CMPLT – Compare O to W – is O lower than W?*
Syntax	label: CMPLT O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The compare statement. The result of comparison O < W is saved to logical result bit of the status register.

Instruction	*BITTST – Test bit status*
Syntax	label: BITTST O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal

Description	This operation tests the status of the specified bit of the operand. If bit is set, the logical result bit of the status register is set (and cleared if the bit of the operand is not set).

Instruction	*SHIFTL – Shift W left for O*
Syntax	label: SHIFTL O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The working register W is shifted left for O places. The result is kept in W. This operation does not support the 'rotate' function – bit that 'fall-off' at the left end of the W register is discarded.

Instruction	*SHIFTR – Shift W right for O*
Syntax	label: SHIFTR O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The working register W is shifted right for O places. The result is kept in W. This operation does not support the 'rotate' function – bit that 'fall-off' at the right end of the W register is discarded.

Instruction	*JMP – Jump to program address*
Syntax	label: JMP O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The PC is loaded with the O.

Instruction	*JMPT – Jump to program address if true*
Syntax	label: JMPT O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The PC is loaded with the O if the logical result bit of the status register is set
Example	The following code is a simple for loop implementation

```
  LOAD  DL[0]  # i = 0
  STORE D0
  LOAD  DL[10]  # i = 0
  STORE D4

for:
  # do something here

  LOAD  D0
  CMPLT D4
  ADD   BL[1]
  STORE D0
  JMPT  for
```

Which in pseudo code would be equal to

```
For i = 1 to 10
   [do something here]
```

However, a more appropriate instruction for the 'for' loop implementation is DECT (decrement and test)

Instruction	*JMPF – Jump to program address if false*
Syntax	label: JMPF O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	The PC is loaded with the O if the logical result bit of the status register is not set

Instruction	*CALL – Call a subroutine*
Syntax	label: CALL O
Operand	O can be any of the direct memory, indexed memory addressing, stack pop or literal
Description	PC+4 is first put to function address stack, then the PC is loaded with the O.

Instruction	*RETURN – Return from subroutine*
Syntax	label: RETURN
Operand	None
Description	The PC is loaded with the first function address stack entry.

Start-up configuration

First PoIL instruction should load the configuration byte into working register W. On startup, PoIL core will execute first command and check the contents of the W. The value is interpreted as

- Bit 0: proceed with core reset code (this will execute code until first task EXIT is encountered)
- Bit 1: proceed with code execution
- Bit 2: disable division by zero and modulus by zero exceptions - these operations will result in 0
- Bit 3: try automatically reloading the project on PoBlocks startup (requires a hash code of the project to be loaded in the second PoIL command)

Shared data management

First 256 bytes of general purpose memory is reserved for shared data table. Each of 64 shared data entries uses 32-bit integer entry as specified below:

Bit	Description
Bits 31:30	Access rights (bit 30: read, bit 31: write)
Bits 29:28	Data type (as specified in table below - Operand data type)
Bits 27:25	Bit ID (for data type 0)
Bits 15:0	Data address pointer - 16-bit

Custom PoIL block application examples

Min, max, average meter over interval

Task: *The block should calculate the average of the input value, find its minimum and maximum value. The block should have the enable input, clock input for signalizing each measurement and the reset input to reset the values back to default*

We will define 4 input variables, 3 output variables, 4 32-bit variables and one 1-bit (logic) variable. The following block is entered into 'Variables declaration'

```
Value : INPUT(1,int32)
EN : INPUT(2,bit)
RST : INPUT(3,bit)
CLK : INPUT(4,bit)
min : OUTPUT(1,int32)
max : OUTPUT(2,int32)
average : OUTPUT(3,int32)

sum : int32
counter : int32
minValue : int32
maxValue : int32
clkBit : bit
```

The code checks the reset input first, then checks enable and clock signals. If all is ok, minimum, maximum and the cumulative sum of input values is calculated. In the end, the average is calculated from the cumulative sum and number of measurements.

```
# Check reset - load 0
LOAD DL[0]
# Compare 0 < RST
CMPLT RST
JMPF checkEnable

# We have 0 already in W register
STORE sum
STORE counter
STORE maxValue
LOAD DL[1000000000]
STORE minValue
JMP exitBlock

# Check enable
checkEnable:
CMPLT EN
JMPF exitBlock

# Check clock signal (condition: CLK[k] AND (NOT CLK[k-1]))
LOAD CLK
STORE S      # put the current clock input state to stack
AND !clkBit
POP clkBit   # save the current clock input state to clkBit memory
CMPEQ DL[1]
JMPF exitBlock

# Increase counter
LOAD counter
ADD DL[1]
STORE counter
# Add input...
LOAD sum
```

```
ADD Value
STORE sum

# Check min and max
LOAD Value
CMPLT minValue
JMPF checkMax
STORE minValue
checkMax:
CMPGT maxValue
JMPF exitBlock
STORE maxValue
exitBlock:
LOAD minValue
STORE min
LOAD maxValue
STORE max

# Produce average
LOAD counter
CMPGT DL[0]
JMPT findAverage     # if counter is zero, skip division
LOAD DL[0]
STORE average
JMP exitFinal

# Calculate the average
findAverage:
LOAD sum
DIV counter
STORE average
exitFinal:
```

Grant of license

support: www.poscope.com