

MARC4
4 Bit Microcontroller
Programmer's Guide

1994

TEMIC
TELEFUNKEN Semiconductors

I. Hardware Description



II. Programming in qFORTH



III. Instruction Set



IV. qFORTH Language Dictionary



Addresses



I. Hardware Description	5
MARC4 microcontroller	7
Introduction	7
Features	7
1. MARC4 architecture	8
1.1 General description	8
1.2 Components of MARC4 core	9
1.2.1 Program memory (ROM)	9
1.2.2 Data memory (RAM)	10
1.2.3 Registers	11
1.2.4 ALU	13
1.2.5 Instruction set	14
1.2.6 I/O bus	14
1.2.7 Interrupt structure	16
1.2.7.1 Software interrupts	18
1.2.7.1 Hardware interrupts	18
1.3 Reset	18
1.4 SLEEP mode	18
1.6 Peripherals	19
1.6.1 Addressing peripherals	19
1.6.2 Input ports	21
1.6.3 Output port	21
1.6.4 Bidirectional ports	22
1.6.5 External interrupt inputs	23
1.7 Emulation	24
1.6.7.1 Stand alone EPROM boards	25
1.8 MARC4 opcode formats	26
1.9 MARC4 instruction set overview	27
II. Programming in qFORTH	29
2. Programming in qFORTH	31
Why program in qFORTH ?	31
2.1. Language overview	32
2.2. The qFORTH vocabulary: words and definitions	34
2.3. Stacks, RPN and comments	35
2.3.1. Reverse polish notation	35

2.3.2.	The qFORTH stacks	35
2.3.3.	Stack notation	35
2.3.4.	Comments	36
2.4.	Constants and variables	36
2.4.1.	Constants	36
2.4.1.1.	Predefined constants	36
2.4.2.	Look-up tables	37
2.5.	Variables and arrays	38
2.5.1.	Defining arrays	38
2.6.	Stack allocation	39
2.6.1.	Stack pointer initialisation	39
2.7.	Stack operations, reading and writing	40
2.7.1.	Stack operations	40
2.7.1.1.	The data stack	40
2.7.1.2.	SWAP	40
2.7.1.3.	DUP, OVER and DROP	40
2.7.1.4.	ROT and <ROT	41
2.7.1.5.	R>, >R, R@ and DROPR	41
2.7.1.6.	Other useful stack operations	42
2.7.2.	Reading and writing (@, !)	43
2.7.3.	Low level memory operations	43
2.7.3.1.	RAM address registers X and Y	43
2.7.3.2.	Bit manipulations in RAM	44
2.8.	MARC4 condition codes	45
2.8.1.	The CCR and the control operations	45
2.9.	Arithmetic operations	45
2.9.1.	Number systems	45
2.9.1.1.	Single and double length operators	45
2.9.2.	Addition and subtraction	46
2.9.3.	Increment and decrement	46
2.9.4.	Mixed-length arithmetic	46
2.9.5.	BCD arithmetic	46
2.9.5.1.	DAA and DAS	46
2.9.6.	Summary of arithmetic words	47
2.10.	Logicals	48
2.10.1.	Logical operators	48
2.10.1.1.	TOGGLE	49

2.10.1.2.	SHIFT and ROTATE operations	49
2.11.	Comparisons	50
2.11.1.	< , >	50
2.11.2.	<= , >=	50
2.11.3.	<> , =	51
2.12.	Control Structures	51
2.12.1.	Selection control structures	52
2.12.1.1.	IF .. THEN	52
2.12.1.2.	The CASE structure	53
2.12.2.	Loops, branches and labels	53
2.12.2.1.	Definite loops	53
2.12.2.2.	Indefinite loops	54
2.12.3.	Branches and labels	55
2.12.4.	Arrays and Look-up tables	56
2.12.5.	Look-up tables	57
2.12.6.	TICK and EXECUTE	57
2.13.	Making the best use of compiler directives	60
2.13.1.	Controlling ROM placement	60
2.13.2.	Macro definitions, EXIT and ;;	60
2.13.3.	Controlling stack side effects	61
2.13.4.	\$INCLUDE directive	61
2.13.5.	Conditional compilation	61
2.13.6.	Controlling XY register optimisations	62
2.14.	Recommended naming conventions	63
2.14.1.	How to pronounce the symbols	64
2.15.	Book list	64
2.1.5.1	Recommended books	64
2.1.5.2	General interest	65
III.	Instruction Set	67
3.	MARC4 instruction set	69
3.1.	Introduction	69
3.1.1.	Description of used identifiers and abbreviations	75
3.1.2.	Stack notation	75
3.2.	The qFORTH language -quick reference guide	76
3.2.1.	Arithmetic/logical	76
3.2.2.	Comparisons	76

3.2.3.	Control structures	77
3.2.4.	Stack Operations	77
3.2.5.	Memory operations	78
3.2.6.	Predefined structures	79
3.2.7.	Assembler mnemonics	79
IV.	qFORTH Dictionary	81
4.1.	Preface	83
4.2.	Introduction	84
4.3.	Stack related conventions	87
4.4.	Flags and condition code register	89
4.5.	MARC4 memory addressing model	90
4.6.	Short form dictionary	91
4.7.	Index	101

I. Hardware Description

II. Programming in qFORTH

III. Instruction Set

IV. qFORTH Language Dictionary

Addresses

MARC4 microcontroller

Introduction

The TEMIC MARC4 microcontroller family bases on a low power 4-bit CPU core. The modular MARC4 architecture is HARVARD like, high level language oriented and well suitable to realize high integrated microcontrollers with a variety of application or customer specific on chip peripheral combinations. The MARC4 controller's low voltage and low power consumption is perfect for hand-held and battery operated applications.

The standard members of the family have selected peripheral combinations for a broad range of applications.

Programming is supported by an easy-to-use PC based software development system with a high level language qFORTH-compiler and an emulator board. The FORTH oriented microcontroller concept enables the qFORTH compiler to generate compact and efficient MARC4 program code.

Features

- 4-bit HARVARD architecture
- High level language oriented CPU
- 256 x 4-bit of RAM
- Up to 9 KBytes of ROM
- 8 vectored prioritised interrupt levels
- Low voltage operating range
- Low power consumption
- Power down mode
- Various on-chip peripheral combination available
- High level programming language qFORTH
- Programming and testing is supported by an integrated software development system

1. MARC4 architecture

1.1 General description

The MARC4 microcontroller consists of an advanced stack based 4-bit CPU core and application specific on-chip peripherals like I/O ports, timers, counters, ADC, etc.

The CPU is based on the HARVARD architecture with physically separate program memory (ROM) and data memory (RAM). Three independent buses, the instruction-, the memory- and the I/O bus are used for parallel communication between ROM, RAM and peripherals. This enhances program execution speed by allowing both instruction prefetching,

and a simultaneous communication to the on-chip peripheral circuitry.

The integrated powerful interrupt controller with 8 prioritized interrupt levels supports fast processing of hardware events.

The MARC4 is designed for the high level programming language qFORTH. A lot of qFORTH instructions and two stacks the return and the expression stack are already implanted. The architecture allows high level language programming without any loss in efficiency and code density.

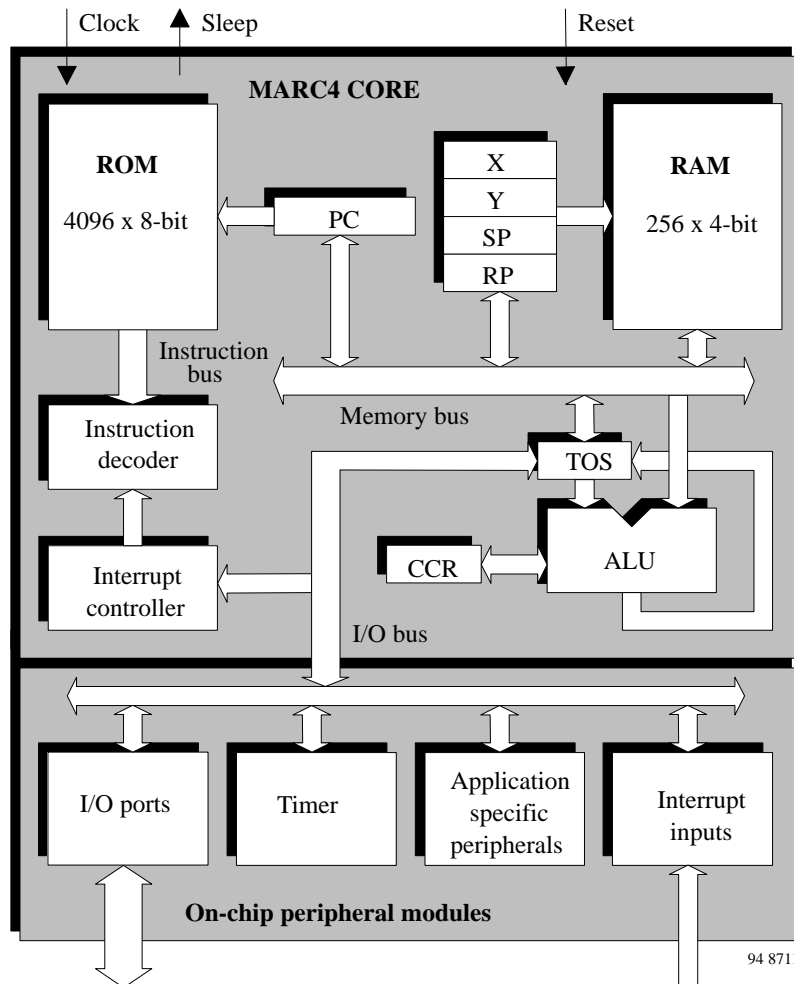


Figure 1 MARC4 core

1.2 Components of MARC4 core

The core contains the program memory (ROM), data memory (RAM), ALU, program counter, RAM address register, instruction decoder and interrupt controller. The following sections describe each of these parts.

1.2.1 Program memory (ROM)

The MARC4's program memory contains the customer application program. The 12-bit wide program counter can address up to 4 Kbytes of program memory. The access program memory with more than 4 K is possible using the bank switching method. One of 4 memory banks can be selected with bit 2 and 3 of the I/O-port D. Each ROM bank has a size of 2 Kbytes and is

placed above the base bank in the upper 2K (800h–FFFh) of the address space. This enables program memory sizes of up to 10 Kbytes. 1 Kbyte of bank 3 are normally reserved for test software purposes. After any hardware reset ROM bank 1 is selected automatically.

The program memory starts with a 512 byte segment (Zero Page) which contains predefined start addresses for interrupt service routines and special subroutines accessible with single byte instructions (SCALL). The corresponding memory map is shown in figure 2.

Look-up tables of constants can also be held in ROM and are accessed via the MARC4 builtin TABLE instruction.

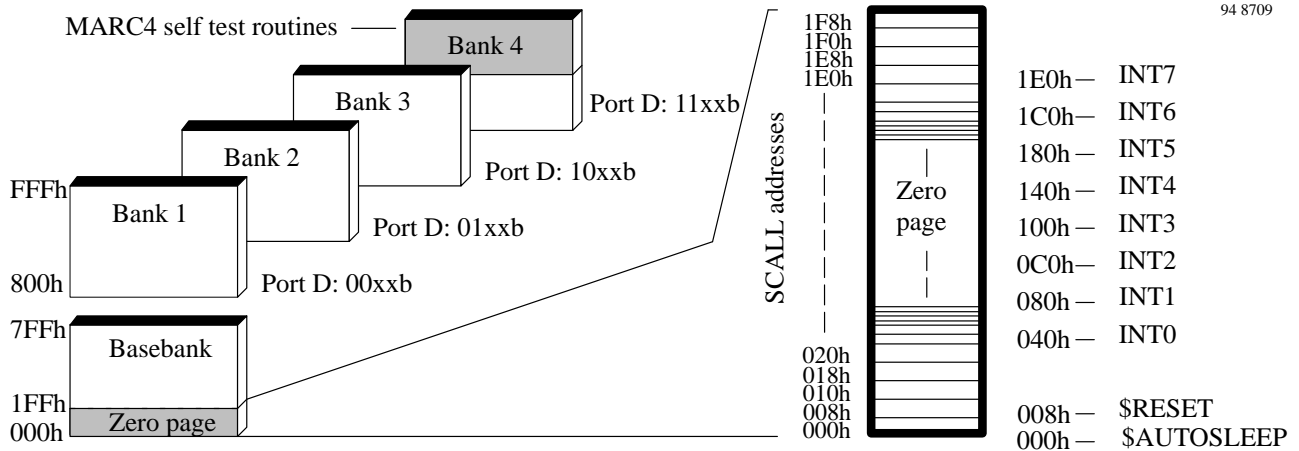


Figure 2 ROM Map

1.2.2 Data memory (RAM)

The MARC4 contains a 256 x 4-bit wide static random access memory (RAM). It is used for the expression stack, the return stack and as data

memory for variables and arrays. The RAM is addressed by any of the four 8-bit wide RAM address registers SP, RP, X and Y.

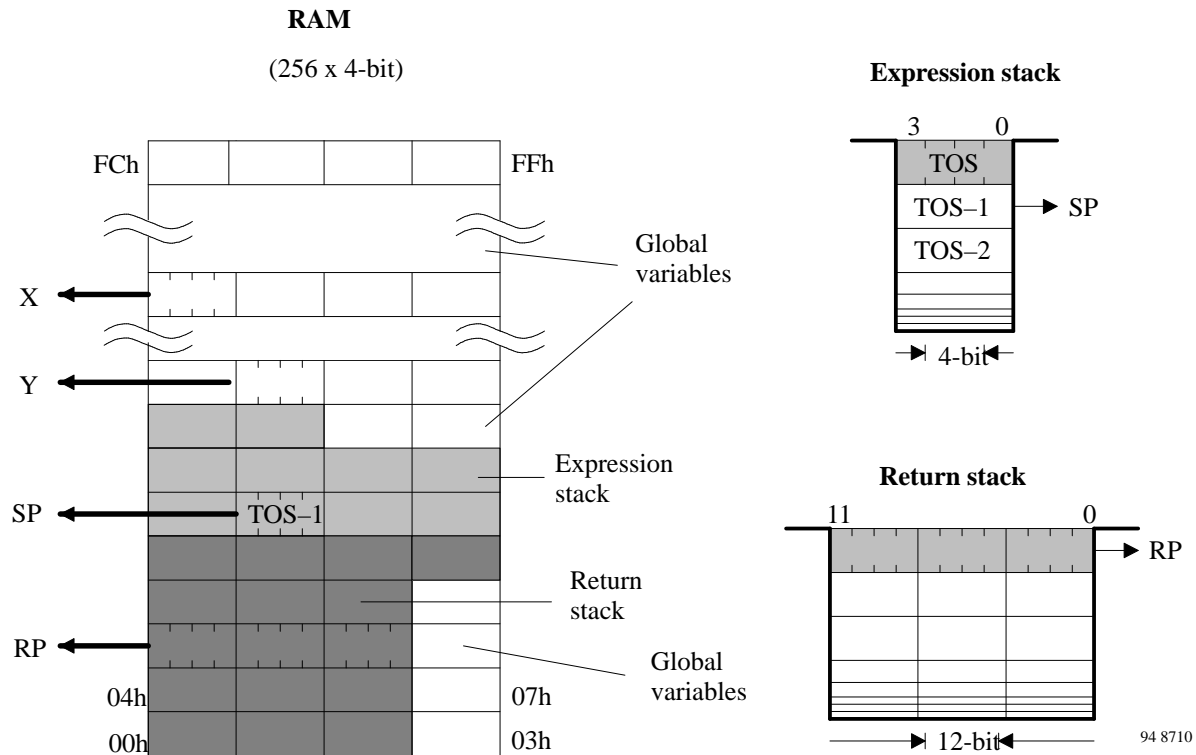


Figure 3 RAM Map

Expression stack

The 4-bit wide expression stack is addressed with the expression stack pointer (SP). All arithmetic, I/O and memory reference operations take their operands from, and return their result to the expression stack. The MARC4 performs the operations with the top of stack items (TOS and TOS-1). The TOS register contains the top element of the expression stack and works like an accumulator.

This stack is also used for passing parameters between subroutines, and as a scratchpad area for temporary storage of data.

Return stack

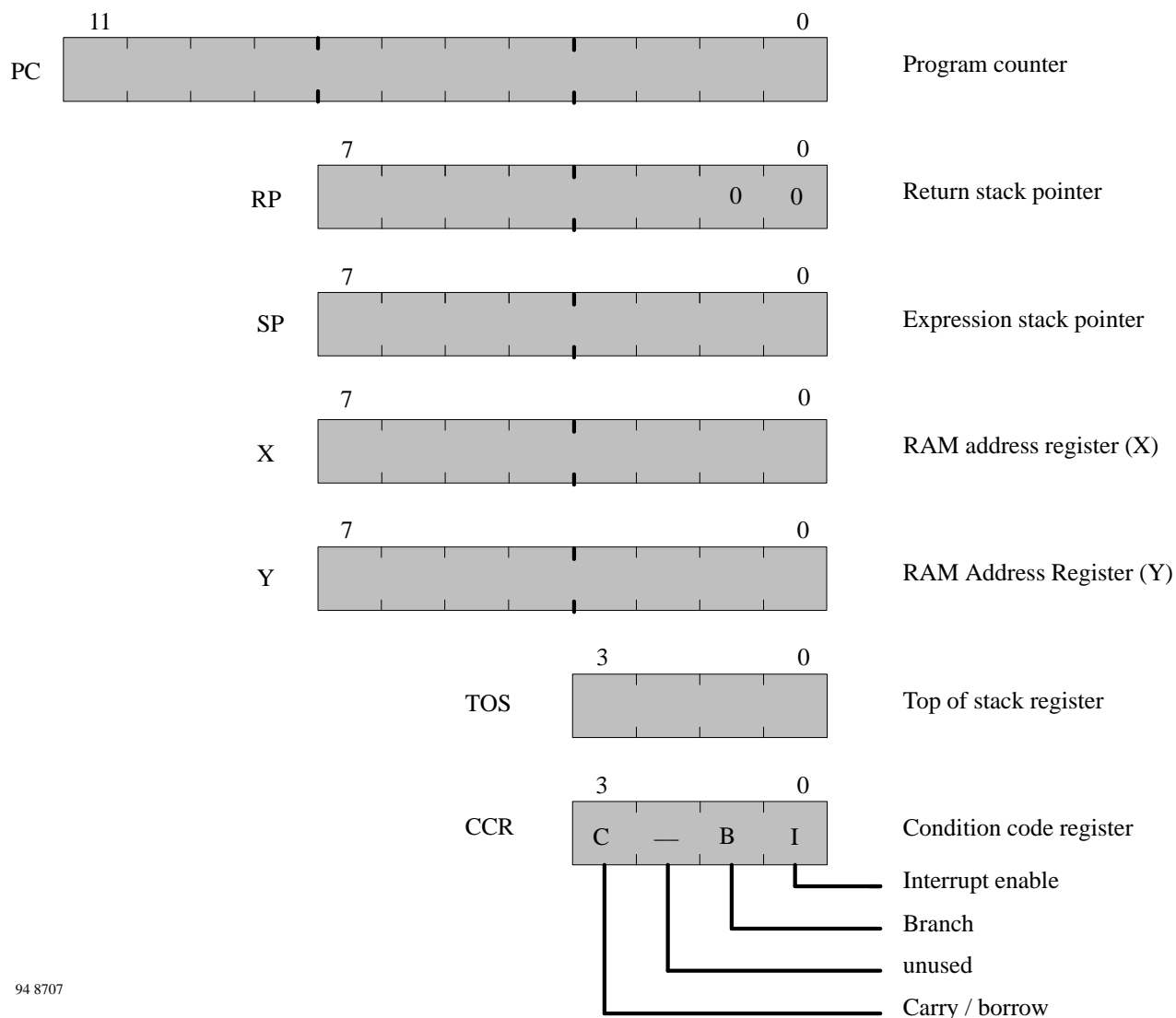
The 12-bit wide return stack is addressed by the return stack pointer (RP). It is used for storing return addresses of subroutines, interrupt routines and for keeping loop index counters. The return stack can also be used as a temporary storage area.

The MARC4 return stack starts with the AUTOSLEEP vector at the RAM location FCh and increases in the address direction 00h, 04h, 08h, ... to the top.

The MARC4 instruction set supports the exchange of data between the top elements of the expression- and the return stack. The two stacks within the RAM have a user definable maximum depth.

1.2.3 Registers

The MARC4 controller has 6 programmable registers and one condition code register. They are shown in the following programming model.



94 8707

Figure 4 Programming model

Program counter (PC)

The Program counter (PC) is a 12-bit register that contains the address of the next instruction to be fetched from the ROM. Instructions currently being executed are decoded in the instruction decoder to determine the internal micro-operations.

For linear code (no calls or branches) the program counter is incremented with every instruction cycle. If a branch, call, return instruction or an interrupt is executed the program counter is loaded with a new address.

The program counter is also used with the table instruction to fetch 8-bit wide ROM constants.

MARC4 Programmer's Guide Hardware Description

RAM address register

The RAM is addressed with the four 8-bit wide RAM address registers SP, RP, X and Y. This registers allow the access to any of the 256 RAM nibbles.

Expression stack pointer (SP)

The stack pointer (SP) contains the address of the next-to-top 4-bit item (TOS-1) of the expression stack. The pointer is automatically pre-incremented if a nibble is pushed onto the stack or post-decremented when a nibble is removed from the stack. Every post-decrement operation moves the item (TOS-1) to the TOS register before the SP is decremented.

After a reset the stack pointer has to be initialized with the compiler variable S0 (“> SP S0”) to allocate the start address of the expression stack area.

Return stack pointer (RP)

The return stack pointer points to the top element of the 12-bit wide return stack. The pointer automatically pre-increments if an element is moved onto the stack or it post-decrements if an element is removed from the stack. The return stack pointer increments and decrements in steps of 4. This means that every time a 12-bit element is stacked, a 4-bit RAM location is left unwritten. These location are used by the qFORTH compiler to allocate 4-bit variables.

To support the AUTOSLEEP feature, read and write operations to the RAM address FCh using the return stack pointer are handled on a special way. Read operations will return the autosleep address 000h, whereby write operatins have no affect.

After a reset the return stack pointer has to be initialized with “> RP FCh”.

RAM address register (X and Y)

The X- and Y-register are used to address any 4-bit element in the RAM. A fetch operation moves the addressed nibble onto the TOS. A store operation moves the TOS to the addressed RAM location.

Using either the pre-increment or post-decrement addressing mode it is convenient to compare, fill or move arrays in the RAM.

Top of stack (TOS)

The top of stack register is the accumulator of the MARC4. All arithmetic/logic-, memory reference- and I/O operations use this register. The TOS register gets the data from the ALU, the ROM, the RAM or via the I/O bus.

Condition code register (CCR)

The 4-bit wide condition code register contains the branch, the carry and the interrupt enable flag. These bits indicates the current state of the CPU. The CCR flags are set or reset by ALU operations. The instructions SET_BCF, TOG_BF, CCR! and DI allow a direct manipulation of the condition code register.

Carry/borrow (C)

The carry/borrow flag indicates that borrow or carry out of arithmetic logic unit (ALU) occurred during the last arithmetic operation. During shift and rotate operations this bit is used as fifth bit. Boolean operations have no affect on the C-flag.

Branch (B)

The branch flag controls the conditional program branching. When the branch flag was set by one of the previous instructions a conditional branch is taken. This flag is affected by arithmetic-, logic-, shift-, and rotate operations.

Interrupt enable (I)

The interrupt enable flag enables or disables the interrupt processing on a global basis. After reset or by executing the DI instruction the interrupt enable flag is cleared and all interrupts are disabled. The μ C does not process further interrupt requests until the interrupt enable flag is set again by either executing an EI, RTI or SLEEP instruction.

1.2.4 ALU

The 4-bit ALU performs all the arithmetic, logical, shift and rotate operations with the top two elements of the expression stack (TOS and

TOS-1) and returns their result to the TOS. The ALU operations affect the carry/borrow and branch flag in the condition code register (CCR).

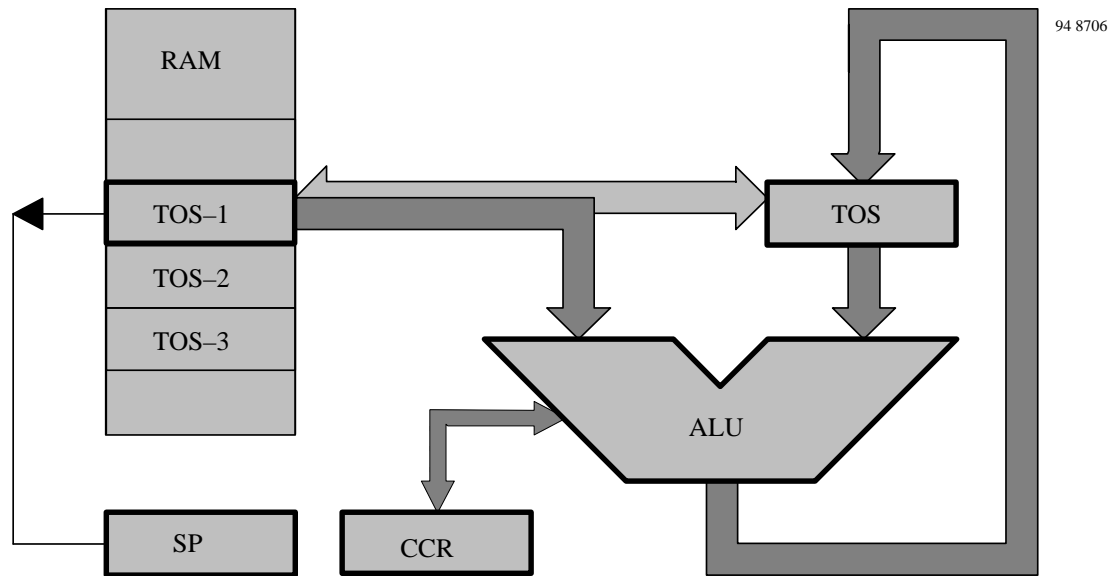


Figure 5 ALU zero address operations

1.2.5 Instruction set

The MARC4 instruction set is optimized for the high level programming language qFORTH. A lot of MARC4 instructions are qFORTH words. This enables the compiler to generate fast and compact program code.

The MARC4 is a zero address machine with a compact and efficient instruction set. The instructions contain only the operation to be performed and no source or destination address information. The operations are performed with the data placed on the stack.

A instruction pipeline enables the controller to fetch the next instruction from ROM at the same time as the present instruction is being executed. There are one and two byte instructions which

are executed within 1 to 4 machine-cycles. Most of the instructions have a length of one byte and are executed in only one machine cycle.

A complete overview of the MARC4 instruction set includes the table instruction set.

MARC4 instruction timing

The internal instruction timing and pipelining during the MARC4's instruction execution are shown in figure 6.

The figure shows the timing for a sequence of three instructions. A machine cycle consists of two system clock cycles. The first and second instruction needs one and the third two machine cycles.

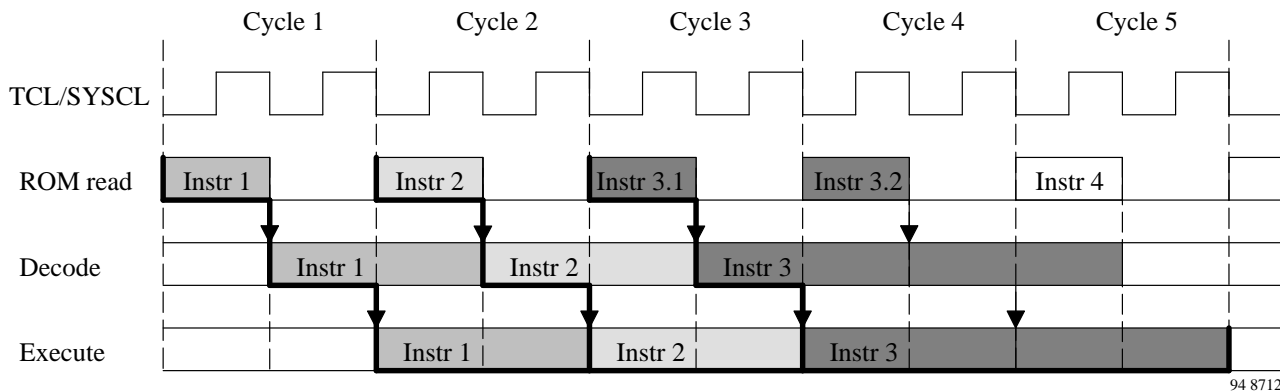


Figure 6 Instruction cycle (pipelining)

1.2.6 I/O bus

The communication between the core and the on chip peripherals takes place via the I/O bus. This bus is used for read and write accesses, for interrupt requests, for peripheral reset and for the SLEEP mode. The operation mode of the 4-bit wide I/O bus is determined by the control signals N_Write, N_Read, N_Cycle and N_Hold. (see table I/O bus modes).

During IN/OUT operations the address and data and during an interrupt cycle for the low and the high priority interrupts are multiplexed using the N_cycle signal. When N_cycle is low the address respectively the low interrupts '0,1,2,3' are sent, when N_cycle is high the data

respectively the higher priority interrupts '4,5,6,7' are transferred.

An IN operation transfers the port address from TOS (top of stack) onto the I/O bus and read the data back on TOS. An OUT operation transfers both the port address from TOS and the data from TOS-1 onto the I/O bus.

Note that the interrupt controller samples interrupt requests during the non-I/O cycles therefore IN and OUT instructions may cause an interrupt delay. To minimize interrupt latency avoid immediate consecutive IN and OUT instructions.

Table 1: I/O bus modes

Mode	N_read	N_write	N_cycle	N_hold	I/O bus
I/O read (address cycle)	0	1	0	1	x
I/O read (data cycle)	0	1	1	x	x
I/O write (address cycle)	1	0	0	1	x
I/O write (data cycle)	1	0	1	1	x
Interrupt 0 to 3 cycle	1	1	0	1	x
Interrupt 4 to 7 cycle	1	1	1	1	x
Sleep mode	0	0	0	1	0
Reset mode	0	0	x	0	Fh

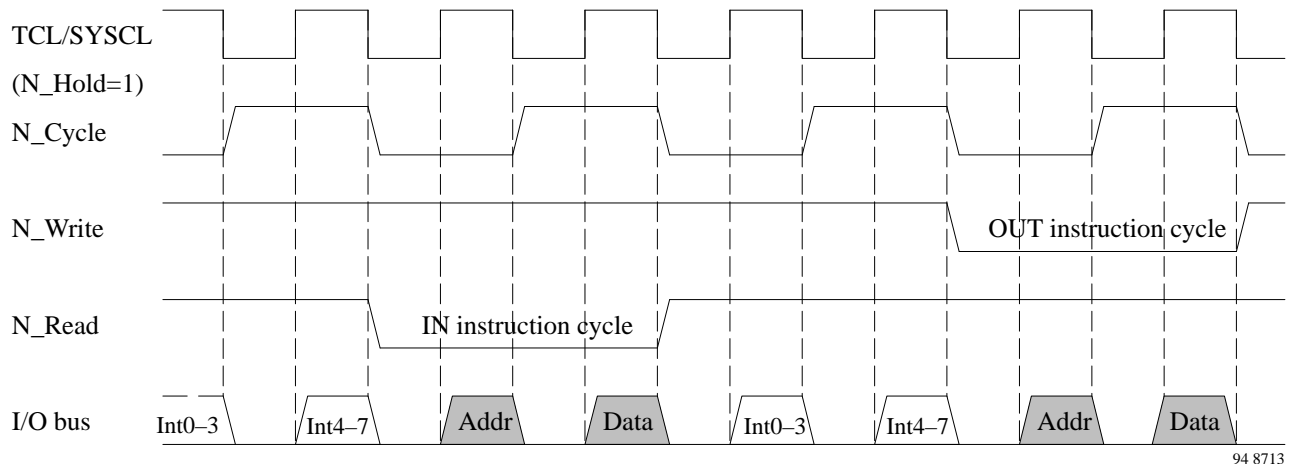


Figure 7 Timing for IN/OUT operations and interrupt requests

The I/O bus is internal and therefore not accessible by the customer on the final microcontroller.

More about the access to on-chip peripherals is described in the chapter peripheral modules.

1.2.7 Interrupt structure

The MARC4 can handle interrupts with 8 different priority levels. They can be generated from internal or external hardware interrupt sources or by a software interrupt from the CPU itself. Each interrupt level has a hard-wired

priority and an associated vector for the service routine in the ROM (see Table 2). The programmer can enable or disable interrupts all together by setting or resetting the interrupt-enable flag (I) in the CCR.

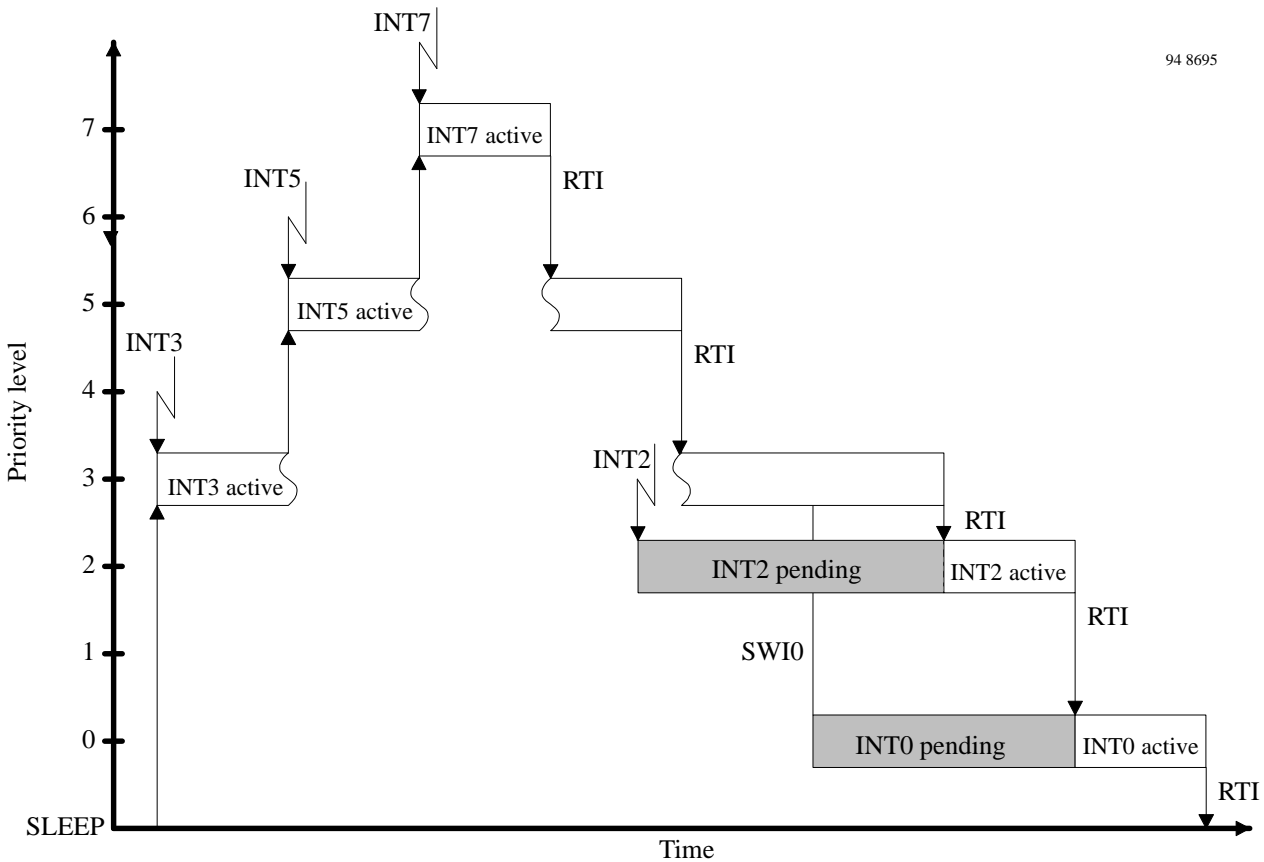


Figure 8 Interrupt handling

Interrupt processing

To process the 8 different interrupt levels the MARC4 contains an interrupt controller with the 8-bit wide interrupt pending and interrupt active register.

The interrupt controller samples all interrupt requests on the I/O bus during every non-I/O instruction cycle and latches them in the interrupt pending register. If no higher priority interrupt is present in the interrupt active register it signals the CPU to interrupt the current program execution. If the interrupt enable bit is

set the processor enters an interrupt acknowledge cycle. During this cycle a SHORT CALL instruction to the service routine is executed and the 12-bit wide current PC is saved on the return stack automatically.

An interrupt service routine is finished with the RTI instruction. This instruction sets the interrupt enable flag, resets the corresponding bits in the interrupt pending/active register and moves the return address from the return stack to the program counter.

When the interrupt enable flag is reset (interrupts are disabled), the execution of interrupts is inhibited but not the logging of the interrupt requests in the interrupt pending register. The execution of the interrupt will be delayed until the interrupt enable flag is set again. But note that interrupts are lost if a interrupt request occurs during the corresponding bit in the pending register is still set.

After any hardware reset (power-on-, external-

or watchdog-reset) the interrupt enable flag, the interrupt pending and interrupt active register are reset.

Interrupt latency

The interrupt latency is the time from the occurrence of the interrupt event to the interrupt service routine being activated. In the MARC4 this takes between 3 to 5 machine cycles depending on the state of the core.

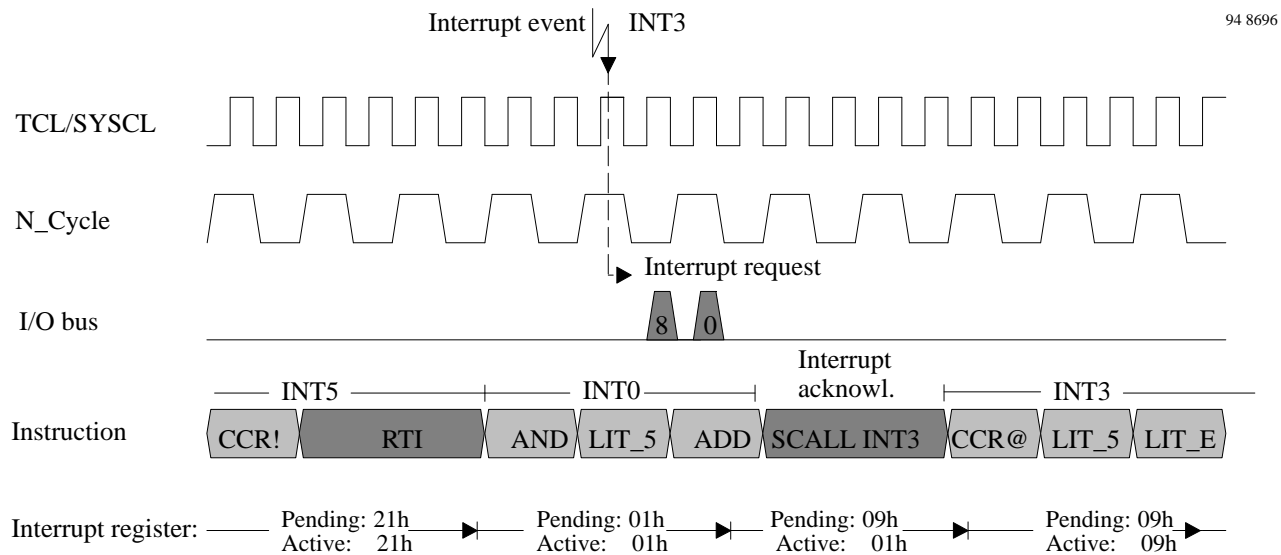


Figure 9 Interrupt request cycle

Table 2: Interrupt priority table

Interrupt	Priority	ROM address	Interrupt opcode (acknowledge)	Pending-/active-bit
INT0	lowest	040h	C8h (SCALL 040h)	0
INT1		080h	D0h (SCALL 080h)	1
INT2		0C0h	D8h (SCALL 0C0h)	2
INT3		100h	E0h (SCALL 100h)	3
INT4		140h	E8h (SCALL 140h)	4
INT5		180h	F0h (SCALL 180h)	5
INT6		1C0h	F8h (SCALL 1C0h)	6
INT7	highest	1E0h	FCh (SCALL 1E0h)	7

1.2.7.1 Software interrupts

The programmer can generate interrupts using the software interrupt instruction (SWI) which is supported in qFORTH by predefined macros named SWI0...SWI7. The software triggered interrupt operates exactly like any hardware triggered interrupt. The SWI instruction takes the top two elements from the expression stack and writes the corresponding bits via the I/O bus to the interrupt pending register. Thus using the SWI instruction, interrupts can be re-prioritized or lower priority processes scheduled for later execution.

1.2.7.1 Hardware interrupts

Hardware interrupt sources like external interrupt inputs, timers etc. are used for an fast automatically event controlled program flow. The different vectored interrupts permits program dividing into different interrupt controlled tasks.

1.4 SLEEP mode

The sleep mode is a shutdown condition which is used to reduce the average system power consumption in applications where the μC is not fully utilised. In this mode the system clock is stopped. The sleep mode is entered with the SLEEP instruction. This instruction sets the interrupt enable bit (I) in the condition code register to enable all interrupts and stops the core. During the sleep mode the peripheral modules remain active and are able to generate interrupts. The μC exits the sleep mode with any interrupt or a reset.

The SLEEP mode can only be kept when none

1.3 Reset

The reset puts the CPU into a well-defined condition. The reset can be triggered by switching on the supply voltage, by a break-down of the supply voltage, by the watchdog timer or by pulling the NRST pad to low.

After any reset the interrupt enable flag in the condition code register (CCR), the interrupt pending register and the interrupt active register are reset. During the reset cycle the I/O bus control signals are set to 'reset mode' thereby initializing all on-chip peripherals.

The reset cycle is finished with a short call instruction (opcode C1h) to the ROM-address 008h. This activates the initialization routine \$RESET. With that routine the stack pointers, variables in the RAM and the peripheral must be initialized.

of the interrupt pending or active register bits are set. The application of \$AUTOSLEEP routine ensures the correct function of the SLEEP-mode.

The total power consumption is directly proportional to the active time of the μC . For a rough estimation of the expected average system current consumption, the following formula should be used:

$$I_{\text{total}}(V_{\text{DD}}, f_{\text{Osc}}) = I_{\text{Sleep}} + (I_{\text{DD}} * T_{\text{active}} / T_{\text{total}})$$

I_{DD} depends on V_{DD} and f_{Osc} .

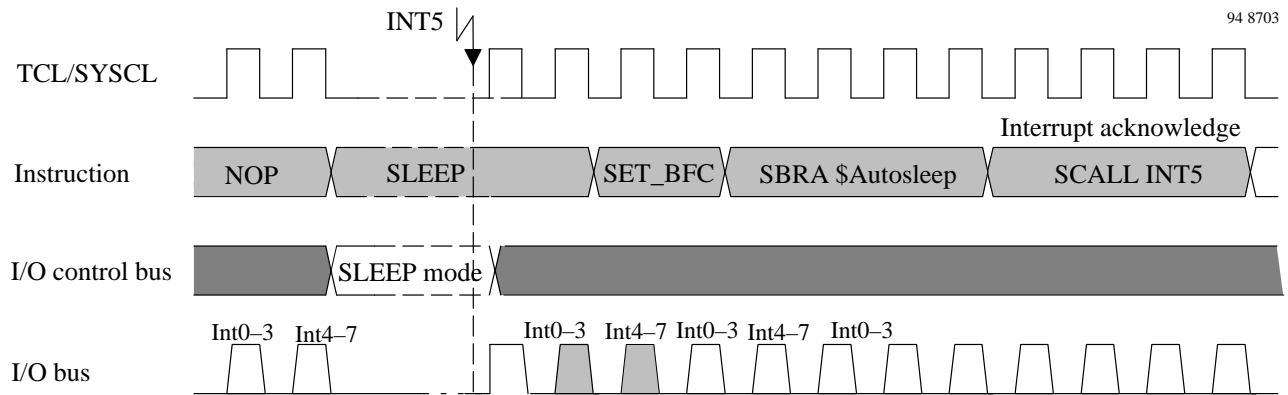


Figure 10 Timing sleep mode

1.6 Peripherals

There are various on-chip peripheral modules like I/O ports, timer specific configurations, counters, EEPROM, etc. which can be combined with the core for special application. A selection of MARC4 standard peripherals are described in the following sections.

1.6.1 Addressing peripherals

The I/O ports and the registers of the peripheral modules are I/O mapped. The MARC4 IN and

OUT instructions allow via the I/O bus a direct read or write access to one of the 16 I/O addresses. For peripherals with a large size of registers the extended addressing mode is used. With two I/O operations an extended I/O port allows the access to 16 subports. The first OUT instruction writes the subport address to the subaddress register the second IN or OUT instruction reads data from or writes data to the addressed subport.

Table 5: I/O addressing

I/O operation	qFORTH instructions	Description
I/O read	Port IN	Read data from port
I/O write	Data port OUT	Write data to port
Extended I/O read	Subaddress port OUT	Write subaddress to port
	Port IN	Read data from subaddress
Extended I/O write	Subaddress port OUT	Write subaddress to port
	Data port OUT	Write data to subaddress
Extended I/O read (byte)	Subaddress port OUT	Write subaddress to port
	Port IN	Read data high nibble from subaddress
	Port IN	Read data low nibble from subaddress
Extended I/O write (byte)	Subaddress port OUT	Write subaddress to port
	Data port OUT	Write data low nibble to subaddress
	Data port OUT	Write data high nibble to subaddress

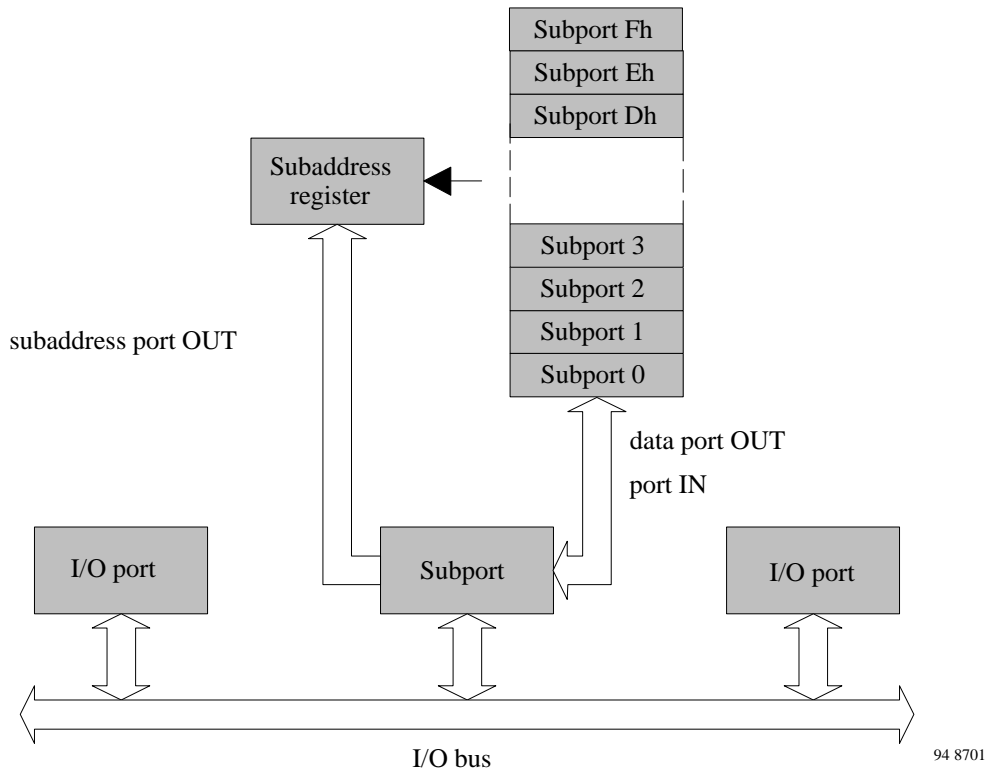


Figure 11 Extended I/O addressing

1.6.2 Input ports

Each input port is 4-bit wide. The data on the addressed port is read-in onto the TOS when an IN instruction is executed. The port address is

transferred from the TOS onto the I/O bus and is replaced by the data read from the addressed port.

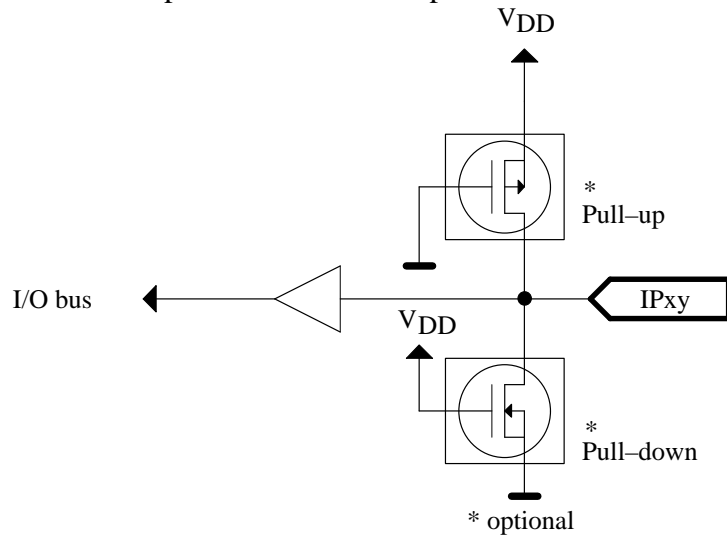


Figure 12 Input port

Input port options:

Integrated input pull-up/pull-down resistor

Output ports are 4-bit wide. The OUT-instruction takes two nibbles from the stack and put them on the I/O bus. The first nibble is used as port address and the second nibble is the output data. This data is latched in the addressed port.

1.6.3 Output port

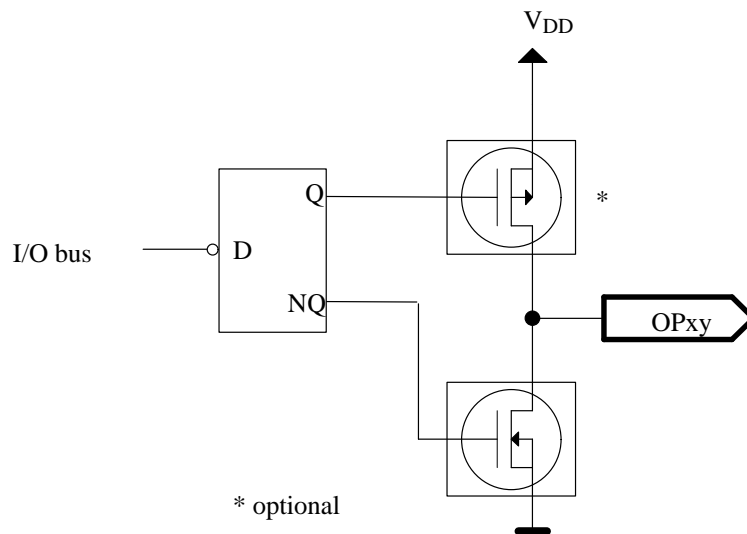


Figure 13 Output port

Output port options:

Open drain / complementary output

1.6.4 Bidirectional ports

The bidirectional ports are 4-bit wide and may be used for data input or output. The data direction is programable for a complete port only. The port is switched to output with an OUT instruction and to input with an IN instruction.

The data written to a port will be stored into the output latches and appears immediately after the OUT instruction at the port pin.

After RESET all output latches are set to Fh and the ports are switched to input mode.

Note:

Care must be taken when switching bidirectional ports from output to input. The capacitive load at this port may cause the data

read to be the same as the last data written to this port. To avoid this, when switching the direction one of the following approaches should be used.

Use two IN instructions and DROP the first data nibble read. The first IN switches the port from output to input, DROP removes the first invalid nibble and the second IN reads the valid nibble.

Use an OUT instruction followed by an IN instruction. With the OUT instruction the capacitive load is charged or discharged depending on the optional pull-up/pull down configuration. Write a '1' for pins with pull-up resistors and a '0' for pins with pull-down resistors.

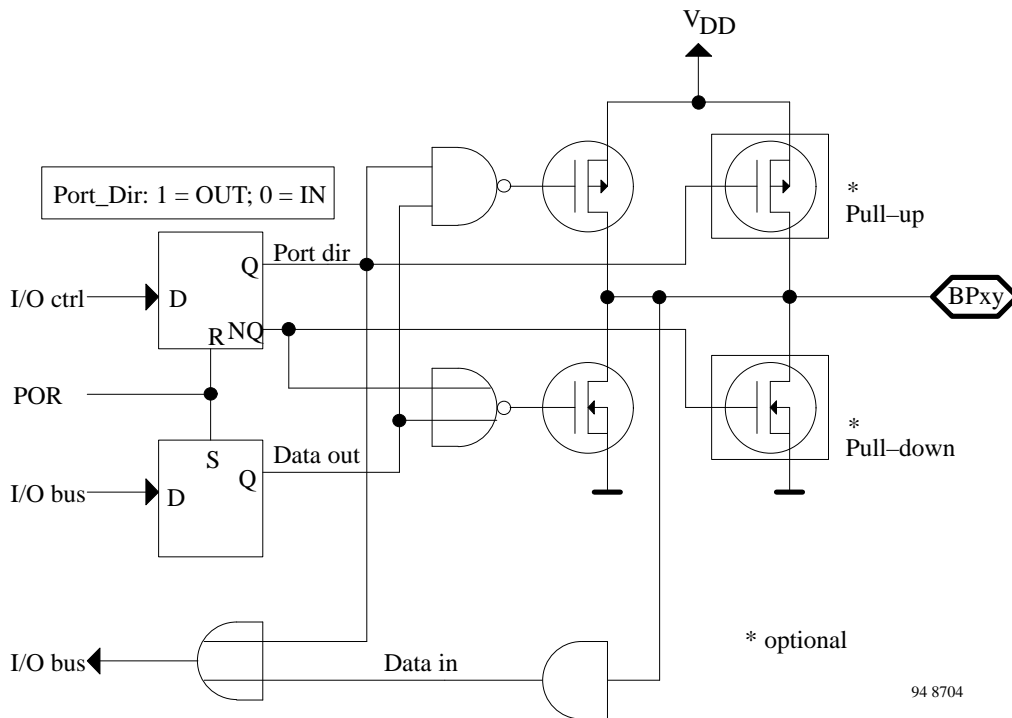


Figure 14 Bidirectional port

Bidirectional port options:

- Integrated input pull-up / pull-down resistor
- Open drain / complementary output

1.6.5 External interrupt inputs

All external interrupt have Schmitt trigger inputs and are edge triggered. The external interrupt logic contains interrupt latches and

stages to synchronize an external interrupt event with the internal timing.

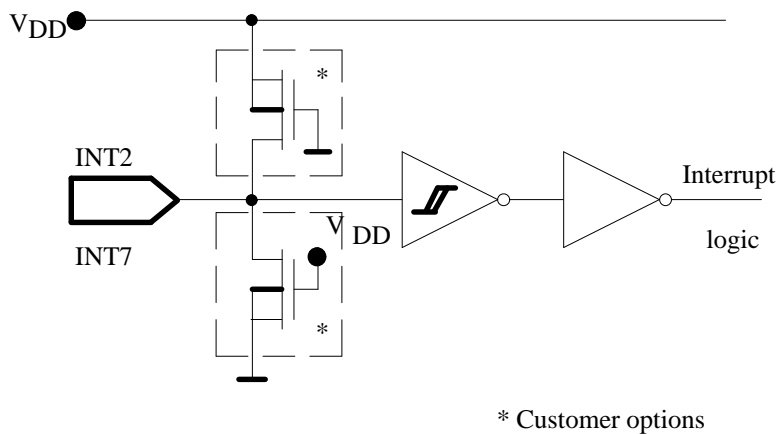


Figure 15 External interrupt inputs

Interrupt input options:

Integrated pull-up / pull-down resistor

1.7 Emulation

The basic function of emulation is to test and evaluate the customer's program and hardware in real time. Thus permits the analysis of any timing-, hardware- or software problem.

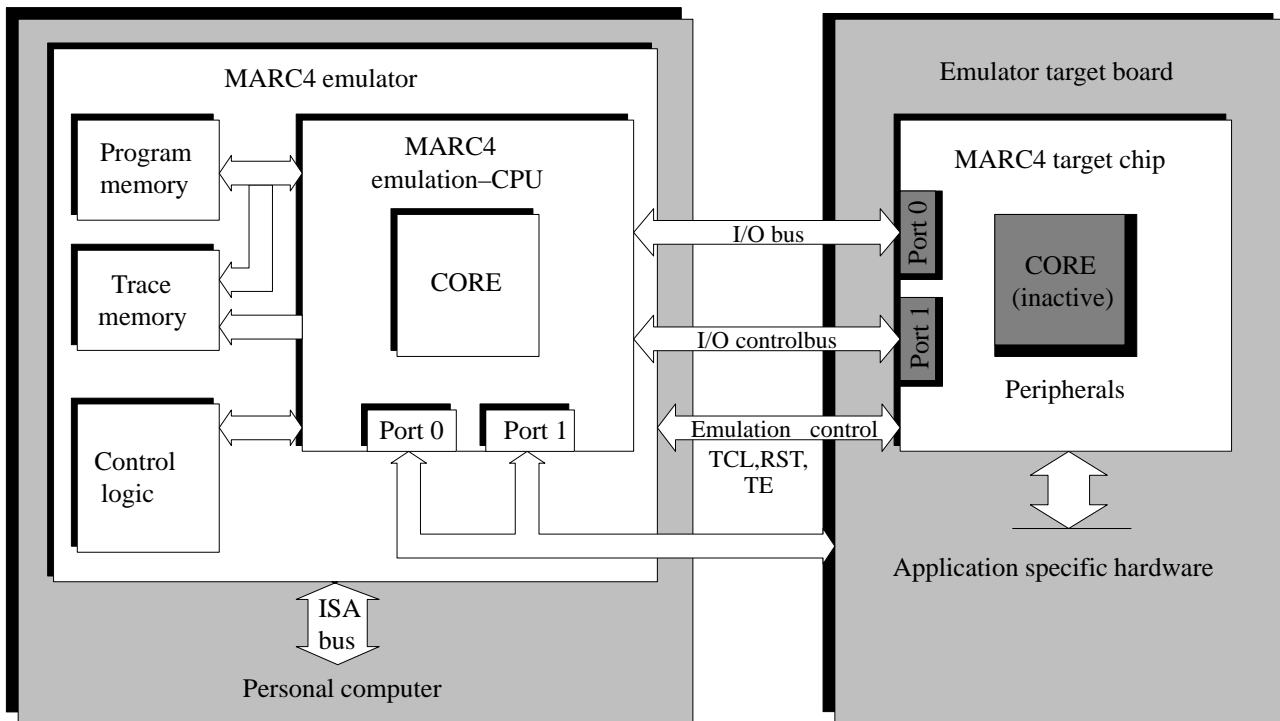
For emulation purposes all MARC4 controllers include a special emulation mode. In this mode the internal CPU core is inactive and the I/O buses are available via port 0 and port 1 to allow an external access to the on-chip peripherals. The MARC4 emulator uses this mode to control the peripherals of any MARC4 controller (target chip) and emulates the lost ports for the application.

A special evaluation chip (EVC) with a MARC4 core, additional breakpoint logic and program memory interface takes over the core function

and executes the program from an external RAM on the emulator board.

The MARC4 emulator can stop and restart a program at specified points during execution, making it possible for the applications engineer to view the memory contents and those of various registers during program execution. The designer also gains the ability to analyze the executed instruction sequences and all the I/O activities.

The emulator is a plug-in board for a PC (IBM-AT) with comfortable PC user interface software. It is independent from different MARC4 peripheral configurations and the customers hardware. For more information about emulation see "Emulator Manual"



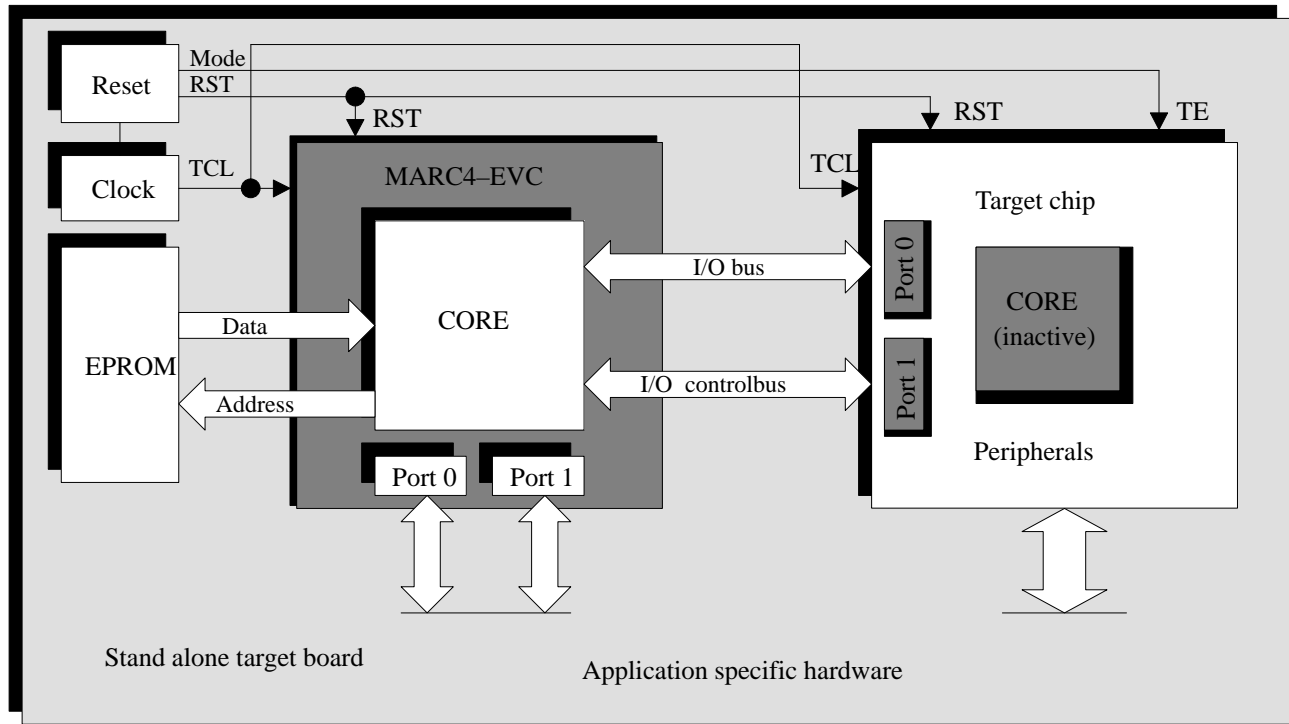
94 8700

Figure 16 MARC4 emulation

1.6.7.1 Stand alone EPROM boards

The evaluation chip (EVC) is used with an external EPROM to build stand alone and mobile customer prototype boards. The emulation CPU executes the program in the

EPROM and controls the peripherals of the customer MARC4's or prototype peripheral modules with discrete standard CMOS parts. Thus permitting to use various peripheral modules before they are available as integrated MARC4 on-chip peripherals.(see Figure 17).



94 8702

Figure 17 Stand alone target board

MARC4 emulation mode and interface signals

The MARC4 emulation mode can be activated by special test pins. There are MARC4 types with either the TST1 and TST2 pin or the TE pin. The emulation mode is activated with the TCL and TE pin or with TCL, TST1 and TST2 pin on the following way.

TCL	low during reset
TE	high
TST1	high
TST2	low

After the emulation mode is activated the core is switched off. The pins of Port 0 and Port 1 are used for the I/O bus and the I/O control signals as external interface to the on chip peripherals. The access to peripherals can be controlled by the four controlsignals at port 1 and the external clock at TCL pin. MARC4 types with OD pin needs additional a signal at this pin to control the data direction of port 1. The following table shows all interface signals which are available during the emulation mode.

Table 3: MARC4 emulation mode interface signals

Port 0				Port 1					
BP03	BP02	BP01	BP00	BP13	BP12	BP11	BP10	TCL	OD
I/O Bus3	I/O Bus2	I/O Bus1	I/O Bus0	N_Hold	N_Write	N_Read	N_Cycle	SYSCL	Direction

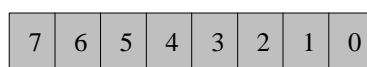
For more and detailed information about emulation refer to the MARC4 emulator manual.

1.8 MARC4 opcode formats

There are five types of instruction formats with a length of one and two bytes.

- 1) Zero address operation
(ADD,SUB, INC, OR,...)

Opcode



94 8708

- 2) Immediate data operation
Literal (LIT_0, LIT_1, ...)

Opcode

4 bit data



- 3) Short ROM address operation
(SCALL, SBRA)

Opcode

6 bit address



- 4) Long ROM address operation
(CALL, BRA)

Opcode

11



12 bit address



0



- 5) RAM address operation
(>SP, >X,...)

Opcode

8 bit address



1.9 MARC4 instruction set overview

Mnemonic	Description	Cycles /Bytes
Arithmetic operations:		
ADD	Add	1/1
ADDC	Add with carry	1/1
SUB	Subtract	1/1
SUBB	Subtract with borrow	1/1
DAA	Decimal adjust	1/1
INC	Increment TOS	1/1
DEC	Decrement TOS	1/1
DECR	Decrement 4-bit index on return stack	2/1
Compare operations:		
CMP_EQ	Compare equal	1/1
CMP_NE	Compare not equal	1/1
CMP_LT	Compare less than	1/1
CMP_LE	Compare less equal	1/1
CMP_GT	Compare greater than	1/1
CMP_GE	Compare greater equal	1/1
Logical operations:		
XOR	Exclusive OR	1/1
AND	AND	1/1
OR	OR	1/1
NOT	1's complement	1/1
SHL	Shift left into carry	1/1
SHR	Shift right into carry	1/1
ROL	Rotate left through carry	1/1
ROR	Rotate right through carry	1/1

Mnemonic	Description	Cycles /Bytes
Flag operations:		
TOG_BF	Toggle branch flag	1/1
SET_BFC	Set branch flag	1/1
DI	Disable all interrupts	1/1
CCR!	Store TOS into CCR	1/1
CCR@	Fetch CCR onto TOS	1/1
Program branching :		
BRA \$xxx	Conditional long branch	2/2
CALL \$xxx	Long call (current page)	3/2
SBRA \$xxx	Conditional short branch	2/1
SCALL \$xxx	Short call (zero page)	2/1
EXIT	Return from subroutine	2/1
RTI	Return from interrupt	2/1
SWI	Software interrupt	1/1
SLEEP	Activate sleep mode	1/1
NOP	No operation	1/1
Register operations:		
SP@	Fetch the current SP	2/1
RP@	Fetch the current RP	2/1
X@	Fetch the contents of X	2/1
Y@	Fetch the contents of Y	2/1
SP!	Move the top 2 into SP	2/1
RP!	Move the top 2 into RP	2/1
X!	Move the top 2 into X	2/1
Y!	Move the top 2 into Y	2/1
>SP \$xx	Store direct address to SP	2/2
>RP \$xx	Store direct address to RP	2/2
>X \$xx	Store direct address into X	2/2
>Y \$xx	Store direct address into Y	2/2

Mnemonic	Description	Cycles /Bytes
Stack operations:		
SWAP	Exchange the top two nibble	1/1
OVER	Copy TOS-1 to the top	1/1
DUP	Duplicate the top nibble	1/1
ROT	Move TOS-2 to the top	3/1
DROP	Remove the top nibble	1/1
>R	Move the top nibble onto the return stack	1/1
2>R	Move the top 2 nibble onto the return stack	3/1
3>R	Move the top 3 nibble onto the return stack	4/1
R@	Copy 1 nibble from the return stack	1/1
2R@	Copy 2 nibbles from the return stack	2/1
3R@	Copy 3 nibbles from the return stack	4/1
DROPR	Remove the top of return stack (12 bit)	1/1
LIT_n	Push immediate value (1nibble) onto TOS	1/1
ROM data operations:		
TABLE	Fetch 8 bit constant from ROM	3

Mnemonic	Description	Cycles /Bytes
Memory operations:		
[X]@ [Y]@	Fetch 1 nibble from RAM indirect addressed by X- or Y-Register	1/1
[+X]@ [+Y]@	Fetch 1 nibble from RAM indirect addressed by pre-incremented X- or Y-register	1/1
[X-]@ [Y-]@	Fetch 1 nibble from RAM indirect addressed by post-decremented X- or Y-register	1/1
[>X]@ \$xx [>Y]@ \$xx	Fetch 1 nibble from RAM direct addressed by X- or Y-register	2/2
[X]! [Y]!	Store 1 nibble into RAM indirect addressed by [X]	1/1
[+X]! [+Y]!	Store 1 nibble into RAM indirect addressed by pre-incremented [X]	1/1
[X-]! [Y-]!	Store 1 nibble into RAM indirect addressed by post-decremented X- or Y-register	1/1
[>X]! \$xx [>Y]! \$xx	Store 1 nibble into RAM direct addressed by X- or Y-register	2/2
I/O operations:		
IN	Read I/O port onto TOS	1/1
OUT	Write TOS to I/O port	1/1

I. Hardware Description

II. Programming in qFORTH

III. Instruction Set

IV. qFORTH Language Dictionary

Addresses

2. Programming in qFORTH

Why program in qFORTH ?

Programming in qFORTH reduces the software development time !

TEMIC's strategy in developing an integrated programming environment for qFORTH was to free the programmer from restrictions imposed by many FORTH environments (e.g. screen, fixed file block sizes), and at the same time to maintain an interactive approach to program development. Using the MARC4 software development system enables the MARC4 programmer to edit, compile, simulate and/or evaluate program code using an integrated package with predefined key codes and pull-down menus. The compiler generated MARC4 code is optimized for demanding application requirement, such as efficient usage of available program memory. One can be assured that the generated code only uses the amount of on-chip memory that is required, and that no additional overhead is attached to the program at the compilation phase.

What other reasons are there for programming in qFORTH ?

Subroutines that are kept short, increase the modularity and program maintainability. Both are related to development cost. Programs that are developed using the Brute Force approach (where the program is realized in software using sequential code) tend to be considerably larger in memory consumption, and are extremely difficult to maintain.

qFORTH programs, engineered using the building block modular approach are compact in size, easy to understand and thus, easier to maintain. The added benefit for the user is a library of software routines which can be interchanged with other MARC4 applications so long as the input and output conditions of your code block correspond. This toolbox of

off-the-shelf qFORTH routines grows with each new MARC4 application and reduces the amount of programming effort that you will require. Programming in qFORTH results in re-usable code. Re-usable for other applications that you will program at a later date. This is an important factor in ensuring that your future software development costs are kept manageable. Routines written by one qFORTH programmer can be easily incorporated by a different qFORTH user.

Langage features:

Expandability

Many of the fundamental qFORTH operations are directly implemented in the MARC4 instruction set.

Stack oriented

All operations communicate with one another via the data stack and use the reverse polish form of notation (RPN)

Structured programming

qFORTH supports structured programming

Reentrant

Different tasks can share the same code.

Recursive

qFORTH routines can call themselves.

Native code inclusion

In qFORTH there is no separation of high level constructs from the native code mnemonics.

2.1. Language overview

qFORTH is based upon the FORTH-83 language standard, the **qFORTH** compiler generates native code for a 4-bit FORTH-architecture single chip microcomputer, the **TEMIC MARC4**.

MARC4 applications are all programmed in **qFORTH** which is designed specifically for efficient real time control. Since the qFORTH compiler generates highly optimized code, there

is no advantage or point in programming the **MARC4** in assembly code. The high level of code efficiency generated by the qFORTH compiler is achieved by the use of modern optimization techniques such as branch-instruction size minimization, fast procedure calls, pointer tracking and many peephole optimizations.

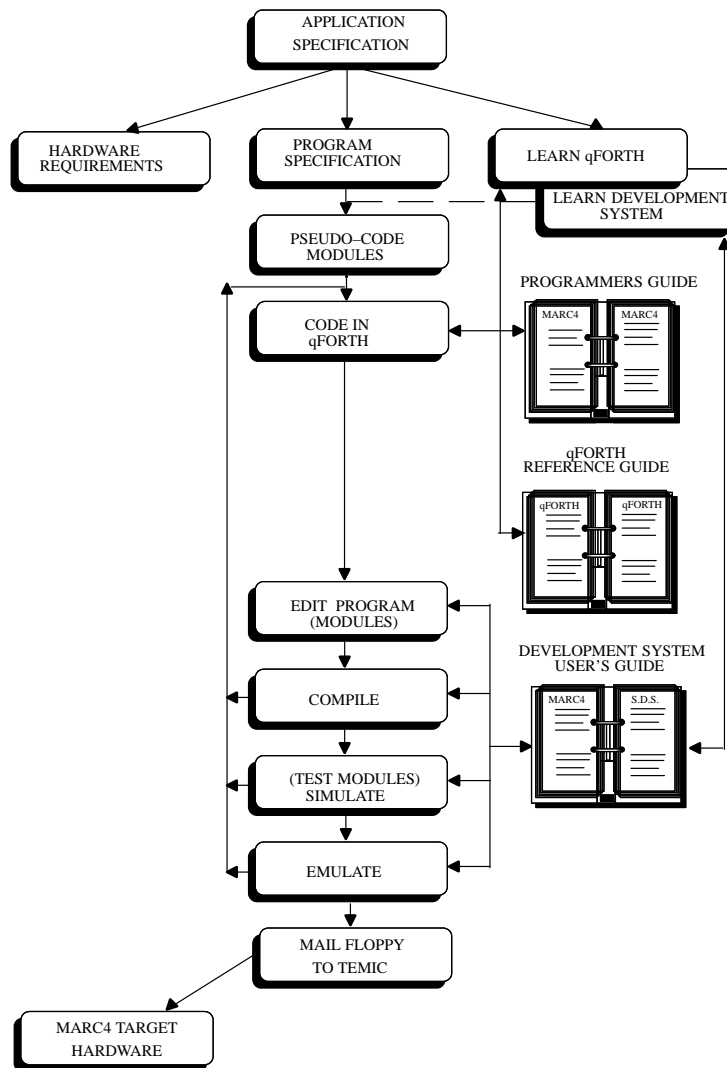


Figure 2.1 Program development with qFORTH

Standard FORTH operations which support string processing, formatting and disk I/O have been omitted from the qFORTH system library, since these instructions are not required in single-chip microcomputer applications.

The following two tables highlight the basic constructs and compare **qFORTH** with the **FORTH-83** language standard.

Table 1.2: qFORTH's FORTH-83 language subset

Arithmetic / logical	Stack operations
- D+ 1+ AND NEGATE + D- 1- NOT DNEGATE * 2* D2* OR / 2/ D2/ XOR	>R <ROT ?DUP OVER 2DUP I R> 2DROP DEPTH DUP PICK 2OVER DROP SWAP 2SWAP J ROT ROLL
Compiler	Control structure
ALLOT \$INCLUDE CONSTANT 2CONSTANT CODE END-CODE VARIABLE 2VARIABLE	?DO DO IS ELSE THEN +LOOP LEAVE UNTIL AGAIN ENDCASE LOOP WHILE BEGIN ENDOF OF CASE EXIT REPEAT EXECUTE
Comparison	Memory operations
< = <> <= >= 0= 0<> D> D0<> D< D0= D>= D= MIN MAX DMIN DMAX D<= D<>	! 2! @ 2@ ERASE MOVE MOVE > FILL TOGGLE

Table 2.2: Differences between qFORTH and FORTH-83

qFORTH	FORTH-83
4-bit expression stack 12-bit return stack The prefix "2" on a keyword (e.g. 2DUP refers to an 8-bit data type) Branch and carry flag in the condition code register Only predefined data types for handling untyped memory blocks, arrays or tables of constants	16-bit expression stack 16-bit return Stack The prefix "2" on a keyword (e.g. 2DUP refers to a 32-bit data type) Flag value on top of the expression stack CREATE, >BUILD .. DOES

2.2. The qFORTH vocabulary: words and definitions

qFORTH is a compiled language with a dictionary of predefined words. Each qFORTH word contained in the system library has, as its basis a set of core words which are very close to the machine level instructions of the MARC4 (such as **XOR**, **SWAP**, **DROP** and **ROT**). Other instructions (such as **D+** and **D+!**) are qFORTH word definitions. The qFORTH compiler parses the source code for words which have been defined in the system dictionary. Once located as being in the dictionary, the compiler then translates the qFORTH definition into MARC4 machine level instructions.

2.2.1. Word definitions

A new word definition which i.e. contains three sub-words: **WORD1**, **WORD2** and **WORD3** in a colon definition called **MASTER-WORD** is written in qFORTH as:

```
: MASTER-WORD WORD1 WORD2 WORD3 ;
```

The colon ':' and the semicolon ';' are the start and stop declarations for the definition. qFORTH programmers refer to a colon definition to specify a word name which follows the colon. The following diagram depicts the execution sequence of these three words:

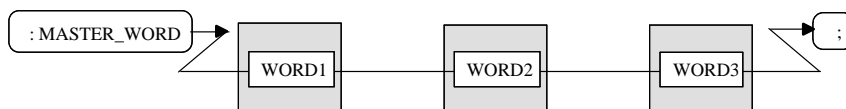


Figure 2.2 Threaded qFORTH word definition

The sequential order shows the way the compiler (and the MARC4) will understand what our program is to do.

1st step Begin the word definition with a ':', followed by a space.

2nd step Specify the <name> of the colon definition.

3rd step List the names of the sequentially organized words which will perform the definition. Remember that each word as shown above can itself be a colon or macro definition of other qFORTH words (such as **D+** or **2DUP**).

3rd step The qFORTH stack notation shows the stack contents before and after the execution of a qFORTH word. The separation of the before and the after operations is via two bars: — . The left hand side of the stack shows the stack before execution of the operation. The right most element before the two bars on the left side is the top of stack before the operation and the right most on the right side is also the top of stack after the operation. Examine the following qFORTH stack notation:

4th step Specify the end of the colon definition with a semicolon.

2.3. Stacks, RPN and comments

In this section we will look at the qFORTH notation known as **RPN**. Other topics to be examined include a look at qFORTH's stacks, constants and variables.

2.3.1. Reverse polish notation

qFORTH is a **Reverse Polish Notation** language (**RPN**), it operates on a stack of data values. **RPN** is a stack based representation of a mathematical problem where the top two numbers on the stack are operated upon by the operation to be performed.

Example:

4 + 2 Is spoken in the English language as '4 plus 2', resulting in the value 6. In our stack based **MARC4** we write this using qFORTH notation as:

4 2 + The first number, **4** must be placed onto the data stack, then the second number will follow it onto the data stack. The **MARC4** then comes to the addition operator. Both the **4** and **2** are taken off the data stack and processed by the **MARC4**'s arithmetic and logic unit, the result (in this case 6) will be deposited onto the top of the data stack.

2.3.2. The qFORTH stacks

The **MARC4** processor is a stack based microcomputer. It uses a hardware constructed storage area onto which data is placed in a last-in-first-out nature.

The **MARC4** has two stacks, the expression stack and the return stack.

The **expression stack**, also known as the data stack, is 4-bit wide and is used for temporary storage during calculations or to pass parameters to words.

The **return stack** is 12-bit wide and is used by the processor to hold the return addresses of subroutines, so that upon completion of the called word, program control is transferred back to the calling qFORTH word. The return stack is used by all colon definitions (i.e. **CALLs**), interrupts and to hold loop control variables.

2.3.3. Stack notation

The qFORTH stack notation shows the stack contents before and after the execution of a qFORTH word. The separation of the before and the after operations is via two bars: **—**. The left hand side of the stack shows the stack before execution of the operation. The right most element before the two bars on the left side is the top of stack before the operation and the right most on the right side is also the top of stack after the operation. Examine the following qFORTH stack notation:

before side	after side	Example	Stack notation
(n3 n2 n1 ↑ TOS	— n3 n2 n1 ↑ TOS	4 2 1 + SWAP	(— 4 2 1) (4 2 1 — 4 3) (4 3 — 3 4)

2.3.4. Comments

Comments in qFORTH are definitions which instruct the qFORTH compiler to ignore the text following the comment character. The comment is included in the source code of your program to aid the programmer in understanding what the code does'. There are two types of comment declarations:

qFORTH comment definitions	
Type _ 1 :	(text)
Type _ 2 :	\ text

Type_1 Comments begin and end with curved brackets, while **Type_2** comments

require only a backslash at the beginning of the comment. **Type_1** declarations do not require a blank space before the closing bracket.

Type_2 Comments start at the second space following the backslash and go till the end of the line. Both types of declarations require that a blank space follows the comment declaration.

Valid	Invalid
(this is a valid comment)	(this is not a valid comment)
\ this is a valid comment	\\ this is not a valid comment

2.4. Constants and variables

In qFORTH data is normally manipulated as unsigned integer values, either as memory addresses or as data values.

2.4.1. Constants

A constant is an un-alterable qFORTH definition. When it's once defined, the value of the constant cannot be altered. In qFORTH 4-bit and 8-bit numerical data can be assigned to a more readable symbolic representation.

qFORTH constant definitions	
value CONSTANT <<constant-name>>	(4-bit constant)
value 2CONSTANT <<constant-name>>	(8-bit constant)

Example:

```
7   CONSTANT   Set-Mode
42h 2CONSTANT  ROM_Okay

: Load-Answer   ROM_Okay ; ( Places 42h
                        on EXP
                        stack)
```

2.4.1.1. Predefined constants

In the qFORTH compiler a number of constants have a predefined function.

\$ROMSIZE

2CONSTANT to define the MARC4's actual ROM size. The values are **1.5K** (default), **2.0K**, **2.5K**, **3.0K** and **4.0K**bytes of ROM.

\$RAMSIZE

2CONSTANT to define the MARC4's actual RAM size in nibbles. Possible values are **111** (default), **167** and **255** nibbles.

\$EXTMEMSIZE

Allows the programmer to define the size of an external memory. Only required, if an external memory is used whereby the default value is set to 255 nibbles.

\$EXTMEMPORT

Allows the definition of a port address via which the external memory is accessed. The default port address for external memory is **Fh**.

\$EXTMEMTYPE

Allows the definition of the type of external memory used. The types **RAM** or **EEPROM** are valid, whereby **RAM** is default if an external memory is used.

Example:

```

6      CONSTANT $EXTMEMPORT
RAM CONSTANT $EXTMEMTYPE
95     2CONSTANT $EXTMEMSIZE
16     2ARRAY    Frequency
                EXTERNAL
: Check_Freq   Frequency [4] 2@ 80h D>=
                IF 0 0      Frequency [5]
                        2! THEN ;
    
```

2.4.2. Look-up tables

Look-up tables of 8-bit bytes are defined by the word **ROMCONST**, followed by the *<table-name>* and a list of single or double-length constants delimited by commas.

The contents of a table is not limited to literals such as 5 or 67h, but may also include user- or predefined constants such as **Set-Mode** or **ROM_Okay**

In the examples below the days of the month are placed into a look-up table called 'Days Of Month', the month (converted to 0 ... 11) is used to access the table in order to return the BCD number of days in the given month.

qFORTH table definition	
ROMCONST <table-name>	Const , Const , Const ,
	Const , Const , Const ,

Examples:

```

ROMCONST DaysOfMonth 31h , 28h , 31h , 30h ,
                    31h , 30h , 31h , 31h ,
                    30h , 31h , 30h , 31h ,
ROMCONST DaysOfWeek  SU , MO , TU , WE ,
                    TH , FR , SA ,
ROMCONST Message    11 , " Hello World " ,
    
```

Note 1: A comma must follow the last table item.

Note 2: Since there is no end-of-table delimiter in qFORTH, only a colon definition, a **VARIABLE** or another **ROMCONST** may follow a table definition (i.e. the last comma).

2.5. Variables and arrays

A variable is a qFORTH word whose name is associated with a memory address. A value can be stored at the memory address by assigning a value to the named variable. The value at this address can be accessed by using the variable name, thereby placing the variable value onto the top of the stack.

The **VARIABLE** definition has a 4-bit memory cell allocated to it. qFORTH permits a double-length 8-bit value also to be assigned as a **2VARIABLE**.

qFORTH variable definitions	
VARIABLE <variable-name>	4-bit variable
2VARIABLE <variable-name>	8-bit variable

Example:

```
VARIABLE Relay#
2VARIABLE Voltage
```

2.5.1. Defining arrays

qFORTH arrays are declared differently from arrays in FORTH-83. In both implementations

of FORTH an array is a collection of elements assigned to a common name. An array can either be defined as being a **VARIABLE** with 8 elements as:

VARIABLE DATA 7 ALLOT

or using the qFORTH array implementation:

8 ARRAY DATA

The array index is running from 0 to <length-1>.

qFORTH array definitions	
ARRAY	allocates RAM space for a short 4-bit array
LARRAY	allocates RAM space for a long 4-bit array
2ARRAY	allocates space for a short 8-bit array
2LARRAY	allocates space for a long 8-bit array

ARRAY and **2ARRAY** may contain up to 16 elements (e.g. nibbles or bytes). **LARRAY** and **2LARRAY** contain more than 16 elements.

2.6. Stack allocation

Both the expression and return stacks are located in RAM. The size of the stacks are variable and must be defined by the programmer by using the predefined variables **R0** and **S0**. In figure 2.3 the location of the stacks in RAM is shown. The return stack variable address **R0** starts at RAM location 00h. The expression stack is located above the return stack starting at the next location called **S0**.

The depth of the expression and return stacks are allocated using the **ALLOT** construct. While the depth (in nibbles) of the expression stack is exactly the number allocated, the return stack depth is expressed by the following formula:

$$\text{RET_Value} := (\text{RET_Depth} - 2) * 4 + 1$$

Example:

```
VARIABLE R0 20 ALLOT \ RET Depth of 7
VARIABLE S0 17 ALLOT \ EXP Depth of 17
```

2.6.1. Stack pointer initialisation

The two stack pointers must be initialized in the **\$RESET** routine.

Note: Return stack pointer **RP** must be set to **FCh** so that the **AUTOSLEEP** feature will work.

Example:

```
VARIABLE R0 32 ALLOT \ RET stack depth = 10
VARIABLE S0 12 ALLOT \ EXP stack depth = 12
                        nibbles

: $RESET
  >RP FCh              \ Initialize the two stack
                        pointers
  >SP S0
  RAM_Test
  ...
;
```

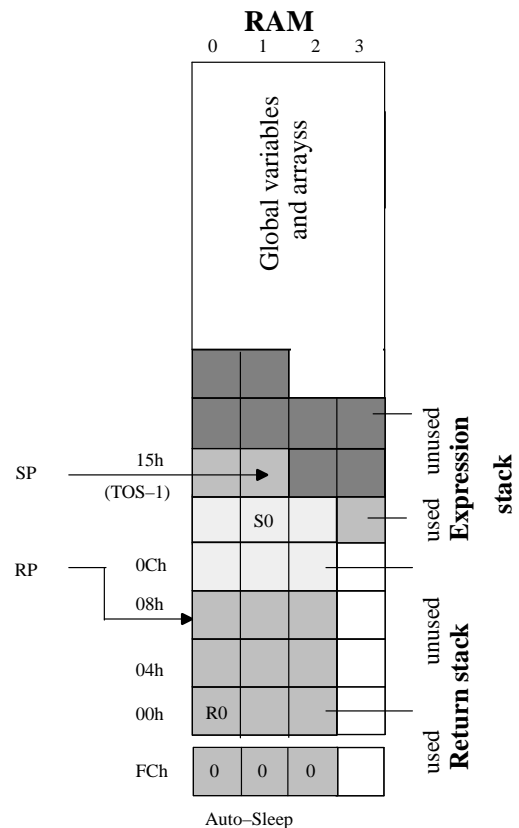


Figure 2.3 Stacks inside RAM

2.7. Stack operations, reading and writing

2.7.1. Stack operations

A number of stack operators are available to the qFORTH programmer. An overview of all the predefined stack words can be found in the **qFORTH Quick Reference Guide**. The most often used stack operators which manipulate the order of the elements on the data stack like **DUP**, **DROP**, **SWAP**, **OVER** and **ROT** are explained later on.

2.7.1.1. The data stack

The 4-bit wide data stack is called the expression stack. Arithmetic and data manipulation are performed on the expression stack. The expression stack serves as a holding device for the data and also as the interface link between words, in that all data passed between the qFORTH words can be located on the expression stack or in global variables.

The qFORTH word

```
: TEN 1 2 3 4 5 6 7 8 9 0 ;
```

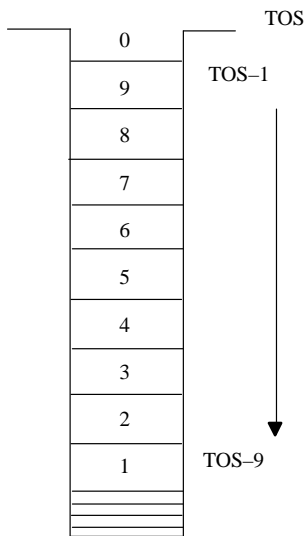


Figure 2.4 Push down data stack

When executed, will have the value **0** at the top and the value **1** at the bottom of the expression stack.

2.7.1.2. SWAP

In many programming applications it is necessary to re-arrange the input data so that it can be handled properly. For example we will use a simple series of data and then **SWAP** them so that they appear in the reverse order.

```
4 2 SWAP ( 4 2—2 4)
```

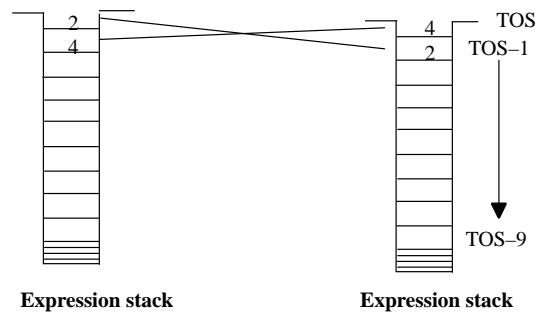


Figure 2.5 The SWAP operation

2.7.1.3. DUP, OVER and DROP

The qFORTH word to duplicate the TOS item is **DUP**. It will make a copy of the current TOS element on the expression stack.

DUP is useful in retaining the TOS value before operations which implicitly **DROP** the TOS following their execution. For example, all of the comparison operations like **>**, **>=**, **<=** or **<** destroy the TOS.

The **OVER** operation makes a copy of the second element on the stack (TOS-1) and deposits it onto the top of the stack.

The MARC4 stack operator **DROP** removes one 4-bit value from the TOS. For example, the qFORTH operation **NIP** will drop the TOS-1 element from the stack. This can be written in qFORTH as:

```
: NIP SWAP DROP ; ( n1 n2 — n2)
```

2.7.1.4. ROT and <ROT

Stack values must frequently be arranged into a defined order. We have already been introduced to the **SWAP** operation. Besides **SWAP**, qFORTH supports the stack rotation operators **ROT** and **<ROT**.

The **ROT** operation moves the third value (TOS-2) to the TOS. The operation **<ROT** (which is the same as **ROT ROT**) does the opposite of **ROT**, moving the value from the TOS to the TOS-2 location on the expression stack.

2.7.1.5. R>, >R, R@ and DROPR

qFORTH also supports data transfers between the expression and the return stack.

The **>R** operation moves the top 4-bit value from the expression stack and pushes the value onto the return stack. **R>** removes the top 4-bit value from the return stack and puts the value onto the expression stack, while **R@** (or **I**) copies the 4-bit value from the return stack and deposits the copied value onto the expression stack. **DROPR** removes the top entry from the return stack.

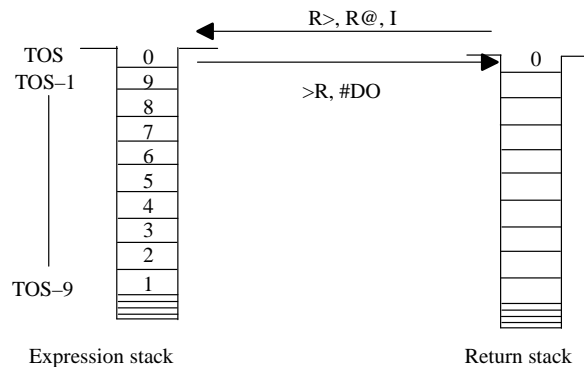


Figure 2.6 Return stack data transfers

2.7.1.6. Other useful stack operations

The following list contains more useful stack operations. Note that for every 4-bit stack operation there is almost always an 8-bit equivalent. A full list of all stack operations may be found in the **qFORTH Quick Reference Guide**.

' <name>	EXP (— ROMAddr)	Places ROM address of colon-definition <name> on EXP stack
<ROT	EXP (n1 n2 n — n n1 n2)	Move top value to 3rd stack pos.
?DUP	EXP (n — n n)	Duplicate top value, if n <>0
I	EXP (— I)	Copy 4-bit loop index I from the return to the expression stack
R@	RET (u u I — u u I)	
NIP	EXP (n1 n2 — n2)	Drop second to top 4-bit value
TUCK	EXP (n1 n2 — n2 n1 n2)	Duplicate top value, move under second item
2>R	EXP (n1 n2 —) RET (— u n2 n1)	Move top two values from expression to return stack
2DROP	EXP (n1 n2 —)	Drop top 2 values from the stack
2DUP	EXP (d — d d)	Duplicate top 8-bit value
2NIP	EXP (d1 d2 — d2)	Drop 2nd 8-bit value from stack
2OVER	EXP (d1 d2 — d1 d2 d1)	Copy 2nd 8-bit value over top value
2<ROT	EXP (d1 d2 d — d d1 d2)	Move top 8-bit value to 3rd pos'n
2R>	EXP (— n1 n2) RET (u n2 n1 —)	Move top 8-bits from return to expression stack
2R@	EXP (— n1 n2) RET (u n2 n1 — u n2 n1)	Copy top 8-bits from return to expression stack
3>R	EXP (n1 n2 n3 —) RET (— n3 n2 n1)	Move top 3 nibbles from the expression onto the return stack
3DROP	EXP (n1 n2 n3 —)	Remove top 12-bit value from stack
3DUP	EXP (t — t t)	Duplicate top 12-bit value
3R>	EXP (— n1 n2 n3) RET (n3 n2 n1 —)	Move top 3 nibbles from return to the expression stack
3R@	EXP (— n1 n2 n3) RET (n3 n2 n1 — n3 n2 n1)	Copy 3 nibbles (1 ROM address entry) from the return stack to the expression stack

2.7.2. Reading and writing (@, !)

In the previous section we mentioned that data can be placed onto and taken off of the expression stack.

The reading and writing operations transfer data values between the data stack and the RAM. Writing a data value to a RAM location which has been specified by a variable name requires that the TOS contains the variable's 8-bit RAM address and that the data to be stored in the RAM be contained at the TOS-2 location.

Read is written in the qFORTH syntax with the @ symbol and is pronounced *Fetch*. The write operator is written in qFORTH with the ! symbol and is pronounced *Store*.

Write two qFORTH colon definitions (words) that will store the numeric value 7 from the TOS to the variable named FRED and then fetch the contents of it back onto the expression stack (TOS).

Example:

```
VARIABLE FRED
: Store    7 FRED ! ;      ( — )
: Fetch    FRED @ ;      ( — n)
```

For 8-bit values, stored at two consecutive locations, qFORTH has the *Double-Fetch* and *Double-Store* words: **2@** and **2!**. To store 1Ah in

the 8-bit 2VARIABLE BERT using the *Double-Store*, examine the following code:

```
2VARIABLE BERT
: Double-Store  1Ah BERT 2! ;
```

Storing the value 1Ah is a two part operation: The high order nibble 1 is stored in the first digit, while at the next 4-bit RAM location the hexadecimal value A will be stored.

```
: Double-Fetch  BERT 2@ ;      (— d)
```

i.e. accesses the 8-bits at the memory address where BERT is placed and load them onto expression stack. The lower order nibble will always end up on TOS.

Note: Hexadecimal values are represented using **h** or **H** following the value.

2.7.3. Low level memory operations

2.7.3.1. RAM address registers X and Y

The MARC4 processor can address any location in RAM indirectly via the 8-bit wide X and Y RAM address registers. These registers are used as pointer registers to organise arrays within the RAM, under CPU control they can be pre-incremented or post-decremented.

The X and Y registers are automatically used by the compiler during fetch (@) and store (!) operations. Hence, care should be taken when referencing these registers explicitly. By default the compiler uses the Y register.

X Register	Description	Y Register
X@	Fetch current X (or Y) register contents	Y@
X!	Move 8-bit address from stack into X (or Y) reg.	Y!
>X xx	Set register address of X (or Y) direct	>Y yy
[>X]@ xx	Direct RAM fetch, X (or Y) addressed	[>Y]@ yy
[>X]! xx	Direct RAL store, X (or Y) addressed	[>Y]! yy
[X]@	Indirect X (or Y) fetch of RAM contents	[Y]@
[X]!	Indirect X (or Y) store of RAM contents	[Y]!
[+X]@	Pre-increment X (or Y) indirect RAM fetch	[+Y]@
[X-]@	Postdecrement X (or Y) indirect RAM fetch	[Y-]@
[+X]!	Pre-increment X (or Y) indirect RAM store	[+Y]!
[X-]!	Postdecrement X (or Y) indirect RAM store	[Y-]!

Memory operators which use the Y			X Register
@	D+!	2!	ERASE
!	D-!	2@	FILL
+!	TD+!	3!	MOVE
1+!	TD-!	3@	MOVE>
-!	T+!	PICK	TOGGLE
1-!	T-!	ROLL	DTOGGLE

Example:

The 4-bit value in TOS is added to an 8-bit RAM value and stored back into the 8-bit RAM variable.

```
: M+! ( n RAM_addr — )
  X! [+X]@ + [X-]!
  0 [X]@ +C [X]!
;
```

```
2VARIABLE Voltage
5 Voltage M+!
```

2.7.3.2. Bit manipulations in RAM

Using the X or Y registers it is possible to manipulate the contents of the RAM on a bitwise basis. The following examples have all the same stack notation.

```
: BitSet ( mask RAM_addr — [branch flag] )
  X! [X]@ ( get data from memory )
  OR [X]! ( mask & store in memory )
;
: BitReset ( mask RAM_addr — [branch flag] )
  X!
  Fh XOR ( Invert mask for AND )
  [X]@ ( get data from memory )
  AND [X]! ( mask & store in memory )
;
: Test0= ( mask RAM_addr — [branch flag] )
  X! [X]@
  AND DROP
;
CODE Test0<> ( mask RAM_addr — [branch flag] )
  Test0= TOG_BF
END-CODE
```

2.8. MARC4 condition codes

The MARC4 processor has within its arithmetic logic unit (ALU) a 4-bit wide condition code register (CCR) which contains 4 flag bits. These are the branch (**B**) flag, the interrupt-enable (**I**) flag and the carry (**C**) flag.

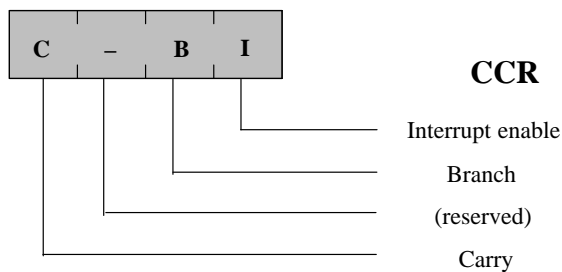


Figure 2.7 MARC4 condition code register flags

For example most arithmetic/logical operations will have an effect on it. If you try to add **12** and **5**, the carry and branch flags will be set, since an arithmetic overflow has occurred.

2.9. Arithmetic operations

The arithmetic operators which are presented here are similar to those which you will find described in most FORTH literature, the underlying difference however is that the qFORTH arithmetic operations are based upon the 4-bit CPU architecture of the MARC4.

2.9.1. Number systems

When coding in qFORTH standard numeric representations are decimal values, for other representations it is necessary to append a single character for that representation.

Example:

```
Bh  -> hexadecimal ( base 16 )
bH  -> hexadecimal ( base 16 )
11  -> decimal     ( base 10 )
1011b->binary     ( base 2  )
1011B->binary     ( base 2  )
```

2.8.1. The CCR and the control operations

The carry flag is set by ALU instructions such as the **+**, **+C**, **-** or **-C** whenever an arithmetic under/overflow occurs. The carry flag also is used during shift/rotate instruction such as **ROR** and **ROL**.

The branch flag is set under CPU control depending upon the current ALU instruction and is a result of the logical combination of the carry flag and the TOS=0 condition.

The branch flag is responsible for generating conditional branches. The conditional branch is performed when the branch flag was set by one of the previous qFORTH operations (e.g. comparison operations).

The **TOG_BF** instruction will toggle the state of the branch flag in the CCR. If the branch flag is set before the **TOG_BF** instruction, then it will be reset following the execution.

The **SET_BCF** instruction will set the branch and carry upon execution, while the **CLR_BCF** operation will reset both flags.

2.9.1.1. Single and double length operators

Examples have already been presented which perform operations on the TOS as a 4-bit (single-length) value or on both the TOS and TOS-1 values. By combining the TOS and TOS-1 locations it is possible to handle the data as an 8-bit value.

Note: In qFORTH all operators which start with a **2** (e.g: **2SWAP** or **2@**) use double length (8-bit) data. Other operators such as **D+** and **D=** are also double length operators.

The qFORTH language also permits triple length operators, these are defined with a **3** prefix (e.g: **3DROP**). Examples for all qFORTH dictionary words are included in the **qFORTH Language Reference Dictionary**.

2.9.2. Addition and subtraction

The algebraic expression $4 + 2$ is spoken in the English language as: *4 plus 2*, and results in a value of 6. In qFORTH we write this expression as `4 2 +`. The 4 is deposited onto the data stack, followed by the 2. The operator says to take the top two values from the data stack and add them together. The result is then placed back onto the data stack. Both the 4 and the 2 are dropped from the stack by the operation.

The stack notation for the addition operator is:

```
+ EXP ( n1 n2 — n1+n2 )
```

qFORTH performs the subtraction similar to the addition operator. The operator is the common algebraic symbol with the stack notation:

```
- EXP ( n1 n2 — n1-n2 )
```

Examples:

```
: TNEGATE      ( 12-bit 2's complement
                  on the TOS )
  0 SWAP -      ( th tm tl — th tm -tl )
  0 ROT -c      ( th tm -tl — th -tl -tm )
  ROT 0 SWAP -c ( th -tl -tm — tl -tm -th )
  SWAP ROT     ( -tl -tm -th — -t )
;
: 3NEG!        ( 12-bit 2's complement in
                  an array )
  Y! 0 [+Y]@ 0 [+Y]@ ( addr — 0 tm 0 tl )
  -[Y-]! -c [Y-]!   ( 0 tm 0 tl — )
  0 [Y]@ -c [Y]!   ( 0 tm -tl — )
;
```

2.9.3. Increment and decrement

Increment and decrement instructions are common to most programming languages, qFORTH supports both with the standard syntax:

```
1+ increment new-TOS: = old-TOS + 1
1- decrement new-TOS: = old-TOS - 1
```

Example:

```
: Inc-Dec      10      ( — Ah )
                1+      ( Ah — Bh )
                1-1- ;  ( Bh — 9h )
```

Note: The carry flag in the CCR is not affected by these MARC4 instructions, whereby the branch flag is set, if the result of the operation gets zero.

2.9.4. Mixed-length arithmetic

qFORTH supports mixed-length operators such as **M+**, **M-**, **M*** and **M/MOD**. In the examples below a 4-bit value is added/subtracted to/from an 8-bit value (generating an 8-bit result) using the **M+** and **M-** operators.

```
Voltage 2@ 5 M+
IF 2DROP 0 0 \ IF overflow, THEN reset Voltage
ELSE 10 M- THEN
Voltage 2!
```

2.9.5. BCD arithmetic

2.9.5.1. DAA and DAS

Decimal numbers are usually represented in 4-bit binary equivalents of each digit using the binary-coded-decimal coding scheme. The qFORTH instruction set includes the **DAA** and **DAS** operations for BCD arithmetic.

DAA Decimal adjust for BCD arithmetic, adds 6 to values between 10 and 15, or if the carry flag was set.

```
Fh ( 1111 ) -> 5 ( 0101 ) and carry flag set
Eh ( 1110 ) -> 4 ( 0100 )
Dh ( 1101 ) -> 3 ( 0011 )
Ch ( 1100 ) -> 2 ( 0010 )
Bh ( 1011 ) -> 1 ( 0001 )
Ah ( 1010 ) -> 0 ( 0000 )
```

DAS Decimal arithmetic for BCD subtraction, builds a 9's complement for DAA and ADDC, the branch and carry flags will be changed.

Examples:

```

: DIG-                                \ Digit count LSD_Addr —
  Y! SWAP  DAS SWAP                  \ Generate 9's complement
  #DO                                \ Digit count — Digit
    [Y]@ +  DAA [Y-]!                \ Transfer carry on stack
    10 -?LEAVE                        \ Exit LOOP, if NO carry
  #LOOP                               \ Repeat until index = 0
  DROP                                \ Skip TOS overflow digit
;
: BCD_1+!                              \ RAM_Addr —
  Y! [Y]@                             \ Increments BCD digit
  1 + DAA [Y]!                        \ in RAM array element
;
: Array_1+                              \ Inc BCD array by 1 ( n array[n] — )
  Y! SET_BCF                          ( Start with carry = 1 )
  BEGIN
    [Y]@ 0 +C DAA [Y-]!
    1-
  UNTIL
  DROP
;

```

2.9.6. Summary of arithmetic words

The following list contain more useful arithmetic words. The full list and implementation may be found in the **MATHUTIL.INC** file

D+	(d1 d2 — d_sum)	Add top two 8-bit elements
D-	(d1 d2 — d2-d1)	Subtract top two 8-bit elements
D+!	(nh nl addr —)	Add 8-bit TOS to memory
D-!	(nh nl addr —)	Subtract 8-bit TOS from memory
M+	(d1 n — d2)	Add 4-bit TOS to an 8-bit value
M-	(d1 n — d2)	Sub 4-bit TOS from 8-bit value
M+!	(n addr —)	Add n to an 8-bit RAM byte
M-!	(n addr —)	Subtract n from 8-bit RAM byte
M/	(d n — d_quotient)	Divide n from d
M*	(d n — d_product)	Multiply d by n
M/MOD	(d n — n_quot n_rem)	Div n from d giving 4-bit results
D/MOD	(d n — d_quot n_rem)	Div 8-bit value & 4-bit remainder
TD+!	(d addr —)	Add 8-bit TOS to 12-bit RAM var.
TD-!	(d addr —)	Sub 8-bit from 12-bit RAM var.
TD+	(d addr — t)	Add 8-bit to 12-bit RAM var.
TD-	(d addr — t)	Sub 8-bit from 12-bit RAM var.
D->BCD	(d — n_100 n_10 n_1)	Convert 8-bit binary to BCD

2.10. Logicals

The logical operators in qFORTH permit bit manipulation. The programmer can input a bit stream from the input port, transfer it onto the expression stack and then shift branches and the

bit pattern left or right, or the bit pattern can be rotated onto the TOS. The branch and carry flag in the CCR are used by many of the qFORTH logical operators.

2.10.1. Logical operators

The truth tables shown here are the standard tables used to represent the effects of the logical

operators on two data values (**n1** and **n2**).

NOT		OR		AND		XOR				
n1	n1'	n1	n2	n1 v n2	n1	n2	n1 ^ n2	n1	n2	n1 XOR n2
1	0	0	0	0	0	0	0	0	0	0
0	1	0	1	1	0	1	0	0	1	1
		1	0	1	1	0	0	1	0	1
		1	1	1	1	1	1	1	1	0

These qFORTH operators take the top values off of the expression stack and perform the desired logical operation. The resultant flag setting and the stack conditions are described in the **qFORTH Language Reference Dictionary**.

The stack notation for all logical qFORTH words is:

EXP (n1 n2 — n3)

As an example examine the logical **AND** operation with the data values 3 and 5. Representing these values in 4-bit binary, and performing the **AND** operator:

```
0101b 0011b      ( — 0101b 0011b      )
AND               ( 0101b 0011b — 0001b  )
```

results in a value of 1 appearing on the TOS. The branch flag will be reset, since the result of the logical operation is non-zero.

Example:

```
: Logicals
  3 7 OR      ( — 7 )
  3 7 AND     ( 7 — 7 3 )
  5 XOR      ( 7 3 — 7 6 )
  NOT       ( 7 6 — 7 9 )
  2DROP     ( 7 9 — )
;
```

A more complex example like the 16-bit CRC generation of the ROM checksum, is described in chapter 3.5.

2.10.1.1. TOGGLE

The TOGGLE operation is classified in the **qFORTH Language Reference Dictionary** as belonging to the set of *memory operations*, though this is true the TOGGLE and its relatives the DTOGGLE are all used to change bit patterns at a specified memory address. For the TOGGLE operation the 4-bit value located at the specified memory location will be exclusive-ORed.

Example:

```
VARIABLE LED_Status
: Toggle-LED
  0001b LED_Status TOGGLE ( toggles bit 0
                           only );
```

2.10.1.2. SHIFT and ROTATE operations

The MARC4 instruction set contains two shift and two rotate instructions shown in figure 2.8. The shift operators multiply (SHL) and divide (SHR) the TOS value by two. These instructions are identical to the qFORTH macros for 2* and 2/.

The rotate instructions ROR and ROL shift the TOS value right/left through the carry flag, will cause the carry and branch flags to be altered. When using these instructions it is advised to set or reset the flags within your initialization routine, using either the SET_BCF or the CLR_BCF instructions.

Mnemonic	Description	Function
SHR 2/	Shift TOS right into carry	
ROR	Rotate TOS right through carry	
SHL 2*	Shift TOS left into carry	
ROL	Rotate TOS left through carry	

Figure 2.8 Shift and rotate instructions

Example:

Write the necessary qFORTH word definitions to flip a data byte (located on TOS) as shown below:

```

Before flip: 3 2 1 0      After flip:  4 5 6 7
              7 6 5 4              0 1 2 3
: FlipBits
  0
  4 #DO
    SWAP  SHR
    SWAP  ROL
  #LOOP
  NIP
;
: FlipByte      FlipBits      SWAP      FlipBits ;

```

2.11. Comparisons

The qFORTH comparison operations (such as > or <) will set the branch flag in the CCR, if the result of the comparison is true. The stack effects of a comparison operation is:

EXP (n1 n2 —)

2.11.1. < , >

The qFORTH word < performs a less than comparison of the top two values on the stack. If the second value on the expression stack is less-than the value on the TOS, then the branch flag in the CCR will be set. Following the operation, the stack will contain neither of the two values which were checked, as they will be dropped from the expression stack.

```

: Less-Example 9 5 ( — 9 5 )
               < ; ( 9 5 — )

```

The > comparison operator determines if the second value on the stack is greater than the TOS

value. If this condition is met, then the branch flag will be set in the CCR.

2.11.2. <= , >=

Using <= in your qFORTH program enables you to determine if the second item on the stack is less-or-equal to the TOS value.

In the GREATER-EQUAL example, the top two stack values 5 and 9 will be removed from the stack and used as input values for the greater-or-equal operation. If the second value (TOS-1) is greater-or-equal the TOS value then the branch flag in the CCR will be set.

After the comparison operation has been performed by the MARC4 processor, neither of the two input values will be contained on the expression stack.

```

: GREATER-EQUAL 9 5 ( — 9 5 )
                >= ; ( 9 5 — [C-B-] )

```

2.11.3. <> , =

These two qFORTH comparison operators can be used to determine the Boolean (true/false) value (e.g. setting/ resetting the branch flag in the CCR). If the second value on the stack is not-equal (<>) to the TOS value, then the branch flag in the CCR will be set. The two values that were on the TOS before the operation, will be dropped off of the stack after the operation, has been executed, except that one or both items on the data stack were duplicated before the operation.

If however, the equality test (=) is executed, then the branch flag will only be set, if both the TOS and the TOS-1 values are identical, again, as with all the comparison operations presented thus far, the contents of the TOS and TOS-1 previous to the operations will be dropped from the stack.

2.12. Control Structures

The control structures presented here can be divided into two categories: Selection and looping. The tables below will compare

2.11.4. Comparisons using 8-bit values

Example:

```
68 2CONSTANT PAR-FOR-COURSE
    2VARIABLE GROSS-SCORE

: Check-golf-score
  GROSS-SCORE2@PAR-FOR-COURSE D-
  0 8 D<= IF GOOD-SCORE THEN
    \ My Handicap is 8
;
```

Note: There is a space between the 0 and 8. This is required because literals less than 16 are assumed to be 4-bit values.

This problem may be improved if an additional **2CONSTANT** is used, since **2CONSTANT** assumes an 8-bit value, e.g. :

```
8 2CONSTANT My-Handicap
: Check-Golf-score
  GROSS-SCORE2@PAR-FOR-COURSE D-
  My-Handicap D<=
  IF GOOD-SCORE THEN
;
```

qFORTH's control structures to those found in PASCAL.

Table 2.3 : qFORTH selection control structures

qFORTH	PASCAL
<condition> IF <operation> THEN	IF <condition> THEN <statements> ;
<condition> IF <operations> ELSE <operations> THEN	IF <condition> THEN <statements> ELSE <statements> ;
<value> CASE .. <n> OF <operations> END OF .. ENDCASE	CASE <value> .. <n> OF <statements> ; .. END ;

As the comparison between the two languages shows, qFORTH offers a rich variety of structures which will permit your program to

branch to different code segments within the program.

Table 2.2: qFORTH loop control structures

qFORTH	PASCAL
BEGIN <operations> <condition> UNTIL BEGIN <condition> WHILE <operations> REPEAT BEGIN <operations> AGAINb <limit> <start> DO <operations> LOOP <limit> <start> DO <operations> <offset> + LOOP <limit> <start> DO <operations> <condition> ?LEAVE <operations> LOOP <n-times> #DO <operations> #LOOP	REPEAT <statements> UNTIL <condition> ; WHILE <condition> DO <statements> ; FOR i := <start> TO <limit> DO <statements> ; FOR i := <start> DOWNTO 0 DO <statements> ;

2.12.1. Selection control structures

The code to be executed is dependant on a specific condition. This condition can be indicated by the setting of the branch flag in the CCR. The control operation sequences such as the **IF .. THEN** and the indefinite loop operations such as **BEGIN .. UNTIL** and **BEGIN .. WHILE .. REPEAT** will only be executed if the branch flag has been set.

2.12.1.1. IF .. THEN

The **IF .. THEN** construct is a conditional phrase permitting the sequence of program

statements to be executed dependant upon the **IF** condition being valid. The qFORTH implementation of the **IF .. THEN** phrase requires that the <condition> computation appears before the **IF** word.

IF .. THEN in PASCAL :

IF <condition> **THEN** < True statements> **ELSE** <False statements> ;

IF .. THEN in qFORTH :

<condition> **IF** <True operations> **ELSE** <False operations> **THEN**

Example:

```

: GREATER-9      ( n — n or 1, IF n > 9      )
  DUP 9 > IF
    DROP 1      ( THEN replace n — 1      )
  THEN          ( ELSE keep original n      )
;
:$RESET
  >SP S0        ( Power-on initialization entry )
  >RP FCh       ( Init both stack pointers first )
  10 Greater-9  ( Compare 10 > 9 ==> BF true   )
  5 Greater-9   ( 1 5 — 1 5                 )
  2DROP        ( 1 5 —                       )
;

```

The qFORTH word **GREATER-9** checks to see if the values given on TOS as a parameter to the word are greater than 9.

First we duplicate the current TOS value. Then 9 is deposited onto the TOS so that the value to be compared to is now in the TOS-1 and TOS-2 location of our data stack. The TOS value is now compared with the TOS-1 value. If TOS-1 is greater than 9 then the condition has been met, so the qFORTH words following the **IF** will be executed. In the first example the TOS value will be dropped and replaced by the value 1.

2.12.1.2. The CASE structure

The **CASE** structure is equivalent to the **IF .. ELSE .. THEN** structure. The **IF .. ELSE .. THEN** permits nested combinations to be constructed in qFORTH. A nested **IF .. ELSE .. THEN** structure can look like this example:

```
: 2BIT-TEST
    DUP 0 = IF BIT0OFF ELSE
    DUP 1 = IF BIT0ON ELSE
    DUP 2 = IF BIT1OFF ELSE
              BIT1ON
    THENTHENTHEN
    DROP ;
```

In the word '2BIT-TEST' the TOS is checked to see if it contains one of three possible values. If either of these three values is on the TOS then the desired word definition will be executed. If none of these three conditions has been met, then a fourth word **BIT1ON** will be executed.

Re-writing the '2BIT-TEST' word using the **CASE .. ENDCASE** structure results in qFORTH code which is better readable and thus easier to understand:

```
: 2BIT-CASE
    CASE
    0 OF BIT0OFF ENDOF
    1 OF BIT0ON ENDOF
    2 OF BIT1OFF ENDOF
        BIT1ON
    ENDCASE ;
```

The **CASE** construct is ideally used in programming an input filter from a data stream,

such as an input port which, depending upon specific data values, it is possible to invoke predefined qFORTH words related to the read input values.

Example:

```
15 CONSTANT TILT
: PIN-BALL ( BALL-CODE — )
    CASE
    0 OF FREE-BALL ENDOF
HIGH-SCORE @ OF REPLAY ENDOF
    TILT OF GAME-OVER ENDOF
( ELSE ) UPDATE-SCORE
    ENDCASE ;
```

Note: Unlike Pascal the case selectors are not limited to constants (e.g. high-score @)

2.12.2. Loops, branches and labels

2.12.2.1. Definite loops

The **DO .. LOOP** control structure is an example of a definite loop. The number of times that the loop will be executed by the MARC4 must be specified by the qFORTH programmer.

Example:

```
: DO-Example
    12 5 ( — Ch 5 )
    DO ( Ch 5 — )
        I 1 OUT ( Copy loop-index I onto TOS )
    LOOP ( Write "5 6 7 8 9 Ah Bh" to port1 )
;
```

Here the *loop index I* is starting at the value 5 and is incremented until the value 12 is reached. This is an example where we have defined a definite looping range (from 5 to 11) for the statements between the **DO** and the **LOOP** to be repeated.

On each iteration of a **DO** loop, the **LOOP** operator will increment the loop-index. It then will compare the index to the loop's limit to determine whether the loop should terminate or continue.

In addition to the FORTH-83 looping construct, the MARC4 has special hardware support for the qFORTH **#DO .. #LOOP**.

As a result of this, the **#DO..#LOOP** is the most code and speed efficient definite loop and is recommended for most loops.

Example:

```
5 #DO HELLO-WORLD #LOOP
```

In this example the loop control variable is set to 5, then decremented at the end of each iteration until 0. Hence **5 #DO .. #LOOP** will loop 5 times.

#LOOPS may also be nested (to any depth). The outer loop control variable is called J, when used inside the inner loop.

Example:

```
: NESTED-LOOPS
  7 #DO          \ OUTER LOOP
    5 #DO        \ INNER LOOP
      I J +
      Port0 OUT
    #LOOP
  #LOOP
;
```

Care should be taken using loops to compute multi-nibble arithmetic (e.g. 16-bit shift right). This is because the standard FORTH-83 definite loops change the carry flag after each iteration of the loop. In such cases the **#DO .. #LOOP** is recommended since the carry flag is not affected.

?DO .. ?LEAVE LOOP,	(limit start —) (—) (—)	IF start = limit THEN skip the loop exit loop if the branch flag is true increment loop-index by 1
DO .. -?LEAVE LOOP	(limit start —) (—) (—)	Init iterative DO..LOOP if branch flag is false, then exit loop increment loop-index by 1
?DO .. LOOP	(limit start —) (—)	IF start = limit THEN skip the loop
DO .. +LOOP	(limit start —) (n —)	Iterative loop with steps by <n> increment loop-index by n
#DO .. #LOOP	(n —) (—)	Execute #LOOP block n-times decrement loop-index until n = 0

2.12.2.2. Indefinite loops

BEGIN indicates the start of an indefinite loop control structure. The sequence of words which are to be performed by the MARC4 processor

will be repeated until a conditional repeat construct (such as **UNTIL** or **WHILE .. REPEAT**) is found. Write a counter value from 3 to 9 to port1, then finish the loop.

Example:

```
: UNTIL-Example
  3 BEGIN
    DUP Port1 OUT ( Write the current value to port1 )
    1+           ( Increment the TOS value 3 .. 9 )
    DUP 9 >     ( DUPLICATE the current value .. )
    UNTIL      ( the comparison will DROP it )
    DROP      ( skip counter value from stack )
;
```


The encapsulated **BEGIN .. UNTIL** loop block will be executed until the branch flag is set (TRUE). The branch flag will be set upon meeting the desired condition (TOS > 9).

The second conditional loops control structure

Example:

```

: BinBCD          \ Converts binary to 2 digit BCD
                  ( d [<99] — Dhi Dlo )
                  \ 1's comp of '0'
  Fh <ROT
  BEGIN
    OVER          0<>
  WHIL           \ High order is zero
    10 M-
    ROT 1- <ROT
  REPEAT         \ Count 10th
  DUP 10 >=
  IF
    10 - ROT 1- <ROT
  THEN
  NIP SWAP NOT SWAP
;
  
```

qFORTH also provides an *infinite loop* sequence, the **BEGIN .. AGAIN** which can only be escaped by **EXIT**, **?!LEAVE** or **?LEAVE**.

qFORTH – indefinite loops	
BEGIN <Condition> WHILE ... REPEAT	Condition tested at start of loop
BEGIN ... <Condition> UNTIL	Condition tested at end of loop
BEGIN ... AGAIN	Unconditional loop

BEGIN .. WHILE .. REPEAT will repeat a sequence of qFORTH words as long as a condition (computed between **BEGIN** and **WHILE**) is being met.

2.12.3. Branches and labels

While not recommended in normal programming, branches and labels have been included in qFORTH for completeness.

Labels have the following format:

```
<Label>: <instruction> | <Word>
```

Note: There is no space allowed between the label and the colon.

Example:

My_Lab1:

Only conditional branches are allowed in qFORTH, i.e. the branch will be taken, if the branch flag is set.

If unconditional branches are required, then care must be taken to set the branch flag before branching.

Example:

```
SET_BCF  BRA My_Lab1
```

Note: The scope of labels is only within a colon definition. It is not possible to branch outside a colon definition.

Example:

```
VARIABLE SINS
VARIABLE TEMPERATURE
: WAS-BAD?
  SINS @ 3 >=
;
: NEXT-LIFE
  WAS-BAD? BRA HELL
HEAVEN: TRA-LA-LA NOP
        SET_BCF BRA HEAVEN
HELL:   TEMPERATURE 1+! WORK
        SET_BCF BRA HELL
;
```

The 'NEXT-LIFE' word can also be written with high-level constructs as:

```
: NEXT-LIFE
  WAS-BAD? TOG_BF
  IF
    BEGIN
      TRA-LA-LA NOP      \ HEAVEN
    AGAIN
  ELSE
    BEGIN
      Temperature 1+! WORK \ HELL
    AGAIN
  THEN
;;
```

2.12.4. Arrays and Look-up tables

2.12.4.1. Array indexing

INDEX is a predefined qFORTH word used to access array locations. The compiler translates **INDEX** into a run-time code definition specific for the type of array being used (2ARRAY, LARRAY, etc.)

2.12.4.2. Initializing and erasing an array

Using the qFORTH word **ERASE** it is possible to erase an arrays' contents to be filled with zeros.

2.12.4.3. Array filling

A third way to initialize an array is with the word **FILL**. **FILL** requires that the beginning address of the array and the size of the array be placed onto the stack, followed by the value to be filled.

```
: FillArray      ( count n addr — )
  Y! DUP [Y]!    ( count n addr — count n )
  SWAP 1-        ( count n — n count-1 )
  #DO DUP [+Y]!
  #LOOP
  DROP
;
```

2.12.4.4. Looping in an array

The qFORTH words contained between the **DO** and **LOOP** words will be repeated from between the start element and the limit element, the element first deposited onto the stack will be decremented following the store instruction.

2.12.4.5. Moving arrays

The words **MOVE** and **MOVE>** will copy a specified number of digits from one address to another within the RAM. The difference between the two instructions is that the **MOVE** copies the specified number of digits starting from the lowest address, while **MOVE>** starts from the highest address.

```
: C-MOVE      ( n Source Dest — )
  Y! X!
  [X]@ [Y]!
  BEGIN
    1- TOG_BF
  WHILE
    [+X]@  [+Y]!
  REPEAT
  DROP
;
```

2.12.4.6. Comparing arrays

The word '**?Arrays=**' compares two array fields, starting at the last field element in descending addresses, the maximum length permitted is 16 elements. The result if the arrays are equal or not is stored in the branch flag.

```
: ?Arrays= ( n Array1[n] Array2[n] —
            [BF=1, if equal])
  X! Y! 0 SWAP
  #DO
    [X-]@  [Y-]@ - OR
  #LOOP
  0=
;
```

Another way of implementing the array comparison function is to use the **BEGIN .. UNTIL** loop as shown below.

```
: ?Arrays= ( n Array1[n] Array2[n] —
            [BF=1, if equal])
  X! Y!
  BEGIN      ( n is decremented in loop )
    [Y-]@[X-]@
    <> ?LEAVE
  1-
  UNTIL
  DROP TOG_BF
;
```

Array examples are included in the **qFORTH Language Reference Dictionary**.

2.12.5. Look-up tables

Look-up tables are implemented in most microprocessors to hold data which can be easily

accessed by means of an offset. qFORTH supports tables with the instructions: **ROMCONST**, **ROMByte@**, **DTABLE@** and **TABLE ;;**.

These instructions are described in the qFORTH Language Reference Dictionary. The basic principle of MARC4 tables is that the data to be referenced is placed into contiguous ROM memory during compile time when defined as a **ROMCONST**. The **ROMByte@** word fetches an 8-bit constant from ROM defined by the 12-bit ROM address which is on the top of the expression stack. The **DTABLE@** word permits the user to access a particular 8-bit constant from the array via the arrays address value and the 4-bit offset.

In the program file '**INCDATE.INC**', found on the applications disk, the days of the month are placed into a look-up table called '**DaysOfMonth**', the month is used to access the table in order to return the number of days in the month.

2.12.6. TICK and EXECUTE

The word '**'** (pronounced *TICK*, represented in FORTH by the apostrophe symbol) locates a word definition in memory and returns its ROM address.

EXECUTE takes the ROM address (located on the expression stack) of a colon definition and executes the word. *TICK* is useful for performing vectored execution where a word definition is executed indirectly, this can be performed by placing the address of a definition into a variable, the contents of the variable is then **EXECUTEd** as desired, this gives the user increased flexibility as he can now perform complicated pointer manipulations.

Example:

```

CODE BCD_+1!           \ <Y> = ^Digit ——<Y-1>
  [Y]@ 1 + DAA [Y-]!   \ Incr. BCD digit in RAM
END-CODE
;
: Inc_Hrs
  Time [Hrs_1] Y! BCD_+1!
  IF
    Time [Hrs_10] 1+!   \ x9:59 -> x+1 0:00
                        \ 0 -> 1 or 1 -> 2
  THEN
  Time [Hrs_10] 2@ 2 4 D= \ 24:00:00      ?
  IF
    0 0 Time [Hrs_10] 2! \ 23:59 -> 00.00
                        \ It's midnight
  THEN
;
: Inc_Hour
  LAP_Timer [Hours] 1+! \ Inc Hours binary by 1
;
; \ Wrap around at 16:00.00
: Inc_Min
  BCD_+1!               \ <Y> = ^Digit[Min_1]
  IF
    [Y]@ 1+ 6 CMP_EQ[Y]! \ 18:29 -> 18:30
                        \ On overflow ..
                        \ 18:59 -> 19:00
    IF
      0 [Y-]!           \ Reset Min_10
      Hours_Inc 3@ EXECUTE \ Computed Hrs_Inc '
      [ E O R 0 ]
    THEN
  THEN
;
: Inc_Secs
  BCD_+1!               \ <Y> = ^Digit[Sec_1]
  IF
    [Y]@ 1+ 6 CMP_EQ[Y]! \ Increment seconds
                        \ 8:25:19 -> 8:25:20
    IF
      0 [Y-]!           \ 8:30:59 -> 8:31:00
                        \ Reset Sec_10
      Inc_Min           \ Incr. Minutes
    THEN
  THEN
;
: Inc_1/100s
  BCD_+1!               \ Increment 10_ms
  IF
    BCD_+1!             \ 25.19.94 -> 25.19.95
                        \ Incr. 100_ms
    IF
      Inc_Secs          \ 30.49.99 -> 30.50.00
                        \ Incr. seconds ..
    THEN
  THEN
;
: IncTime
  ' Inc_Hrs Hours_Inc [2] 3! \ Incr. T.O.D.
  Time [Sec_1] Y! Inc_Secs \ Note use of Tick
                        \ Increment seconds
;
: Inc_10ms
  ' Inc_Hour Hours_Inc [2] 3! \ Incr. LAP timer
  LAP_Timer [10_ms] Y! \ Note use of TICK
  Inc_1/100s \ Increment 1/100 sec
;

```

\ Excerpts of program 'TEST_05' which includes TICKTIME

9	CONSTANT Seed	\ Random display update
6	ARRAY Time	\ Current Time Of Day
7	ARRAY LAP_Timer	\ Stop Watch time
3	ARRAY Hours_Inc	\ Dest. of computed GOTO
2	ARRAY C_INT6	\ INT6 counter
	VARIABLE RandomUpdate	
	VARIABLE LAP_Mode	\ LAP_Timer or T.O.D. display
	VARIABLE TimeCount	\ Count RTC interrupts

\$INCLUDE LCD_3to1

\$INCLUDE TickTime

```

: StopWatch
  C_INT6 [1] D-1!
  IF
    26 C_INT6 2!
    Inc_10ms RandomUpdate 1-!
    IF
      Seed RandomUpdate !
      LAP_Timer [1] Show6Digits
    THEN
  THEN
;

: INT5
  1 TimeCount TOGGLE
  IF DI IncTime EI THEN
;
\ Real-Time Clock Interrupt every 1/2s
\ Be on the save side

: INT6
  LAP_Mode @ 0=
  IF StopWatch THEN
;
\ Stop Watch Interrupt every 244.1 usec

: $RESET
  >SP S0 >RP FCh
  Vars_Init ( etc. );
\ Init stack pointers first
\ Setup arrays and prescaler

```

2.13. Making the best use of compiler directives

Compiler directives allow the programmer to have direct manual control over the generation and placement of program code and RAM variables.

The qFORTH compiler will automatically generate efficient code, so it is not necessary or recommended to hand optimise the application program at the beginning of the project. However, as the first version of the application is completed, the following compiler directives can be used to "fine tune" the program.

A complete list of all compiler directives may be found in the documentation shipped with the qFORTH2 compiler release diskette.

2.13.1. Controlling ROM placement

By forcing a zero page placement of commonly used words, a single byte short call will be used to access the word, hence saving a byte per call.

Examples:

```
: Called-a-lot SWAP DUP [ Z ] ;
    \ Place anywhere in zero page

: Once-in-a-blue-moon Init-RAM [ N ] ;
    \ Don't place in zero Page

: Very-Small-Word 3>R DUP 3R@ ; AT 23h
    \ Place in 5 byte hole between
    \ zero page words
```

2.13.2. Macro definitions, EXIT and ;;

If fast execution is required then critical words may be invoked as macros and expanded 'in-line'. In general, macros are identical in syntax to word definitions, except the colon and semicolon are replaced with **CODE .. END-CODE**.

Clearly 'CODE' definitions have no implied EXIT (or subroutine return) on termination. Occasionally, a colon definition does not require an EXIT on termination, in this case the ';;' statement is used instead of the ';'.

Examples:

```
07 2CONSTANT Duff-Value

nCODE Must-be-fast X! [X]@ 1+ [X-]!
    END-CODE

: Correlate-Temperature
  Read-Temperature      ( — Th Tl )
  2DUP Duff-Value D=   IF \ Make a quick exit if
    2DROP              \ duff data read in
    EXIT               \ Note use of EXIT

  THEN
  Do-Correlation        ( Th Tl — )
;
: HALT BEGIN AGAIN ;; \ Since this loop
                      \ never terminates,
                      \ then we can save
                      \ the EXIT
```

2.13.3. Controlling stack side effects

The qFORTH compiler attempts to calculate the stack effects of each word. Sometimes this is not possible, hence the two directives [E <number> R <number>] allow the programmer to manually set stack effects of the expression and the return stack.

Examples:

```

: I-know-what-I'm-doing  BEGIN DUP 1-
                        UNTIL [ E 0 ] ;
: Get_Numbers           \ Depending on the
                        value of Flag
Flag @ 5 =             \ the IF .. ELSE .. THEN
                        block will have
IF                     \ a stack effect of +4 or
                        +3.
1 2 3 4
ELSE
1 2 3 [ E 4 ]
THEN
;

```

2.13.4. \$INCLUDE directive

It is good programming practice to split a large program into a number of smaller modules, one file per module. qFORTH allows the programmer to do this with the \$INCLUDE

<filename[.INC]> directive. This directs the compiler to temporarily take the input source from another file.

Include files may be nested up to a maximum level of four.

Example:

```

$INCLUDE Lcd-Words      \ include the LCD "tool
                        box"
: Update_LCD
Colon-State @ Blink-Colon?
;

```

2.13.5. Conditional compilation

Conditional compilation enables the programmer to control which parts of the program are to be compiled. For example a typical program under development has extra code to aid debugging. This code will be removed on the final version. Using conditional compilation the programmer can keep the all the debugging information in the source, but generate code only for the application simply by commenting out the \$DEFINE DEBUG directive.

Examples:

```

$DEFINE Debug      \ IF this directive is
                   \   commented out
                   \ THEN no debugging
                   \   code is generated

: INT2

$IFDEF Debug
  CPU-Status Port6  OUT
$ENDIF
  Process-Int2
;

$DEFINE Emulatio   \ Use EVA prescaler at
                   \   address Eh
                   \ to emulate the e3405.

$IFDEF Emulation
  Eh CONSTANT Prescaler_2
  Ch CONSTANT 4_KHz
$ELSE
  Fh CONSTANT Prescaler_2
  Dh CONSTANT 4_KHz
$ENDIF

$IFDEF Emulation
  : INT4 process ;
$ELSE
  : INT6 process ;
$ENDIF

```

2.13.6. Controlling XY register optimisations

The X/Y optimize qualifiers of the qFORTH compiler help to control the depth of desired optimization steps.

- **XYLOAD**

the sequence **LIT_p .. LIT_q X!** will be optimized to: **>X \$pq**

- **XY@!**

the sequence **>X \$pq [X]!** will be optimized to **[>X]! \$pq**

- **XYTRACE**

reloading the X or Y register (i.e: sequences of **>X \$pq** will be replaced by **[+X]@** or **[Y-]!** operations whenever possible.

The qFORTH compiler keeps track of which variable is cached in the X and Y registers inside a colon definition.

Example:

The variables 'OnTime' and 'SwitchNr' are stored in consecutive RAM locations.

qFORTH source	Intermediate code	XYLOAD, XY@! optimized	Final code after XYTRACE
On Time @	LIT_3 LIT_4	[>X]@ \$OnTime	[>X]@ \$OnTime
SwitchNr +!	X! [X]@	[>Y]@ \$SwitchNr	[+X]@
	LIT_3 LIT_5	ADD	ADD
	Y! [Y]@	[Y]!	[X]!
	ADD		
	[Y]!		
	10 Bytes	6 Bytes	5 Bytes

2.14. Recommended naming conventions

<i>Form</i>	<i>Example</i>	<i>Meaning</i>
Arithmetic		
1name	1+	integer 1 (4-bit)
2name	2DUP	integer 2 (8-bit)
+name	+DRAW	takes relative input parameters
*name	*DRAW	takes scaled input parameters
Data structures		
names	EMPLOYEES	table or array
#name	#EMPLOYEES	total number of elements
name#	EMPLOYEE#	current item number (variable)
(n) name	EMPLOYEE [13]	sets current item
+name	+EMPLOYEE	advance to next element
name+	DATE+	size of offset to item from beginning of structure
/name	/SIDE	size of (elements "per")
>name	>IN	index pointer
Direction, conversion		
name<	SLIDE<	backwards
name>	MOVE>	forwards
<name	<PORT4	from
>name	>PORT0	to
name>name	FEET>METERS	convert to
\name	\LINE	downward
/name	/LINE	upward
Logic, control		
name?	SHORT?	return boolean value
-name?	-SHORT?	returns reversed boolean
?name	?DUP (maybe DUP)	operates conditionally
+name	+CLOCK	enable
name	BLINKING	or, absence of symbol
-name	-CLOCK	disable
	-BLINKING	
Memory		
@name	@CURSOR	save value of
!name	!CURSOR	restore value of
name!	SECONDS!	store into
name@	INDEX@	fetch from
' name	' INC-MINUTE	address of name
Numeric types		
Dname	D+	2 cell size, 2's complement integer encoding
Mname	M*	mixed 4 and 8-bit operator
Tname	T*	3 cell size
Qname	Q*	4 cell size

These naming conventions are based on a proposal given by Leo Brodie in his book 'Thinking FORTH'.

2.14.1. How to pronounce the symbols

!	store	[]	square brackets
@	fetch	“	quote
#	sharp or “number”	,	as prefix: Tick; as suffix: prime
\$	dollar	~	tilde
%	percent		bar
^	caret	\	backslash
&	ampersand	/	slash
*	star	<	less-than; left dart
()	left paren and right paren ; paren	>	greater-than; right dart
-	dash; not	?	question or “query”
+	plus	,	comma
=	equals	.	dot
{ }	faces or “curly brackets”		

2.15. Book list

2.1.5.1 Recommended books

“**Starting Forth**” is highly recommended as a good general introduction to FORTH especially chapters 1.6.

“**Starting FORTH**” is also now available in German, French, Dutch, Japanese and Chinese.

“**Thinking FORTH**” is the follow-on book to “Starting FORTH” and discusses more advanced topics, such as system level programming.

“**Complete FORTH**” has been acknowledged as the definitive FORTH text book.

Title: “**Starting FORTH**” (2nd edition)

Author: Leo Brodie

Publisher: Prentice Hall, 1987

ISBN: 0-13-843079-9

Title: “**Programmieren in FORTH**” (German Version)

Author: Leo Brodie

Publisher: Hanser, 1984

ISBN: 3-446-14070-0

Title: “**Thinking FORTH**”

Author: Leo Brodie

Publisher: Prentice Hall, 1984

ISBN: 0-13-917568-7

Title: “**Complete FORTH**”

Author: Winfield

Publisher: Sigma Technical Press, 1983

2.1.5.2 General interest

The following list of books and papers show the spectrum of **FORTH** literature. These books are of background interest **ONLY** and may contain information not 100% relevant to programming in **qFORTH** on the MARC4.

Title: **“Mastering FORTH”**

Author: Leo Brodie

Publisher: Brady Publishing, 1989

ISBN: 0-13-559957-1

Title: **“Dr. Dobbs Tool-Box of FORTH Vol. II”**,

Publisher: M&T Books, 1987

ISBN: 0-934375-41-0

Title: **“FORTH”** (Byte Magazine)

Author: L. Topin

Publisher: McGraw Hill, 1985

Title: **“The use of FORTH in process control”**

Proc. of the International '77 Mini-Micro Computer Conference, Geneva;

Authors: Moore & Rather, Publisher: IPC and Technology Press, England, 1977

Title: **“FORTH: A cost saving approach to Software Development”**

Author: Hicks

Publisher: Wescon/Los Angeles, 1978

Title: **“FORTH's Forte is Tighter Programming”**

Author: Hicks

Publisher: Electronics (Magazine), March 1979

Title: **“FORTH a text and reference”**

Authors: Kelly & Spier

Publisher: Prentice Hall, 1986

ISBN: 0-13-326331-2

Title: **“FORTH: the next step”**

Author: Ron Geere

Publisher: ..., 1986

ISBN: 0-261-18050

I. Hardware Description

II. Programming in qFORTH

III. Instruction Set

IV. qFORTH Language Dictionary

Addresses

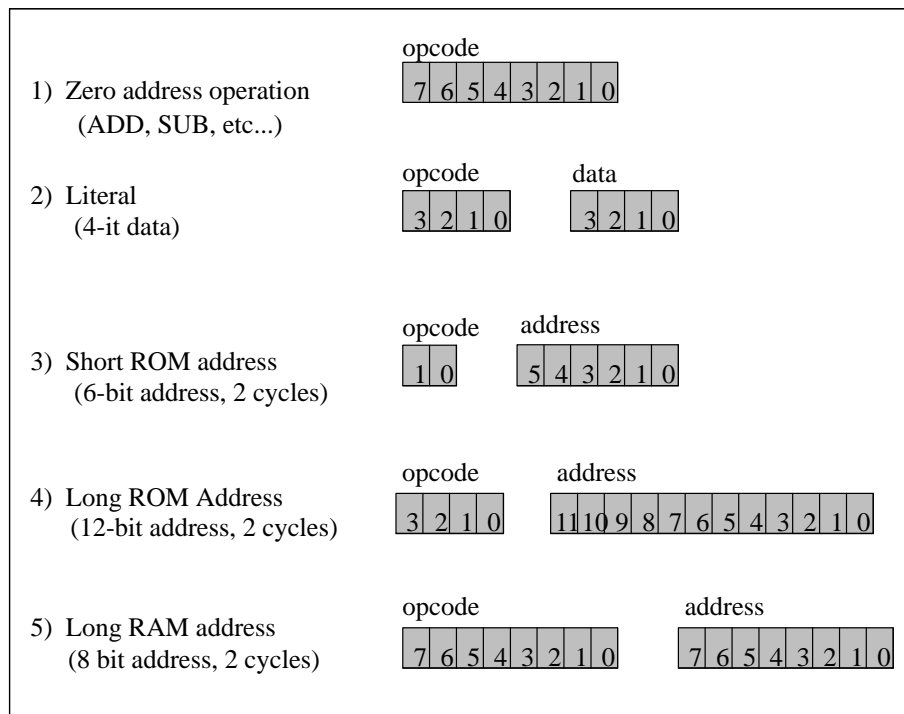
3. MARC4 instruction set

3.1. Introduction

Most of the MARC4 instructions are single byte instructions. The MARC4 is a zero address machine where the instruction to be performed contains only the operation

and not the source or destination addresses of the data. Altogether there are five types of instruction formats for the MARC4 processor.

Table 3.1: MARC4 opcode formats



A Literal is a 4-bit constant value which is placed on the data stack. In the MARC4 native code they are represented as **LIT_<value>**, where <value> is the hexadecimal representation from 0 to 15 (0..F). This range is a result of the MARC4's 4-bit data width.

The long RAM address format is used by the four 8-bit RAM address registers which can be pre-incremented, post-decremented or loaded directly from the MARC4's internal bus. This results in a direct accessible RAM address space of up to 256×4 -bit.

The 6-bit short address and the 12-bit long address formats are both used to address the byte-wide ROM via call and conditional branch instructions. This results in a ROM address space of up to $4k \times 8$ -bit words.

The MARC4 instruction set includes both short and long

call instructions as well as conditional branch instructions. The short instructions are single byte instructions, with the jump address included in the instruction. On execution, the lower 6-bits from the instruction word are directly loaded into the PC.

Short call (**SCALL**) and short branch (**SBRA**) instructions are handled in different ways. **SCALL** jumps to one of 64 evenly distributed addresses within the zero page (from 000 to 1FF hex). The short branch instruction allows a jump to one of 64 addresses contained within the current page. Long jump instructions can jump anywhere within the ROM area. The **CALL** and **SCALL** instructions write the incremented program counter contents to the return stack. This address is loaded back to the PC when the associated **EXIT** or **RTI** instruction is encountered.

MARC4 Programmer's Guide

Instruction Set

TEMIC

TELEFUNKEN Semiconductors

MARC4 instruction set [V 2.0]							
00	ADD	10	SHL	20	TABLE	30	[X]@
01	ADDC	11	ROL	21	—	31	[+X]@
02	SUB	12	SHR	22	>R	32	[X-]@
03	SUBB	13	ROR	23	I / R@	33	[>X]@ \$xx
04	XOR	14	INC	24	—	34	[Y]@
05	AND	15	DEC	25	EXIT	35	[+Y]@
06	CMP_EQ	16	DAA	26	SWAP	36	[Y-]@
07	CMP_NE	17	NOT	27	OVER	37	[>Y]@ \$xx
08	CMP_LT	18	TOG_BF	28	2>R	38	[X]!
09	CMP_LE	19	SET_BCF	29	3>R	39	[+X]!
0A	CMP_GT	1A	DI	2A	2R@	3A	[X-]!
0B	CMP_GE	1B	IN	2B	3R@	3B	[>X]! \$xx
0C	OR	1C	DECR	2C	ROT	3C	[Y]!
0D	CCR@	1D	RTI	2D	DUP	3D	[+Y]!
0E	CCR!	1E	SWI	2E	DROP	3E	[Y-]!
0F	SLEEP	1F	OUT	2F	DROPR	3F	[>Y]! \$xx
40	CALL \$0xx	50	BRA \$0xx	60	LIT_0	70	SP@
41	CALL \$1xx	51	BRA \$1xx	61	LIT_1	71	RP@
42	CALL \$2xx	52	BRA \$2xx	62	LIT_2	72	X@
43	CALL \$3xx	53	BRA \$3xx	63	LIT_3	73	Y@
44	CALL \$4xx	54	BRA \$4xx	64	LIT_4	74	SP!
45	CALL \$5xx	55	BRA \$5xx	65	LIT_5	75	RP!
46	CALL \$6xx	56	BRA \$6xx	66	LIT_6	76	X!
47	CALL \$7xx	57	BRA \$7xx	67	LIT_7	77	Y!
48	CALL \$8xx	58	BRA \$8xx	68	LIT_8	78	>>SP \$xx
49	CALL \$9xx	59	BRA \$9xx	69	LIT_9	79	>>RP \$xx
4A	CALL \$Axx	5A	BRA \$Axx	6A	LIT_A	7A	>>X \$xx
4B	CALL \$Bxx	5B	BRA \$Bxx	6B	LIT_B	7B	>>Y \$xx
4C	CALL \$Cxx	5C	BRA \$Cxx	6C	LIT_C	7C	NOP
4D	CALL \$Dxx	5D	BRA \$Dxx	6D	LIT_D	7D	—
4E	CALL \$Exx	5E	BRA \$Exx	6E	LIT_E	7E	—
4F	CALL \$Fxx	5F	BRA \$Fxx	6F	LIT_F	7F	—
80..BF	SBRA \$xxx	Short branch inside current page					
C0..FF	SCALL \$xxx	Short subroutine CALL into 'zero page'					

Code [hex]	Mnemonic	Operation	Symbolic description [Stack effects]	Instr. cycles	Flags C % B I
00	ADD	Add the top 2 stack digits	$E (n1\ n2 \leftarrow n1+n2)$ If overflow then $B:=C:=1$ else $B:=C:=0$	1	xxx-
01	ADDC	Add with carry the top 2 stack digits	$E (n1\ n2 \leftarrow n1+n2+C)$ If overflow then $B:=C:=1$ else $B:=C:=0$	1	xxx-
02	SUB	2's complement subtraction of the top 2 digits	$E (n1\ n2 \leftarrow n1+n2+1)$ If overflow then $B:=C:=1$ else $B:=C:=0$	1	xxx-
03	SUBB	1's complement subtraction of the top 2 digits	$E (n1\ n2 \leftarrow n1+n2/C)$ If overflow then $B:=C:=1$ else $B:=C:=0$	1	xxx-
04	XOR	Exclusive-OR top 2 stack digits	$E (n1\ n2 \leftarrow n1\ XOR\ n2)$ If result=0 then $B:=1$ else $B:=0$	1	-xx-
05	AND	Bitwise-AND top 2 stack digits	$E (n1\ n2 \leftarrow n1\ AND\ n2)$ If result=0 then $B:=1$ else $B:=0$	1	-xx-
0C	OR	Bitwise-OR top 2 stack digits	$E (n1\ n2 \leftarrow n1\ OR\ n2)$ If result=0 then $B:=1$ else $B:=0$	1	-xx-
06	CMP_EQ	Equality test for top 2 stack digits	$E (n1\ n2 \leftarrow n1)$ If $n1=n2$ then $B:=1$ else $B:=0$	1	xxx-
07	CMP_NE	Inequality test for top 2 stack digits	$E (n1\ n2 \leftarrow n1)$ If $n1 < n2$ then $B:=1$ else $B:=0$	1	xxx-
08	CMP_LT	Less-than test for top 2 stack digits	$E (n1\ n2 \leftarrow n1)$ If $n1 < n2$ then $B:=1$ else $B:=0$	1	xxx-
09	CMP_LE	Less-or-equal for top 2 stack digits	$E (n1\ n2 \leftarrow n1)$ If $n1 \leq n2$ then $B:=1$ else $B:=0$	1	xxx-
0A	CMP_GT	Greater-than for top 2 stack digits	$E (n1\ n2 \leftarrow n1)$ If $n1 > n2$ then $B:=1$ else $B:=0$	1	xxx-
0B	CMP_GE	Greater-or-equal for top 2 stack digits	$E (n1\ n2 \leftarrow n1)$ If $n1 \geq n2$ then $B:=1$ else $B:=0$	1	xxx-
0E	CCR!	Restore condition codes	$E (n \leftarrow R \leftarrow)$	1	xxxx
0F	SLEEP	CPU in 'sleep mode', interrupts enabled	$E (\leftarrow) R (\leftarrow)$ $I:=1$	1	-x-1
10	SHL	Shift TOS left into carry	$C \leftarrow 197 > 3210 < \leftarrow 0$ $B:=C:=MSB$	1	xxx-
11	ROL	Rotate TOS left through carry	$\dots \leftarrow C \leftarrow 3210 < \leftarrow C \leftarrow \dots$ $B:=C:=MSB$	1	xxx-
12	SHR	Shift TOS right into Carry	$0 \rightarrow 3210 \rightarrow C$ $B:=C:=LSB$	1	xxx-
13	ROR	Rotate TOS right through carry	$\dots \rightarrow C \rightarrow 3210 \rightarrow C \rightarrow \dots$ $B:=C:=LSB$	1	xxx-

Code [hex]	Mnemonic	Operation	Symbolic description [stack effects]	Instr. cycles	Flags C % B I
14	INC	Increment TOS	E (n — n+1) If result=0 then B:=1 else B:=0	1	- x x -
15	DEC	Decrement TOS	E (n — n-1) If result=0 then B:=1 else B:=0	1	- x x -
16	DAA	Decimal adjust for addition (in BCD arithmetic)	If TOS>9 OR C=1 then E (n — n+6) B:=C:=1 else E (n — n) R (—) B:=C:=0	1 1	1 x 1 - 0 x 0 -
17	NOT	1's complement of TOS	E (n — /n) If result=0 then B:=1 else B:=0	1	- x x -
18	TOG_BF	Toggle branch flag	If B = 1 then B:=0 else B:=1	1	- x x -
19	SET_BCF	Set branch and carry flag	B:=C:=1	1	1 x 1 -
1A	DI	Disable all interrupts	E (—) R (—) I:=0	1	- x - 0
1B	IN	Read data from 4-bit I/O port	E (port — n) If port=0 then B:=1 else B:=0	1	- x x -
1C	DECR	Decrement index on return stack	R (uun — uun-1) If n-1=0 then B:=0 else B:=1	2 2	- 1 0 -- 0 1 -
1D	RTI	Return from interrupt routine; enable all interrupts	E (—) R (\$xxx —) PC := \$xxx I:=1	2	- - - 1
1E	SWI	Software interrupt	E (n1 n2 —) R (—) [n1,n2 = 0,1,2,4,8]	1	- x - -
1F	OUT	Write data to 4-bit I/O port	E (n port —) R (—)	1	- x - -
20 21	TABLE	Fetch an 8-bit ROM constant,, perform an EXIT	E (— d [\$xxx2]) R (\$xxx1 \$xxx2 —) PC:=\$xxx1	3	- - - -
22	>>R	Move (loop) index onto return stack	E (n —) R (— uun)	1	- - - -
23	I R@	Copy (loop) index from the return stack onto TOS	E (— n) R (uun — uun)	1	- - - -
24 25	EXIT	Return from subroutine (' ; ')	E (—) R (\$xxx —) PC:=\$xxx	2	- - - -
26	SWAP	Exchange the top 2 digits	E (n1 n2 — n2 n1) R (—)	1	- - - -
27	OVER	Push a copy of TOS-1 onto TOS	E (n1 n2 — n1 n2 n1) R (—)	1	- - - -
28	2>R	Move top 2 digits onto return stack	E (n1 n2 —) R (— un1n2)	3	- - - -
29	3>R	Move top 3 digits onto return stack	E (n1 n2 n3 —) R (— n1n2n3)	4	- - - -
2A	2R@	Copy 2 digits from return to expression stack	E (— n1 n2) R (un1n2 — un1n2)	2	- - - -
2B	3R@	Copy 3 digits from return to expression stack	E (— n1 n2 n3) R (n1n2n3 — n1n2n3)	4	- - - -

Code [hex]	Mnemonic	Operation	Symbolic description [stack effects]	Instr. cycles	Flags C % B I
2C	ROT	Move third digit onto TOS	E (n1 n2 n3 — n2 n3 n1) R (—)	3	----
2D	DUP	Duplicate the TOS digit	E (n — n n) R (—)	1	----
2E	DROP	Remove TOS digit from the expression stack	E (n —) R (—) SP:=SP-1	1	----
2F	DROPR	Remove one entry from the return stack	E (—) R (uuu —) RP:=RP-4	1	----
30	[X]@	Indirect fetch from RAM addressed by the X register	E (— n) R (—) X:=X Y:=Y	1	—
31	[+X]@	Indirect fetch from RAM addressed by preincremented X register	E (— n) R (—) X:=X+1 Y:=Y	1	—
32	[X-]@	Indirect fetch from RAM addressed by the postdecremented X reg.	E (— n) R (—) X:=X-1 Y:=Y	1	—
33 xx	[>X]@ \$xx	Direct fetch from RAM addressed by the X register	E (— n) R (—) X:=\$xx Y:=Y	2	—
34	[Y]@	Indirect fetch from RAM addressed by the Y register	E (— n) R (—) X:=X Y:=Y	1	—
35	[+Y]@	Indirect fetch from RAM addressed by preincremented Y register	E (— n) R (—) X:=X Y:=Y+1	1	—
36	[Y-]@	Indirect fetch from RAM addressed by postdecremented Y reg.	E (— n) R (—) X:=X Y:=Y-1	1	—
37 xx	[>Y]@ \$xx	Direct fetch from RAM addressed by the Y register	E (— n) R (—) X:=X Y:=\$xx	2	—
38	[X]!	Indirect store into RAM addressed by the X register	E (n —) R (—) X:=X Y:=Y	1	----
39	[+X]!	Indirect store into RAM addressed by pre-incremented X register	E (n —) R (—) X:=X+1 Y:=Y	1	----
3A	[X-]!	Indirect store into RAM addressed by the postdecremented X reg.	E (n —) R (—) X:=X-1 Y:=Y	1	----
3B xx	[>>X]! \$xx	Direct store into RAM addressed by the X register	E (n —) R (—) X:=\$xx Y:=Y	2	----
3C	[Y]!	Indirect store into RAM addressed by the Y register	E (n —) R (—) X:=X Y:=Y	1	----
3D	[+Y]!	Indirect store into RAM addressed by pre-incremented Y register	E (n —) R (—) X:=X Y:=Y+1	1	----
3E	[Y-]!	Indirect store into RAM addressed by the postdecremented Y reg.	E (n —) R (—) X:=X Y:=Y-1	1	----

MARC4 Programmer's Guide

Instruction Set

TEMIC

TELEFUNKEN Semiconductors

Code [hex]	Mnemonic	Operation	Symbolic description [stack effects]	Instr. cycles	Flags C % B I
3F xx	[>>Y]! \$xx	Direct store into RAM addressed by the Y register	E (n —) R (—) X:=X Y:=\$xx	2	----
70	SP@	Fetch the current expression stack pointer	E (— SPh SPI+1) R (—) SP:=SP+2	2	—
71	RP@	Fetch current return stack pointer	E (— RPh RPI) R (—)	2	—
72	X@	Fetch current X register contents	E (— Xh XI) R (—)	2	—
73	Y@	Fetch current Y register contents	E (— Yh YI) R (—)	2	—
74	SP!	Move address into the expression stack pointer	E (dh dl — ?) R (—) SP:=dh_dl	2	----
75	RP!	Move address into the return stack pointer	E (dh dl —) R (— ?) RP:=dh_dl	2	----
76	X!	Move address into the X register	E (dh dl —) R (—) X:=dh_dl	2	----
77	Y!	Move address into the Y register	E (dh dl —) R (—) Y:=dh_dl	2	----
78 xx	>>SP \$xx	Set expression stack pointer	E (—) R (—) SP:=\$xx	2	----
79 xx	>>RP \$xx	Set return stack pointer direct	E (—) R (—) RP:=\$xx	2	----
7A xx	>>X \$xx	Set RAM address register X direct	E (—) R (—) X:=\$xx	2	----
7B xx	>>Y \$xx	Set RAM address register Y direct	E (—) R (—) Y:=\$xx	2	----
7C	NOP	No operation	PC:=PC+1	1	----
7D..7F	NOP	Illegal instruction	PC:=PC+1	1	----
4x xx	CALL \$xxx	Unconditional long CALL	E (—) R (— PC+2) PC:=\$xxx	3	----
5x xx	BRA \$xxx	Conditional long branch	If B=1 then PC:=\$xxx else PC:=PC+1	2 2	--1-- -0-
6n	LIT_n	Push literal/constant n onto TOS	E (— n) R (—)	1	----
80..BF	SBRA \$xxx	Conditional short branch in page	If B=1 then PC:= \$xxx else PC:=PC+1	2 2	----- ---
C0..FF	SCALL \$xxx	Unconditional short CALL	E (—) R (— PC+1) PC:= \$xxx	2	----

3.1.1. Description of used identifiers and abbreviations

n1 n2 n3	Three nibbles on the expression stack	RP	Return stack pointer (8-bits), the RAM address register which points to the last entry on the return address stack.
n3n2n1	Three nibbles on the return stack which combine to form a 12-bit word	X	RAM address register X (8-bits),
un2n1	Two nibbles on the return stack (i.e. DO loop index and limit), 'u' is an unused (undefined) nibble on the return stack,	Y	RAM address register Y (8-bits), these registers can be used in 3 different addressing modes (direct, pre-incremented or postdecremented addressing).
/n	1's complement of the 4-bit word n,	TOS	Top of (expression) stack (4-bits),
3210	Numbered bits within a 4-bit word,	CCR	Condition code register (4-bits), which contains:
\$xx	8-bit hexadecimal RAM address,	I [bit 0]	Interrupt-enable flag,
\$xxx	12-bit hexadecimal ROM address,	B [bit 1]	Branch flag,
PC	Program counter (12-bits),	% [bit 2]	Reserved flag, (currently unused),
SP	Expression stack pointer (8-bits), the RAM address register which points to the RAM location con-taining the second digit (TOS-1) on the expression stack	C [bit 3]	Carry flag,
		/C	NOT carry flag.

3.1.2. Stack notation

E (n1 n2 — n)	Expression stack contents (rightmost 4-bit digit is in TOS)
R (n1n2n3 —)	Return stack contents (rightmost 12-bit word is top entry)
RET (— ROMAddr)	Return address stack effects
EXP (—)	Expression / data stack effects
True condition	= branch flag set in CCR
False condition	= branch flag reset in CCR
n	4-bit data value
d	8-bit data value
addr	8-bit RAM address
ROMAddr	12-bit ROM address

3.2. The qFORTH language -quick reference guide

3.2.1. Arithmetic/logical

-	EXP (n1 n2 — n1-n2)	Subtract the top two values
+	EXP (n1 n2 — n1+n2)	Add up the two top 4-bit values
-C	EXP (n1 n2 — n1+/n+/C)	1's compl. subtract with borrow
+C	EXP (n1 n2 — n1+n2+C)	Add with carry top two values
1+	EXP (n — n+1)	Increment the top value by 1
1-	EXP (n — n-1)	Decrement the top value by 1
2*	EXP (n — n*2)	Multiply the top value by 2
2/	EXP (n — n DIV 2)	Divide the 4-bit top value by 2
D+	EXP (d1 d2 — d1+d2)	Add the top two 8-bit values
D-	EXP (d1 d2 — d1-d2)	Subtract the top two 8-bit values
D2/	EXP (d — d/2)	Divide the top 8-bit value by 2
D2*	EXP (d — d*2)	Multiply the top 8-bit value by 2
M+	EXP (d1 n — d2)	Add a 4-bit to an 8-bit value
M-	EXP (d1 n — d2)	Subtract 4-bit from an 8-bit value
AND	EXP (n1 n2 — n1^n2)	Bitwise AND of top two values
OR	EXP (n1 n2 — n1 v n2)	Bitwise OR the top two values
ROL	EXP (—)	Rotate TOS left through carry
ROR	EXP (—)	Rotate TOS right through carry
SHL	EXP (n — n*2)	Shift TOS value left into carry
SHR	EXP (n — n/2)	Shift TOS value right into carry
NEGATE	EXP (n — -n)	2's complement the TOS value
DNEGATE	EXP (d — -d)	2's complement top 8-bit value
NOT	EXP (n — /n)	1's complement of the top value
XOR	EXP (n1 n2 — n3)	Bitwise Ex-OR the top 2 values

3.2.2. Comparisons

>	EXP (n1 n2 —)	If n1>n2, then branch flag set
<	EXP (n1 n2 —)	If n1<n2, then branch flag set
>=	EXP (n1 n2 —)	If n1>=n2, then branch flag set
<=	EXP (n1 n2 —)	If n1<=n2, then branch flag set
<>	EXP (n1 n2 —)	If n1<>n2, then branch flag set
=	EXP (n1 n2 —)	If n1=n2, then branch flag set
0<>	EXP (n —)	If n <>0, then branch flag set
0=	EXP (n —)	If n = 0, then branch flag set
D>	EXP (d1 d2 —)	If d1>d2, then branch flag set
D<	EXP (d1 d2 —)	If d1<d2, then branch flag set
D>=	EXP (d1 d2 —)	If d1>=d2, then branch flag set
D<=	EXP (d1 d2 —)	If d1<=d2, then branch flag set
D=	EXP (d1 d2 —)	If d1=d2, then branch flag set
D<>	EXP (d1 d2 —)	If d1<>d2, then branch flag set
D0<>	EXP (d —)	If d <>0, then branch flag set
D0=	EXP (d —)	If d =0, then branch flag set
DMAX	EXP (d1 d2 — dMax)	8-bit maximum value of d1, d2
DMIN	EXP (d1 d2 — dMin)	8-bit minimum value of d1, d2
MAX	EXP (n1 n2 — nMax)	4-bit maximum value of n1, n2
MIN	EXP (n1 n2 — nMin)	4-bit minimum value of n1, n2

3.2.3. Control structures

AGAIN	EXP (—)	Ends an infinite loop BEGIN .. AGAIN
BEGIN	EXP (—)	BEGIN of most control structures
CASE	EXP (n — n)	Begin of CASE .. ENDCASE block
DO	EXP (limit start —) RET (— u limit start)	Initializes an iterative DO..LOOP
ELSE	EXP (—)	Executed when IF condition is false
ENDCASE	EXP (n —)	End of CASE..ENDCASE block
ENDOF	EXP (n — n)	End of <n> OF .. ENDOF block
EXECUTE	EXP (ROMAddr —)	Execute word located at ROMAddr
EXIT	RET (ROMAddr —)	Unstructured EXIT from ‘:’-definition
IF	EXP (—)	Conditional IF .. ELSE .. THEN block
LOOP	EXP (—)	Repeat LOOP, if index+1<limit
<n> OF	EXP (c n —)	Execute CASE block, if n=c
REPEAT	EXP (—)	Unconditional branch to BEGIN of BEGIN .. WHILE .. REPEAT
THEN	EXP (—)	Closes an IF statement
UNTIL	EXP (—)	Branch to BEGIN, if condition is false
WHILE	EXP (—)	Execute WHILE .. REPEAT block, if condition is true
+LOOP	EXP (n —) RET (u limit I — u limit I+n)	Repeat LOOP, if I+n < limit
#DO	EXP (n —) RET (— u u n)	Execute the #DO .. #LOOP block n-times
#LOOP	EXP (—) RET (u u I—u u I-1)	Decrement loop index by 1 downto zero
?DO	EXP (Limit Start —)	if start=limit, skip LOOP block
?LEAVE	EXP (—)	Exit any loop, if condition is true
-?LEAVE	EXP (—)	Exit any loop, if condition is false

3.2.4. Stack Operations

0 .. Fh,	EXP (— n)	Push 4-bit literal on EXP stack
0 .. 15	EXP (— n)	Places ROM address
' <name>	EXP (— ROMAddr)	of colon-definition <name> on EXP stack
<ROT	EXP (n1 n2 n — n n1 n2)	Move top value to 3rd stack pos.
>R	EXP (n —) RET (— u u n)	Move top value onto the return stack
?DUP	EXP (n — n n)	Duplicate top value, if n <>0
DEPTH	EXP (— n)	Get current expression stack depth
DROP	EXP (n —)	Remove the top 4-bit value
DUP	EXP (n — n n)	Duplicate the top 4-bit value
I	EXP (— I) RET (u u I — u u I)	Copy loop index I from return to expression stack
J	EXP (— J)	Fetch index value of outer loop [2nd return stack level entry]
NIP	EXP (n1 n2 — n2)	Drop second to top 4-bit value
OVER	EXP (n1 n2 — n1 n2 n1)	Copy 2nd over top 4-bit value
PICK	EXP (× — n[x])	Copy the x-th value from the expression stack onto TOS
RFREE	EXP (— n)	Get # of unused RET stack entries
R>	EXP (— n) RET (u u n —)	Move top 4-bits from return to expression stack
R@	EXP (— n) RET (u u n — u u n)	Copy top 4-bits from return to expression stack
ROLL	EXP (n —)	Move n-th value within stack to top
ROT	EXP (n1 n2 n — n2 n n1)	Move 3rd stack value to top pos.

SWAP	EXP (n1 n2 — n2 n1)	Exchange top two values on stack
TUCK	EXP (n1 n2 — n2 n1 n2)	Duplicate top value, move under second item
2>R	EXP (n1 n2 —)	Move top two values from expression to return stack
	RET (— u n2 n1)	
2DROP	EXP (n1 n2 —)	Drop top 2 values from the stack
2DUP	EXP (d — d d)	Duplicate top 8-bit value
2NIP	EXP (d1 d2 — d2)	Drop 2nd 8-bit value from stack
2OVER	EXP (d1 d2 — d1 d2 d1)	Copy 2nd 8-bit value over top value
2<ROT	EXP (d1 d2 d — d d1 d2)	Move top 8-bit value to 3rd pos'n
2R>	EXP (— n1 n2)	Move top 8-bits from return to expression stack
	RET (u n2 n1 —)	
2R@	EXP (— n1 n2)	Copy top 8-bits from return to expression stack
	RET (u n2 n1 — u n2 n1)	
2ROT	EXP (d1 d2 d — d2 d d1)	Move 3rd 8-bit value to top value
2SWAP	EXP (d1 d2 — d2 d1)	Exchange top two 8-bit values
2TUCK	EXP (d1 d2 — d2 d1 d2)	Tuck top 8-bits under 2nd byte
3>R	EXP (n1 n2 n3 —)	Move top 3 nibbles from the expression onto the return stack
	RET (— n3 n2 n1)	
3DROP	EXP (n1 n2 n3 —)	Remove top 3 nibbles from stack
3DUP	EXP (t — t t)	Duplicate top 12-bit value
3R>	EXP (— n1 n2 n3)	Move top 3 nibbles from return to the expression stack
	RET (n3 n2 n1 —)	
3R@	EXP (— n1 n2 n3)	Copy 3 nibbles (1 entry) from the return to the expression stack
	RET (n3 n2 n1 — n3 n2 n1)	

3.2.5. Memory operations

!	EXP (n addr —)	Store a 4-bit value in RAM
@	EXP (addr — n)	Fetch a 4-bit value from RAM
+!	EXP (n addr —)	Add 4-bit value to RAM contents
1+!	EXP (addr —)	Increment a 4-bit value in RAM
1-!	EXP (addr —)	Decrement a 4-bit value in RAM
2!	EXP (d addr —)	Store an 8-bit value in RAM
2@	EXP (addr — d)	Fetch an 8-bit value from RAM
D+!	EXP (d addr —)	Add 8-bit value to byte in RAM
D-!	EXP (d addr —)	Subtract 8-bit value from a byte in RAM
DTABLE@	EXP (ROMAddr n — d)	Indexed fetch of a ROM constant
DTOGGLE	EXP (d addr —)	Exclusive-OR 8-bit value with byte in RAM
ERASE	EXP (addr n —)	Sets n memory cells to 0
FILL	EXP (addr n n1 —)	Fill n memory cells with n1
MOVE	EXP (n from to —)	Move a n-digit array in memory
ROMByte@	EXP (ROMAddr — d)	Fetch an 8-bit ROM constant
TOGGLE	EXP (n addr —)	Ex-OR value at address with n
3!	EXP (nh nm nl addr —)	Store 12-bit value into a RAM array
3@	EXP (addr — nh nm nl)	Fetch 12-bit value from RAM
T+!	EXP (nh nm nl addr —)	Add 12-bits to 3 RAM cells
T-!	EXP (nh nm nl addr —)	Subtract 12-bits from 3 nibble RAM array
TD+!	EXP (d addr —)	Add byte to a 3 nibble RAM array
TD-!	EXP (d addr —)	Subtract byte from 3 nibble array

3.2.6. Predefined structures

(cccccc)		In-line comment definition
\ cccccc		Comment until end of the line
: <name>	RET (—)	Begin of a colon definition
;	RET (ROMAddr —)	Exit; ends any colon definition
[FIRST]	EXP (— 0)	Index (=0) for first array element
[LAST]	EXP (— n d)	Index for last array element
CODE	EXP (—)	Begins an in-line macro definition
END-CODE	EXP (—)	Ends an In-line macro definition
ARRAY	EXP (n —)	Allocates space for a 4-bit array
2ARRAY	EXP (n —)	Allocates space for an 8-bit array
CONSTANT	EXP (n —)	Defines a 4-bit constant
2CONSTANT	EXP (d —)	Defines an 8-bit constant
LARRAY	EXP (d —)	Allocates space for a long 4-bit array with up to 255 elements
2LARRAY	EXP (d —)	Allocates space for a long byte array
Index	EXP (n d addr—addr')	Run-time array access using a variable array index
ROMCONST	EXP (—)	Define ROM look-up table with 8-bit values
VARIABLE	EXP (—)	Allocates memory for 4-bit value
2VARIABLE	EXP (—)	Creates an 8-bit variable
<n> ALLOT		Allocate space for <n+1> nibbles of un-initialized RAM
AT <address>		Fixed <address> placement
: INTx	RET (— ROMAddr)	Interrupt service routine entry
\$AutoSleep		Entry point address on return stack underflow
: \$RESET	EXP (—)	Entry point on power-on reset

3.2.7. Assembler mnemonics

ADD	EXP (n1 n2 — n1+n2)	Add the top two 4-bit values
ADDC	EXP (n1 n2 — n1+n2+C)	Add with carry top two values
CCR!	EXP (n —)	Write top value into the CCR
CCR@	EXP (— n)	Fetch the CCR onto top of stack
CMP_EQ	EXP (n1 n2 — n1)	If n1=n2, then branch flag set
CMP_GE	EXP (n1 n2 — n1)	If n1>=n2, then branch flag set
CMP_GT	EXP (n1 n2 — n1)	If n1>n2, then branch flag set
CMP_LE	EXP (n1 n2 — n1)	If n1<=n2, then branch flag set
CMP_LT	EXP (n1 n2 — n1)	If n1<n2, then branch flag set
CMP_NE	EXP (n1 n2 — n1)	If n1<>n2, then branch flag set
CLR_BCF	EXP (—)	Clear branch and carry flag
SET_BCF	EXP (—)	Set branch and carry flag
TOG_BF	EXP (—)	Toggle the branch flag
DAA	EXP (n>9 or C set — n+6)	BCD arithmetic adjust [addition]
DAS	EXP (n — 10+/n+C)	9's complement for BCD subtract
DEC	EXP (n — n-1)	Decrement top value by 1
DECR	RET (u uI — u uI-1)	Decrement value on the return stack
DI	EXP (—)	Disable interrupts
DROPR	RET (u u u —)	Drop element from return stack
EXIT	RET (ROMAddr —)	Exit from current ':'-definition
EI	EXP (—)	Enable interrupts
IN	EXP (port — data)	Read data from an I/O port
INC	EXP (n — n+1)	Increment the top value by 1
NOP	EXP (—)	No operation
NOT	EXP (n — /n)	1's complement of the top value
RP!	EXP (d —)	Store as return stack pointer
RP@	EXP (— d)	Fetch current RET stack pointer

RTI	RET (RETAddr —)	Return from interrupt routine
SLEEP	EXP (—)	Enter 'sleep-mode', enable all interrupts
SWI0 SWI7	EXP (—)	Software triggered interrupt
SP!	EXP (d —)	Store as stack pointer
SP@	EXP (— d)	Fetch current stack pointer
SUB	EXP (n1 n2 — n1-n2)	2's complement subtraction
SUBB	EXP (n1 n2 — n1+/n2+C)	1's compl. subtract with borrow
TABLE	EXP (— d)	
	RET (RetAddr RomAddr —)	Fetches an 8-bit constant from an address in ROM
OUT	EXP (data port —)	Write data to I/O port
X@	EXP (— d)	Fetch current × register contents
[X]@	EXP (— n)	Indirect × fetch of RAM contents
[+X]@	EXP (— n)	Pre-incr. × indirect RAM fetch
[X-]@	EXP (— n)	Postdecr. × indirect RAM fetch
[>X]@ \$xx	EXP (— n)	Direct RAM fetch, × addressed
X!	EXP (d —)	Move 8-bit address to × register
[X]!	EXP (n —)	Indirect × store of RAM contents
[+X]!	EXP (n —)	Pre-incr. × indirect RAM store
[X-]!	EXP (n —)	Postdecr. × indirect RAM store
[>X]! \$xx	EXP (n —)	Direct RAM store, × addressed
Y@	EXP (— d)	Fetch current Y register contents
[Y]@	EXP (— n)	Indirect Y fetch of RAM contents
[+Y]@	EXP (— n)	Pre-incr. Y indirect RAM fetch
[Y-]@	EXP (— n)	Postdecr. Y indirect RAM fetch
[>Y]@ \$xx	EXP (— n)	Direct RAM fetch, Y addressed
Y!	EXP (d —)	Move address to Y register
[Y]!	EXP (n —)	Indirect Y store of RAM contents
[+Y]!	EXP (n —)	Pre-incr. Y indirect RAM store
[Y-]!	EXP (n —)	Postdecr. Y indirect RAM store
[>Y]! \$xx	EXP (n —)	Direct RAM store, Y addressed
>RP \$xx	EXP (—)	Set return stack pointer
>SP \$xx	EXP (—)	Set expression stack pointer
>X \$xx	EXP (—)	Set × register immediate
>Y \$xx	EXP (—)	Set Y register immediate

I. Hardware Description

II. Programming in qFORTH

III. Instruction Set

IV. qFORTH Language Dictionary

Addresses

4.1. Preface

This dictionary is written as a reference-guide for the programmers of the MARC4 microcontroller family.

The qFORTH DICTIONARY categorizes each qFORTH word and MARC4 assembler instruction according to its function (PURPOSE), category, stack effects and changes to the stack(s) by the instruction. The affected condition code flags, X and Y register changes are also described in detail. The length of each instruction is specified in the number of bytes generated at compile time. A short demonstration program for each instruction is also included

The language qFORTH is described in the section II **“Programming in qFORTH”** which includes a language tutorial and learners guide.

First time programmers of qFORTH are urged to read the Programmers Guide before consulting this guide.

The associated effects and changes which the listed qFORTH word will result in are described in this reference guide.

The entries are sorted in alphabetical order. You can find a reference in the index for unused MARC4 assembler mnemonics.

4.2. Introduction

Every entry in this dictionary is listed on a separate page.
The page structure for every entry contains the following topics:

1. "Purpose:"

This section gives a short explanation of each qFORTH vocabulary entry explaining its operational function.

2. "Category:"

Classification of the qFORTH vocabulary entries.

All entries in this dictionary are classified in the following categories (same categories are used in the 'MARC4 qFORTH Quick Reference Guide'):

A: Usage specific categories:

Arithmetic/logical

Arithmetic ('+', '-' ...), logical operations ('AND', 'OR') and bit manipulations ('ROR' ..) on 4-bit or 8-bit values.

Comparisons

Comparison operations on either single or double length values resulting in the BRANCH condition flag being set to determine the program flow ('>', '>=', ...).

Control structures

Control structures are used for conditional branches like IF ... ELSE ... THEN and loops (DO ... LOOP).

Interrupt handling

The MARC4 instruction set allows the programmer to handle up to 8 hardware/software interrupts, to enable/disable all interrupts. Other qFORTH words permit the programmer to determine the actual used depth or available free space on the expression and return stack.

Memory operations

Read, modify and store single, double or multiple length values in memory (RAM).

Stack operations

The sequence of the items, the number or the value of items held on the stack may be modified by stack operations. Stack operations may be of single, double or triple(12-bit) length ('SWAP', ...).

B: Language specific categories:

Assembler instructions

qFORTH programs may contain MARC4 native code instructions; all qFORTH words consist of assembler and/or qFORTH colon definitions and/or qFORTH macros (see there).

qFORTH colon definitions

All 'qFORTH colon definitions begin with a ':' and end with a ';'. They are processed like subroutines in other high level languages; that means, that they are "called" with a short (1) or long CALL (2 bytes) at execution time. The ';' will be translated to an EXIT instruction (return from subroutine). At execution time the program counter will be loaded with the calling address from the return stack. Colon definitions can be "called" from various program locations as opposed to qFORTH macros, which are placed "in-line" by the compiler at each "calling" address.

qFORTH macro definitions

All qFORTH macros begin with a 'CODE' and end with an 'END-CODE'. The compiler replaces the macro definition by in-line code.

Predefined data structures

Predefined data structures don't use any ROM-bytes (except ROM look-up tables), they are used for defining constants, variables or arrays in the RAM. With 'AT' you can force the compiler to place a qFORTH Word at a specific address in the ROM or a variable at a specific address in the RAM.

Compiler directives

Used to include other source files at compile time, define RAM or ROM sizes for the target device or to control the RAM or ROM placement (p.e. \$INCLUDE, \$RAMSIZE, \$ROMSIZE).

Most entries belong to a usage specific category and a language specific category. i.e. '+' belongs to the category arithmetic/logical and to the category assembler instructions; 'VARIABLE' belongs only to the category predefined data structures.

3. “Library implementation”

For qFORTH words which are not MARC4 assembler instructions, the assembly level implementation is included in the description. Refer to the library items “CODE” / “END-CODE” and “:” / “;” to better understand this representation.

These items will help when simulating/emulating the generated code with the simulator/emulator or when optimizing your program for ROM length is required.

The MARC4 native code is written in the dictionary for MARC4 assembler instructions.

The assembler mnemonic ‘(S)BRA’ means, that the compiler tries to optimise all BRA mnemonics to SBRA (short branches – only one byte), if the option is switched on and the optimizing is possible (page boundaries can’t be crossed by the SBRA, only by the BRA).

4. “Stack effect”

Describes what will happen to the expression and return stack, when executing the described instruction. See ‘stack related conventions’ to better understand the herein used syntax and semantics.

5. “Stack changes”

These lines include the number of elements, which will be popped from or pushed onto the stacks, when executing the instruction.

6. “Flags”

The “flags” part of each entry describes the flag effect of the instruction.

7. “X Y registers”

In this part the effect on the X and Y registers is described. This is only important, if the X or Y registers are explicitly referenced.

Note: The compiler optimizer changes the used code inside of colon definitions through the X/Y-register tracking technique.

Attention: The X register could be replaced in qFORTH macros by the Y register and vice versa (see the explanation of the optimizer in the qFORTH compiler user’s guide)!

8. "Bytes used"

Number of bytes used in the MARC4 ROM by the qFORTH colon definition, the qFORTH macro or by the assembler instruction.

Note: The optimizer of the compiler may shorten the actual program module.

9. "See also"

Includes similar qFORTH words or words in the same category. The '%' sign in this field signifies that there are no words similar for this entry.

10. "Example"

An example for using the described qFORTH word. All examples are tested and may be demonstrated with the MARC4 software development system.

4.3. Stack related conventions

Expression stack

Contains the program parameters, this stack is referred to as either the "EXP stack" or just "EXP", "data stack" or just "stack". 4-, 8- and 12-bit data elements are placed onto the stack with the least significant nibble on top.

Return stack

Contains the subroutine return addresses, loop indices and is also used to temporarily unload parameters from the expression stack. This stack is referred to as either the "RET stack" or just "RET".

X/Y-registers

The two general purpose 8-bit registers X and Y permit direct and indirect access (with additional pre-increment or post-decrement addressing modes) to all RAM cells.

Stack notation

addr	8-bit memory address
n	4-bit value (nibble — single length)
byte	8-bit value (represented as a double nibble)
d	8-bit unsigned integer (double length)
h m l	'higher middle lower' nibble of a 12-bit value.
t	12-bit memory/stack operation (triple length)
flags	the condition code register contains 3 flags: C% B I (see 1.4).

The stack effects shown in the dictionary represent the stack contents separated by two dashes (—) before and after execution of the instruction.

The top of stack (TOS) is always shown on the right. As an example, the SWAP and DUP instructions have the following expression stack effects:

	before:	after the operation.
	V	V
SWAP	EXP : (n2 n1 — n1 n2)	
DUP	EXP : (n1 — n1 n1)	
		TOS (top of stack)

A similar representation, specifying the stack effect of an instruction, shows the stack contents after execution.

Expression stack:

1	2	2	TOS	SWAP	1	TOS	DUP	1	TOS
Push two		1		Swap top	2		Duplicate	1	
constants		?		two ele-	?		top elements	2	
		.		ments	.			.	
		.			.			.	

Return stack notation:

A return stack entry contains a maximum of 3 nibbles on each level (normally a 12-bit ROM address).

If (e.g. in a DO..LOOP) only 2 nibbles of 3 possibles are required, there is 'u' for 'undefined' or 'don't care' used in the notation:

3 1 DO (RET: — u|limit|index) or (RET: — u|3|1)

4.4. Flags and condition code register

There are three flags, which interact with qFORTH instructions. Together with a fourth flag, which is reserved for TEMIC, they are accessible via a 4-bit register ("condition code register" – "CCR").

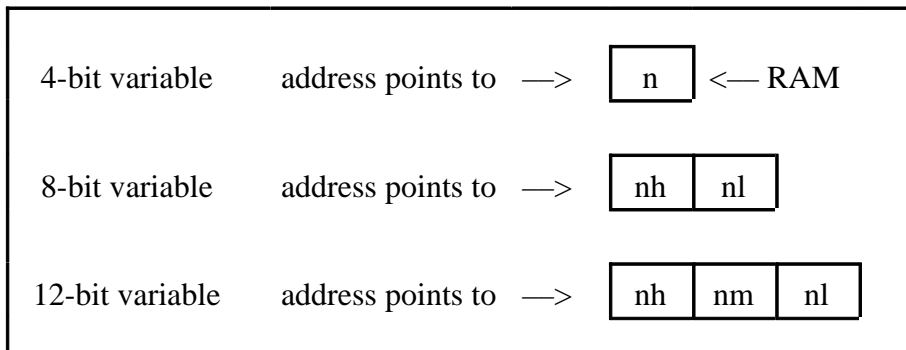
A binary value 1 indicates the corresponding flag has been set, a binary value 0 indicates a cleared flag.

The order of the flags in the CCR is as follows: used in text:

Carry	C	bit 3	CARRY flag (MSB)
%	%	bit 2	(reserved)
Branch	B	bit 1	BRANCH flag
Interrupt enable	I	bit 0	I_ENABLE flag (LSB)

4.5. MARC4 memory addressing model

Memory operations



nh = most significant nibble
 nl = least significant nibble

See the entries 2/VARIABLE, 2/L/ARRAY and 2/3@ for further information.

The following example shows you, how to handle an 8-bit variable:

```
1 CONSTANT n_low          ( constant and variable declaration )
2VARIABLE KeyPressTime    ( 8-bit variable )
```

: Example

```
0 0 KeyPressTime 2!        ( initialise this variable )
KeyPressTime 2@ 1 M+      ( increment by 1 the 8-bit variable )
                          ( the lower nibble is on top of )
IF DROP 1 THEN           ( reset to 01h on overflow )
KeyPressTime 2!          ( store the new 8-bit value back )
;
```

4.6. Short form dictionary

MARC4 – Control commands

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
AGAIN	3					CY	B	
BEGIN	0							
DO	1	limit index—	— limit index					
#DO	1	index—	— u u index					
?DO	5	limit index—	— u limit index			CY	B	
LOOP	9	— (n1 n2 n3) —	— (-1 level) —			CY	B	
#LOOP	4		u u index— u u index-1				B	
+LOOP	10	n —	u limit index— u limit index+n			CY	B	
?LEAVE	2							
-?LEAVE	3						B	
REPEAT	3					CY	B	
UNTIL	3						B	
WHILE	3						B	
CASE	0							
ELSE	3					CY	B	
ENDCASE	1	n —						
ENDOF	3					CY	B	
IF	3						B	
OF	4	n1 — n1n2 —				CY	B	
THEN	0					CY	B	
CCR@	1	— n						
CCR!	1	n —				CY	B	I
CLR_BCF	2	-(0)-				CY	B	
EI	2					CY	B	I
EXIT	1		oldPC —					
DI	1							I
SET_BCF	1							I
SWI0.... SW17	4	— (2 level) —						I
TOG_BF	1						B	

MARC4 – Stack commands

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
DECR	1		u u n — u u n-1				B	
DEPTH	9	-(SPh SPI S0h S0l)—n	— (1 level) —			CY	B	
DROP	1	n1 —						
2DROP	2	n1 n2 —						
3DROP	3	n1 n2 n3 —						
DROPR	1		u u u —					
DUP	1	n1 — n1 n1						
?DUP	5	n — n n				CY	B	
2DUP	2	d — d d						
3DUP	4	n1n2n3 —n1n2n3n1n2n3	-(n1 n2 n3)-					
I	1	— index	u u index—u u index					
INDEX	8	n addr —	— (1 level) —			CY	B	
J	6	— J	— (-1 level) —					
NIP	2	n1 n2 — n2						
2NIP	4	d1 d2 — d2						
OVER	1	n2 n1 — n2 n1 n2						
2OVER	8	d1 d2 —d1 d2 d1	— (2 level u n2 n1) —					
PICK	13	x -(2 level)- n[x]	— (2 level) —	X	Y	CY	B	
R@	1	— n2	u u n1 — u u n1					
2R@	1	— n1 n2	u n1 n2 — u n1 n2					
3R@	1	— n1 n2 n3	n3 n2 n1—n3 n2 n1					
>R	1	n1	— u u n1					
2>R	1	n1 n2 —	—u n2 n1					
3>R	1	n1 n2 n3 —	— n3 n2 n1					
R>	2	— n	u u n —					
2R>	2	— n1 n2	u n2 n1 —					
3R>	2	— n1 n2 n3	n3 n2 n1 —					
RDEPTH	13	-(2 level)- n	— (1 level) —			CY	B	
RFREE	30	-(3 level)- n	— (2 level) —			CY	B	
ROT	1	n1 n2 n3 — n2 n3 n1						
2ROT	8	d1 d2 d3 — d2 d3 d1	— (1 level u d1) —					
<ROT	2	n1 n2 n3 — n3 n1 n2						
2<ROT	14	d1 d2 d3 — d3 d1 d2	— (4 level) —					
RP@	1	— RPh RPl						
RP!	1	RPh RPl —						
SP@	1	— SPH SPI+1						
SP!	1	SPH SPI —						
SWAP	1	n2 n1 — n1 n2						
2SWAP	8	d1 d2 — d2 d1	— (1 level u u d2l) —					
TUCK	2	n1 n2 — n2 n1 n2						
2TUCK	6	d1 d2 — d2 d1 d2	— (1 level d1l d2) —					

MARC4 – Mathematic commands

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
ADD	1	n1 n2 — n1+n2				CY	B	
+	1	n1 n2 — n1+n2				CY	B	
+	4	n addr —		X	Y	CY	B	
INC	1	n — n+1					B	
1+	1	n — n+1					B	
1+	4	addr —		X	Y		B	
ADDC	1	n1 n2 — n1+n2+CY				CY	B	
+C	1	n1 n2 — n1+n2+CY				CY	B	
D+	7	d1 d2 — d1+d2	— (1 level) —			CY	B	
D+!	8	d addr —	— (1 level) —	X	Y	CY	B	
M+	5	d1 n — d1+n	— (1 level) —			CY	B	
T+!	19	nh nm nl addr — (1 level) —	— (2 level) —	X	Y	CY	B	
TD+!	20	d addr — (1 level) —	—(2 level)—	X	Y	CY	B	
DAA	1	n — n+6				CY	B	
SUB	1	n1 n2 — n1-n2				CY	B	
-	1	n1 n2 — n1-n2				CY	B	
DEC	1	n — n-1					B	
1-	1	n — n-1					B	
1-!	4	addr —		X	Y		B	
SUBB	1	n1 n2 — n1+/n2+CY				CY	B	
-C	1	n1 n2 — n1+/n2+CY				CY	B	
D-	8	d1 d2 — d1-d2	— (1 level) —			CY	B	
D-!	10	d addr —	— (1 level) —	X	Y	CY	B	
M-	5	d1 n — d1-n	— (1 level) —			CY	B	
T-!	22	nh nm nl — (1 level) —	— (2 level) —	X	Y	CY	B	
TD-	22	d addr — (1 level) —	— (2 level) —	X	Y	CY	B	
DAS	3	n — 9-n				CY	B	
2*	1	n — n*2				CY	B	
D2*	4	d — d*2				CY	B	
2/	1	n — n/2				CY	B	
D2/	4	d — d/2				CY	B	
CMP_EQ	1	n1 n2 — n1				CY	B	
=	2	n1 n2 —				CY	B	
0=	3	n —				CY	B	
D=	13	d1 d2 —	— (1 level) —			CY	B	

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
D0=	2	d —					B	
CMP_GE	1	n1 n2 — n1				CY	B	
>=	2	n1 n2 —				CY	B	
D>=	19	d1 d2 —	— (1 level u d2h d2l) —			CY	B	
CMP_GT	1	n1 n2 — n1				CY	B	
>	2	n1 n2 —				CY	B	
D>	16	d1 d2 —	— (1 level u d2h d2l) —			CY	B	
CMP_LE	1	n1 n2 — n1				CY	B	
<=	2	n1 n2 —				CY	B	
D<=	19	d1 d2 —	— (1 level u d2h d2l) —			CY	B	
CMP_LT	1	n1 n2 — n1				CY	B	
<	2	n1 n2 —				CY	B	
D<	16	d1 d2 —	— (1 level u d2h d2l) —			CY	B	
CMP_NE	1	n1 n2 — n1				CY	B	
<>	2	n1 n2 —				CY	B	
0<>	3	n —				CY	B	
D0<>	3	d —					B	
D<>	10	d1 d2 —	— (1 level) —			CY		
MAX	7	n1 n2 — nmax	— (1 level) —			CY	B	
D MAX	30	d1 d2 —d1 d1 d2— dmax	— (3 level) —			CY	B	
MIN	7	n1 n2 — nmin	— (1 level) —			CY	B	
D MIN	30	d1 d2 —d1 d1 d2— dmin	— (3 level) —			CY	B	
NEGATE	2	n1 — -n1					B	
D NEGATE	8	d — -d	— (1 level) —			CY	B	
NOT	1	n1 — /n1					B	
ROL	1					CY	B	
ROR	1					CY	B	
SHL	1	n — n*2				CY	B	
SHR	1	n — n/2				CY	B	
AND	1	n1 n2 — n1^n2					B	
OR	1	n1 n2 — n1 v n2					B	
XOR	1	n1 n2 — n1 xor n2					B	
TOGGLE	4	n1 addr —		X	Y		B	
D>S	2	d — n						
S>D	2	n — d						

MARC4 – Memory commands

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
@	2	addr — n		X	Y			
2@	3	addr — nh nl		X	Y			
3@	4	addr — nh nm nl		X	Y			
X@	1	— xh xl						
[X]@	1	— n						
[+X]@	1	— n		X				
[X-]@	1	— n		X				
Y@	1	— yh yl						
[Y]@	1	— n						
[+Y]@	1	— n			Y			
[Y-]@	1	— n			Y			
DTABLE@	14	ROMaddr n — conh conl	— (2 level) —			CY	B	
ROM-BYTE@	2	ROMaddr — consth constl -(2 level)-						
!	2	n addr—		X	Y			
2!	4	nh nl addr —		X	Y			
3!	7	nh nm nl —	— (1 level) —	X	Y			
X!	1	xh xl —		X				
[X]!	1	n —						
[+X]!	1	n —		X				
[X-]!	1	n —		X				
Y!	1	yh yl —			Y			
[Y]!	1	n —						
[+Y]!	1	n —			Y			
[Y-]!	1	n —			Y			
ERASE	14	addr n—	— (2 level) —	X	Y		B	
FILL	24	addr n1 n2 —	— (3 level) —	X	Y		B	
MOVE	14	n from to —	— (2 level) —	X	Y			
MOVE>	10	n from to —	— (2 level) —	X	Y			
IN	1	port — data					B	
OUT	1	data port —						

MARC4 – commands

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
!	2	n addr—		X	Y			
'	3	—ROMaddr						
#DO	1	index—	— u u index					
#LOOP	4		u u index— u u index-1				B	
+LOOP	10	n —	u limit index— u limit ind+n			CY	B	
-?LEAVE	3						B	
<ROT	2	n1 n2 n3 — n3 n1 n2						
>RPxxh	2							
>SPxxh	2							
?DO	5	limit index—	— u limit index			CY	B	
?DUP	5	n — n n				CY	B	
?LEAVE	2							
@	2	addr — n		X	Y			
[+X]!	1	n —		X				
[+X]@	1	— n		X				
[+Y]!	1	n —			Y			
[+Y]@	1	— n			Y			
[X-]!	1	n —		X				
[X-]@	1	— n		X				
[X]!	1	n —						
[X]@	1	— n						
[Y-]!	1	n —			Y			
[Y-]@	1	— n			Y			
[Y]!	1	n —						
[Y]@	1	— n						
2!	4	nh nl addr —		X	Y			
2<ROT	14	d1 d2 d3 — d3 d1 d2	— (4 level) —					
2@	3	addr — nh nl		X	Y			
2DROP	2	n1 n2 —						
2DUP	2	d — d d						
2NIP	4	d1 d2 — d2						
2OVER	8	d1 d2 —d1 d2 d1	--(2 level u n2 n1) —					
R@	1	— n2	u u n1 — u u n1					

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
2R@	1	— n1 n2	u n1 n2 — u n1 n2					
2ROT	8	d1 d2 d3 — d2 d3 d1	— (1 level u d1) —					
2SWAP	8	d1 d2 — d2 d1	— (1 level u u d2l) —					
2TUCK	6	d1 d2 — d2 d1 d2	— (1 level d1l d2) —					
3!	7	nh nm nl addr—	— (1 level) —	X	Y			
3@	4	addr — nh nm nl		X	Y			
3DROP	3	n1 n2 n3 —						
3DUP	4	n1n2n3 — n1n2n3n1n2n3	— (n1 n2 n3) —					
3R@	1	— n1 n2 n3	n3 n2 n1 — n3 n2 n1					
DAA	1	n — n+6				CY	B	
ADD	1	n1 n2 — n1+n2				CY	B	
+	1	n1 n2 — n1+n2				CY	B	
+	4	n addr —		X	Y	CY	B	
INC	1	n — n+1					B	
1+	1	n — n+1					B	
1+!	4	addr —		X	Y		B	
ADDC	1	n1 n2 — n1+n2+CY				CY	B	
+C	1	n1 n2 — n1+n2+CY				CY	B	
D+	7	d1 d2 — d1+d2	— (1 level) —			CY	B	
D+!	8	d addr —	— (1 level) —	X	Y	CY	B	
DAS	3	n — 9-n				CY	B	
SUB	1	n1 n2 — n1-n2				CY	B	
-	1	n1 n2 — n1-n2				CY	B	
DEC	1	n — n-1					B	
1-	1	n — n-1					B	
1-!	4	addr —		X	Y		B	
SUBB	1	n1 n2 — n1+/n2+CY				CY	B	
-C	1	n1 n2 — n1+/n2+CY				CY	B	
D-	8	d1 d2 — d1-d2	— (1 level) —			CY	B	
D-!	10	d addr —	— (1 level) —	X	Y	CY	B	
2*	1	n — n*2				CY	B	
D2*	4	d — d*2				CY	B	
2/	1	n — n/2				CY	B	
D2/	4	d — d/2				CY	B	
AGAIN	3					CY	B	
AND	1	n1 n2 — n1^n2					B	
BEGIN	0							

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
CASE	0	n — n						
CCR!	1	n —				CY	B	I
CCR@	1	— n						
CLR_BCF	2	-(0)-				CY	B	
CMP_EQ	1	n1 n2 — n1				CY	B	
=	2	n1 n2 —				CY	B	
0=	3	n —				CY	B	
D=	13	d1 d2 —	— (1 level) —			CY	B	
D0=	2	d —					B	
CMP_GE	1	n1 n2 — n1				CY	B	
>=	2	n1 n2 —				CY	B	
D>=	19	d1 d2 —	--(1 level u d2h d2l)--			CY	B	
CMP_GT	1	n1 n2 — n1				CY	B	
>	2	n1 n2 —				CY	B	
D>	16	d1 d2 —	--(1 level u d2h d2l)--			CY	B	
CMP_LE	1	n1 n2 — n1				CY	B	
<=	2	n1 n2 —				CY	B	
D<=	19	d1 d2 —	--(1 level u d2h d2l)--			CY	B	
CMP_LT	1	n1 n2 — n1				CY	B	
<	2	n1 n2 —				CY	B	
D<	16	d1 d2 —	--(1 level u d2h d2l)--			CY	B	
CMP_NE	1	n1 n2 — n1				CY	B	
<>	2	n1 n2 —				CY	B	
0<>	3	n —				CY	B	
D<>	10	d1 d2 —	— (1 level) —			CY	B	
D0<>	3	d —					B	
DECR	1		u u n — u u n-1				B	
DEPTH	9	-(SPh SPI S0h S0l)— n	— (1 level) —			CY	B	
DI	1							I
DMAX	30	d1 d2 —d1 d1 d2— dmax	— (3 level) —			CY	B	
DMIN	30	d1 d2 —d1 d1 d2— dmin	— (3 level) —			CY	B	
DNEGATE	8	d — -d	— (1 level) —			CY	B	
DO	1	limit index—	— limit index					
DROP	1	n1 —						
DROPR	1		u u u —					

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
DTABLE@	14	ROMaddr — const.h const.l	— (2 level) —			CY	B	
DTOGGLE	8	d addr —	— (1 level) —	X	Y		B	
DUP	1	n1 — n1 n1						
EI	2					CY	B	I
ELSE	3					CY	B	
ENDCASE	1	n —						
ENDOF	3					CY	B	
ERASE	14	addr n—	— (2 level) —	X	Y		B	
EXIT	1		oldPC —					
EXECUTE		ROMaddr	— (2 level) —					
FILL	24	addr n1 n2 —	— (3 level) —	X	Y		B	
I	1	— index	u u index—u u index					
IF	3						B	
IN	1	port — data					B	
INDEX	8	n addr —	— (1 level) —			CY	B	
J	6	— J	— (-1 level) —					
LOOP	9	-(n1 n2 n3)-	— (-1 level) —			CY	B	
M+	5	d1 n — d1+n	— (1 level) —			CY	B	
M-	5	d1 n — d1-n	— (1 level) —			CY	B	
MAX	7	n1 n2 — nmax	— (1 level) —			CY	B	
MIN	7	n1 n2 — nmin	— (1 level) —			CY	B	
MOVE	14	n from to —	— (2 level) —	X	Y			
MOVE>	10	n from to —	— (2 level) —	X	Y			
NEGATE	2	n1 — -n1					B	
NIP	2	n1 n2 — n2						
D>S	2	d — n						
NOP	1							
NOT	1	n1 — /n1					B	
OF	4	n1 n2 —				CY	B	
OR	1	n1 n2 — n1 v n2					B	
OUT	1	data port —						
OVER	1	n2 n1 — n2 n1 n2						
PICK	13	x -(2 level)- n[x]	— (2 level) —	X	Y	CY	B	
>R	1	n1	— u u n1					
2>R	1	n1 n2 —	—u n2 n1					
3>R	1	n1 n2 n3 —	— n3 n2 n1					

Command	Byte	Expression stack	Return stack	X	Y	CY	B	I
R>	2	— n	u u n —					
2R>	2	— n1 n2	u n2 n1 —					
3R>	2	— n1 n2 n3	n3 n2 n1 —					
RDEPTH	13	-(2 level)- n	— (1 level) —			CY	B	
REPEAT	3					CY	B	
RFREE	30	-(3 level)- n	— (2 level) —			CY	B	
ROL	1					CY	B	
ROLL	57	x -(2 level)-	— (4 level) —	X	Y	CY	B	I
ROM-BYTE@	2	ROMaddr — consth constl	— (2 level) —					
ROR	1					CY	B	
ROT	1	n1 n2 n3 — n2 n3 n1						
RP!	1	RPh RPl —						
RP@	1	— RPh RPl						
S>D	2	n — d						
SET_BCF	1					CY	B	
SHL	1	n — n*2				CY	B	
SHR	1	n — n/2				CY	B	
SP!	1	SPh SPi —						
SP@	1	— SPH SPi+1						
SWAP	1	n2 n1 — n1 n2						
SWI0..SW17	4	— (2 level) —						I
T+!	19	nh nm nl addr— (1 level) —	— (2 level) —	X	Y	CB	B	
T-!	22	nh nm nl addr —	— (2 level) —	X	Y	CY	B	
TD+!	20	d addr — (1 level) —	— (2 level) —	X	Y	CY	B	
TD-!	22	d addr — (1 level) —	— (2 level) —	X	Y	CY	B	
THEN	0					CY	B	
TOG_BF	1						B	
TOGGLE	4	n1 addr —		X	Y		B	
TUCK	2	n1 n2 — n2 n1 n2						
UNTIL	3						B	
WHILE	3						B	
X!	1	xh xl —		X				
X@	1	— xh xl						
XOR	1	n1 n2 — n1 xor n2			Y			
Y!	1	yh yl —						
Y@	1	— yh yl						

4.7. Index

Symbols

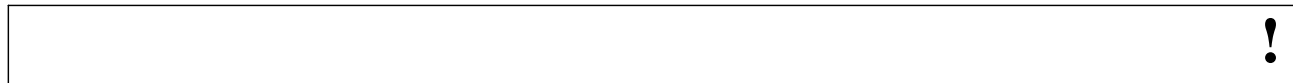
!	105
?DO	217
?DUP	219
?LEAVE	221
; EXIT RTI	199
:	197
' (COMMENT)'\	115
'0 ... Fh (15)' 'LIT_0 ... LIT_F'	131
'INT0 ... INT7'	333
#DO	107
#LOOP	109
\$ NCLUDE \$RAMSIZE \$ROMSIZE \$SHIFT \$INTS	247
\$AUTOSLEEP SLEEP	111
\$RESET	113
@	223
+ ADD	117
+!	119
+C ADDC	121
+LOOP	123
- SUB	125
-?LEAVE	127
-C SUBB	129
= CMP_EQ	209
> CMP_GT	211
>= CMP_GE	213
>R	215
< CMP_LT	201
<= CMP_LE	203
< ROT	207
<> CMP_NE	205

Numbers

0<>	133
0=	135
1+ INC	137
1+!	139
1- DEC	141
1-!	143
2<ROT	151
2!	145
2@	155
2* SHL	147
2/ SHR	149
2>R	153
2ARRAY	157
2CONSTANT	159
2DROP	161
2DUP	163
2LARRAY	165
2NIP	167
2OVER	169
2R@	173
2R>	171
2ROT	175
2SWAP	177
2TUCK	179
2VARIABLE	181
3!	183
3@	187
3>R	185
3DROP	189
3DUP	191
3R@	195
3R>	193

A		DAS	283
AGAIN	225	DECR	285
ALLOT	227	DEPTH	287
AND	229	DI	289
ARRAY	231	DMAX	291
AT	233	DMIN	293
B		DNEGATE	295
BEGIN	235	DO	297
C		DROP	299
CASE	237	DROPR	301
CCR!	239	DTABLE@	303
CCR@	241	DTOGGLE	305
CLR_BCF	243	DUP	307
CODE	245	E	
CONSTANT	249	EI	309
D		ELSE	311
D<	267	END-CODE	313
D<=	269	ENDCASE	315
D<>	271	ENDOF	317
D+	251	ERASE	319
D+!	253	EXIT	321
D-	255	F	
D-!	257	FILL	323
D=	273	I	
D>	275	I R@	325
D>=	277	IF	327
D>S NIP	279	IN	329
D0<>	259	INDEX	331
D0=	261	J	
D2*	263	J	335
D2/	265		
DAA	281		

L		ROT	385
LARRAY	337		
LOOP	339		
M		S	
M+ M-	341	S>D	389
MAX	343	SET_BCF	391
MIN	345	'Stack Pointer Operations'	393
MOVE	347	'SWI0 ... SWI7'	397
MOVE>	349	SWAP	395
		T	
N		T+!	399
NEGATE	351	T-!	401
NOP	353	TD+!	403
NOT	355	TD-!	405
		THEN	407
O		TOG_BF	411
OF	357	TOGGLE	409
OR	359	TUCK	413
OVER	363		
OUT	361	U	
		UNTIL	415
P		V	
PICK	365	VARIABLE	417
		W	
R		WHILE	419
R>	367		
RDEPTH	369	X	
REPEAT	371	X-Register-Operations	421
RFREE	373	XOR	423
'Return Stack Pointer Operations'	387		
ROL	375	Y	
ROLL	377	Y-Register-Operations	425
ROMByte@TABLE;;	379		
ROMCONST	381		
ROR	383		



“Store”

Purpose:

Stores a 4-bit value at a specified memory location.

Category: qFORTH macro

Library implementation: CODE ! Y! (n addr — n)
 [Y]! (n —)
 END-CODE

Changes in the really generated code sequence can result through the compiler optimizing techniques (register tracking)!

Stack effect: EXP (n RAM_addr —)
 RET (—)

Stack changes: EXP: 3 elements are popped from the stack
 RET: not affected

Flags: not affected

X Y registers: The contents of the Y or X register may be changed.

Bytes used: 2 (1 when optimised)

See Also: +! 2! 3! @

Example:

```
VARIABLE ControlState
VARIABLE Semaphore

: InitVariables      ( initialise VARs      )
  0 ControlState !  ( Setup control flag  )
  2 Semaphore !     ( Setup a preset value )
;
```

#DO

"Hash-DO"

Purpose:

#DO indicates the start of an iterative 'decrement-if-nonzero' loop structure. It is used only within a macro colon definition in a pair with #LOOP. The value on top of the stack at the time #DO is executed determines the number of times the loop repeats. The value on top is the initial loop index which will be decremented on each iteration of the loop (see example 2).

If the current loop index *I* is not accessed inside of a loop block, this control structure executes much faster than an equivalent `n 0 DO ... LOOP`. The standard FORTH-83 loop structure `n 0 DO ... -1 +LOOP` maps directly to the behaviour of the `n #DO ... #LOOP` structure.

Category: Control structure / qFORTH macro

Library implementation: CODE #DO >R
\$#DO:
END-CODE

Stack effect: EXP (Index —)
RET (— u|u|Index)

Stack changes: EXP: 1 element is popped from the stack
RET: 1 element is pushed onto the stack

Flags: #DO : not affected
#LOOP : CARRY flag not affected
BRANCH flag = Set, if (Index-1 <> 0)

X Y registers: not affected

Bytes used: 1

See also: #LOOP ?LEAVE -?LEAVE

Example 1:

```
8 ARRAY result      ( 8 digit BCD number      )
: ERASE              ( Fill a block of memory with zero  )
  <ROT Y!            ( addr count —                )
  0 [Y]! 1-          ( count — count-1                )
  #DO
    0 [+Y]!          ( Use the MARC4 pre-incremented store )
  #LOOP              ( REPEAT until length-1 = 0          )
;
: Clear result
Result 8 ERASE      ( Clear result array                )
```

Example 2:

```
1 CONSTANT port1
: HASH-DO-LOOP
0 #DO                ( loop 16 times                )
  I 1- port1 OUT     ( write data to 'port1': F, E, D, C...1,0. )
  #LOOP              ( repeat the loop.                )
;
```

#LOOP

“Hash-LOOP”

Purpose:

#LOOP indicates the end of an iterative 'decrement-if-nonzero' loop structure.

#LOOP is used only within a colon definition in a pair with #DO. The value on top of the stack at the time #DO is executed determines the number of times the loop repeats. The loop index is decremented on the return stack on each iteration, ie. the execution of the #LOOP word, until the index reaches zero. If the new index is decremented to zero, the loop is terminated and the loop index is discarded from the return stack. Otherwise, control branches back to the word just after the corresponding #DO word.

If the current loop index I is not used inside a loop block this structure executes much faster than an equivalent DO ...LOOP. The behaviour of the standard FORTH loop structure 0 DO ... -1 +LOOP is identical to the #DO ... #LOOP structure.

Category: Control structure / qFORTH macro

Library implementation: CODE #LOOP DECR (decrement loop index on RET)
 (S)BRA _\$#DO
 _\$LOOP: DROPR (drop loop index from RET)
 END-CODE

Stack effect: EXP (—)
 IF Index-1 > 0 THEN RET (u|u|Index —u|u|Index-1)
 ELSE RET (u|u|Index —)

Stack changes: EXP: not affected
 RET: If #LOOP terminates, top element is popped from the stack

Flags: CARRY flag not affected
 BRANCH flag = set, if (index-1 <> 0)

X Y registers: not affected

Bytes used: 3 - 4

See also: #DO ?LEAVE -?LEAVE

Example:

16 ARRAY LCD-buffer

```
: FILL-IT Y!           ( n count addr — n count  )
  #DO                 ( end address was on stack )
    DUP [Y-]!        ( duplicate and store value )
  #LOOP
```

```
; Setup_Full_House
  Fh 0 LCD-buffer [15] FILL-IT
;
```


\$AUTOSLEEP	SLEEP
-------------	-------

“Dollar–autosleep”

Purpose:

The \$AUTOSLEEP function will automatically be placed at ROM address \$000 by the compiler and is NOT re–definable by the user. The return stack pointer is initialized in \$RESET to FCh.. After the last interrupt routine is processed and no other interrupt is pending, the PC will be automatically loaded to the address \$000 (\$AUTOSLEEP). This forces the MARC4 into sleep mode through processing the \$AUTOSLEEP routine.

This sleep mode is a shutdown condition which is used to reduce the average system power consumption, whereby the CPU is halted as well as the internal clocks.

The internal RAM data keeps valid during sleep mode. To wake up the CPU again, an interrupt must be received from a module (timer/counter, external interrupt pin or other modules). The CPU starts running at the ROM address, where the to the interrupt belonging INTx routine is placed.

Attention ! : It is not recommended to use the SLEEP instruction otherwise then in the \$RESET or \$AUTOSLEEP, because it might result in unwanted side effects in other interrupt routines !

Category: interrupt handling / qFORTH macro / assembler instruction

MARC4 opcode: 0F hex (SLEEP)

Stack effect: EXP & RET empty

Stack changes: EXP: not affected
RET: not affected

Flags: SLEEP sets the I_ENABLE flag;
CARRY and BRANCH flags are set by \$AUTOSLEEP

X Y registers: not affected

Bytes used: 4

See also: \$RESET, INT0 ... INT7, interrupt handling (DI EI RTI)

Example:

(qFORTH word \$AUTOSLEEP will be placed at address 000h in ROM)

```
:$AUTOSLEEP      ( Library implementation:      )
  _$Tired: NOP    ( Gives time to the Interrupt logic.  )
    SLEEP        ( less than 1 uA will be used.    )
                ( checks for new interrupt request )
    SET_BCF      ( for security: if there is only a  )
    BRA _$Tired  ( glitch, but no real interrupt.   )
    [ E 0 R 0 ]  ( suppress the compiler warnings.   )
    ;;          ( no EXIT or RTI is necessary.    )
```

\$RESET

“Dollar–reset”

Purpose:

The power-on-reset colon definition \$RESET is placed at ROM address \$008 automatically and is re-definable by the user. The maximum length of this part in the Zero Page is 56 bytes, when INT0 is used too.

It is normally used to initialize the two stack pointers, as well as the connected I/O devices, like timer/counter, LCD and A/D-converter.

An optional SELFTEST is executable on every power-on-reset, if port 0 is forced to zero (customer specific reset routine).

Category: Interrupt handling / predefined qFORTH colon definition

Stack effect: EXP stack pointer initialized
RET stack pointer initialized

Stack changes: EXP: empty
RET: empty

Flags: CARRY flag Undefined after power-on reset
BRANCH flag Undefined after power-on reset
I_ENABLE flag Reset by hardware during POR–
Set by the RTI instruction at the end of \$RESET.

X Y registers: not affected

Bytes used: modified by the customer

See also: \$AUTOSLEEP

Example:

```
0 CONSTANT port0
FCh 2CONSTANT NoRAM
VARIABLE S0 16 ALLOT ( Define expression stack space. )
VARIABLE R0 31 ALLOT ( Define RET stack )

: $RESET ( Possible $RESET implement. of a customer )
  >RP NoRAM ( Init RET stack pointer to non-existent memory )
  >SP S0 ( Init EXP stack pointer; above RET stack )
  Port0 IN 0=
  IF
  RAM_Test
  ROM_Test
  THEN
  Reset_LCD
  Init_Timer
; ( RTI enable Interrupts, goto $AUTOSLEEP )
```

(comment) \

“Paren”, ”Backslash”

Purpose:

Begins a comment, used either in the form

(ccccc) or \ comment is rest of line

The characters “cccc” delimited by the closing parenthesis are considered a comment and are ignored by the qFORTH compiler. The characters ”comment is rest of line” are delimited by the end of line control character(s).

NOTE: The ' (' and ' \ ' characters must be immediately preceded and followed by a blank. Comments should be used freely for documenting programs. They do not affect the size of the compiled code.

Category: predefined data structure

Stack effect: EXP (—)
 RET (—)

Stack changes: EXP: not affected
 RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 0

See also: %

Example:

```
: ExampleWord
  1 3          ( — 1 3          )
  DUP         ( ***** This is a qFORTH comment ***** )
  ROT         \ This is a comment 'til the end of line.
  SWAP        ( 3 3 1 — 3 1 3   )
  3DROP       ( clean table again. )
;             ( End of ':'-definition )
```

+ **ADD**

“Plus”

Purpose:

Add the top two 4-bit values and replace them with the 4-bit result on top of the stack.

Category (+) : Arithmetic/logical (single-length)
 (ADD): MARC4 mnemonic

MARC4 opcode: 00 hex

Stack effect: EXP (n1 n2 — n1+n2)
 RET (—)

Stack changes: EXP: stack depth reduced by 1, new top element.
 RET: not affected

Flags: CARRY flag Set on arithmetic overflow (result > 15)
 BRANCH flag = CARRY flag

X Y registers: not affected

Bytes used: 1

See also: D+! 1+! 1-! - +! +C -C

Example:

VARIABLE Result

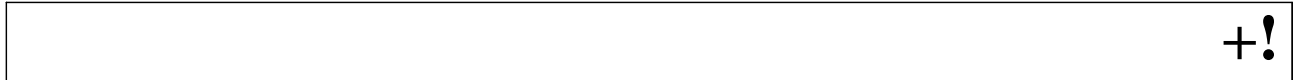
: Single-addition

5 3 + (RPN addition: 5 + 3 := 8)

Result ! (Save result in memory)

3 Result +! (Add 3 to memory location)

;



“Plus-store ”

Purpose:

Adds a 4-bit value to the contents of a 4-bit variable. On entry to the function, the TOS value is the 8-bit RAM address of the variable.

Category: Memory operation (single-length) / qFORTH macro

Library implementation: CODE +! Y! (n addr — n)
 [Y]@ + (n — n @RAM[Y] — sum)
 [Y]! (sum —)
 END-CODE

The qFORTH compiler optimizes a word sequence like ”5 semaphore +!” into an instruction sequence of the following form :

```
[>Y]@ semaphore
ADD [Y]!
```

Stack effect: EXP (n RAM_addr —)
 RET (—)

Stack changes: EXP: 3 elements are popped from the stack
 RET: not affected

Flags: CARRY flag: Set on arithmetic overflow (result > 15)
 BRANCH flag = CARRY flag

X Y registers: The contents of the Y or X register may be changed.

Bytes used: 4

See also: + !

Example:

VARIABLE Ramaddress

: PLUS-STORE

15 Ramaddress ! (15 is stored in the variable)

9 Ramaddress +! (9 is added to this variable)

;

	+C	ADDC
--	-----------	-------------

“Plus-C”

Purpose:

ADD with CARRY of the top two 4-bit values and replace them with the 4-bit result [n1 + n2 + CARRY] on top of the stack.

Category:

(+C) : arithmetic/logical (single-length)
(ADDC) : MARC4 mnemonic

MARC4 opcode:

01 hex

Stack effect:

EXP (n1 n2 — n1+n2+CARRY)
RET (—)

Stack changes:

EXP: 1 element is popped from the stack
RET: not affected

Flags:

CARRY flag: Set on arithmetic overflow (result > 15)
BRANCH flag = CARRY flag

X Y registers:

not affected

Bytes used:

1

See also:

-C + DAA ADD

Examples:

```
: Overflow 10 8 +C;                ( — 2 ; BRANCH, CARRY flag set      )
: PLUS-C
  SET_BCF 4 3 +C                    ( — 8 ; flags : ————— )
;
: D+ \ 8-bit addition using +C      (d1 d2 — d1+d2                      )
  ROT +
  <ROT +C                           ( more comment see 'D+' !          )
  SWAP
;
: ADDC-Example
  50 190 D+                          ( — 240 = F0h ; no CARRY or BRANCH  )
  PLUS-C
  Overflow
  2DROP 2DROP                        ( the results.                      )
;
```

+LOOP

“Plus-LOOP”

Purpose:

+LOOP terminates a DO loop. Used inside a colon definition in the form DO ... n +LOOP. On each iteration of the DO loop, +LOOP increments the loop index by n. If the new index is incremented across the limit (\geq), the loop is terminated and the loop control parameters are discarded. Otherwise, execution returns just after the corresponding DO.

Category: Control structure / qFORTH macro

Library implementation:

```
CODE +LOOP 2R>( Move Limit & Index on EXP )
    ROT + OVER
    CMP_LT ( Check for Index < Limit )
    2>R
    (S)BRA _$DO
    _$LOOP: DROPR ( Skip Limit & Index from RET )
END-CODE
```

Stack effect: EXP (n —)
 IF Index+n < Limit
 THEN RET (u|Limit|Index — u|Limit|Index+n)
 ELSE RET (u|Limit|Index —)

Stack changes: EXP: top element is popped from the stack
 RET: top entry is popped from the stack, if +LOOP is terminated

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 10

See also: DO ?DO #DO #LOOP LOOP I J ?LEAVE -?LEAVE

Example:

```
: INCREMENT-COUNT
  10 0 DO
    I
    2 +LOOP      ( NOTE: The BRANCH and CARRY flag are      )
                  ( altered during execution of +LOOP      )
;                ( EXP after execution — 8 6 4 2 0      )
```

	—	SUB
--	----------	------------

“Minus”

Purpose:

2's complement subtract the top two 4-bit values and replace them with the result $[n1 + /n2 + 1]$ on top of the stack ($/n2$ is the 1's complement of $n2$).

Category: (–) :arithmetic/logical (single-length)
 (SUB) : MARC4 mnemonic

MARC4 opcode: 02 hex

Stack effect: EXP ($n1\ n2\ —\ n1 + n2 + 1$)
 RET (—)

Stack changes: EXP: top element is popped from the stack
 RET: not affected

Flags: CARRY flag: Set on arithmetic overflow ($n1 + /n2 + 1 > 15$)
 BRANCH flag = CARRY flag

X Y registers: not affected

Bytes used: 1

See also: –C SUBB

Example:

```
: MINUS    5 3 -      ( TOS = 2 ; flags : —      )  
;  
: Underflow 3 5 -    ( TOS = E; BRANCH, CARRY flag set )  
;
```


–?LEAVE

“Not–Query–Leave”

Purpose:

Conditional exit from within a LOOP structure, if the previous tested condition was FALSE (ie. the BRANCH flag is RESET). –?LEAVE is the opposite to ?LEAVE (condition TRUE).
The standard FORTH word sequence NOT IF LEAVE THEN is equivalent to the qFORTH word –?LEAVE.

–?LEAVE transfers control just beyond the next LOOP, +LOOP or #LOOP or any other loop structure like BEGIN ... UNTIL, WHILE

Category: Control structure / qFORTH macro

Library implementation:

```
CODE –?LEAVE TOG_BF ( Toggle BRANCH flag setting )
      (S)BRA _$LOOP      ( Exit LOOP if BRANCH flag set)
END-CODE
```

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag: not affected
BRANCH flag = NOT BRANCH flag

X Y registers: not affected

Bytes used: 3

See also: DO LOOP +LOOP #LOOP ?LEAVE

Example:

```
8  CONSTANT Length
7  CONSTANT LSD
Length ARRAY  BCD_Number          ( 8 digit BCD value  )
```

```
: DIGIT+          \ Add digit to n-digit BCD value
  Y!              ( digit n LSD_Addr — digit n      )
  CLR_BCF         ( Clear BRANCH and CARRY flag     )
  #DO             ( Use length as loop index        )
  [Y]@ +C DAA     ( n — m+n [BRANCH set on overflow] )
  [Y-]! 0        ( m' — 0                          )
  -?LEAVE        ( Finish loop, if NO overflow     )
#LOOP            ( Decrement index & repeat if >0   )
DROP
```

;

```
: Add8
  BCD_Number Length ERASE      ( clear the array )
  8 Length BCD_Number [LSD] Digit+
```

;

-C SUBB

“Minus-C”

Purpose:

Subtract with BORROW [= NO CARRY oder /CARRY] 1's complement of the top two 4-bit values and replace them with the 4-bit result [= n1+/n2+/CARRY] on top of the stack (/n2 is the inverse bit pattern [1's complement] of n2).

Category: (-C) : arithmetic/logical (single-length)
(SUBB) : MARC4 mnemonic

MARC4 opcode: 03 hex

Stack effect: EXP (n1 n2 — n1+/n2+/CARRY)
RET (—)

Stack changes: EXP: top element is popped from the stack
RET: not affect

Flags: CARRY flag: Set on arithmetic underflow
(n1+/n2+/CARRY > 15)
BRANCH flag = CARRY flag

X Y registers: not affected

Bytes used: 1

See also: SUB TCS DAA - +C

Example:

```
: DNEGATE \ Two's complement of top byte    ( d1 — -d1 )  
  0 -    SWAP  
  0 -C   SWAP  
;
```

0 ... Fh (15) LIT_0 ... LIT_F

“Literal”

Purpose :

PUSH the LITeral <n> (0...15) onto the expression stack.

Category:

Stack operation / assembler instruction

MARC4 opcode:

60 ... 6F hex

Stack effect:

EXP (— n) < n = 0 .. Fh >
RET (—)

Stack changes:

EXP: one element is pushed onto the stack.
RET: not affected

Flags:

not affected

X Y registers:

not affected

Bytes used:

1

See also:

CONSTANT 2CONSTANT

Example:

: LITERAL-example

LIT_A	(is equivalent to A hex or 10 decimal)
LIT_0	(is equivalent to 0 decimal)
+	(results in the LIT_A remaining on the TOS)
DROP	(drop the result.)

(better used for above sequence :)

Ah 0 + DROP

21h ABh	(— 2 1 — 2 1 A B)
D+ 2DROP	(2 1 A B — C C —)
12 1 + DROP	(C — C 1 — D —)

;



”Zero-not-equal”

Purpose:

Compares the 4-bit value on top of the stack to zero.

If the value on the stack is not zero, then the BRANCH flag is set in the condition code register (CCR). Unlike standard FORTH, whereby a BOOLEAN value (0 or 1), depending on the comparison result, is pushed onto the stack.

Category: Comparison (single-length) / qFORTH macro

Library implementation: CODE 0<>
0 CMP_NE DROP
END-CODE

Stack effect: EXP (n —)
RET (—)

Stack changes: EXP: top element is popped from the stack
RET: not affected

Flags: CARRY flag: affected
BRANCH flag: set, if (TOS <> 0)

X Y registers: not affected

Bytes used: 3

See also: 0= D0= D0<>

Example:

```
: NOT-EQUALS-ZERO
  5 6      ( — 6 5      )
  0<>      ( 5 6 — 5; BRANCH flag SET )
  DROP 0   ( 5 — 0      )
  0=       ( 0 — ; BRANCH flag SET )
;
```


0=

“Zero–equal”

Purpose:

Compares the 4-bit value on top of the stack to zero.

If the value on the stack is equal to zero, then the BRANCH flag is set in the condition code register. Unlike standard FORTH, whereby a BOOLEAN value (0 or 1), depending on the comparison result, is pushed onto the stack.

Category: Comparison (single-length) / qFORTH macro

Library implementation: CODE 0=
0 CMP_EQ DROP
END-CODE

Stack effect: EXP (n —)
RET (—)

Stack changes: EXP: top element is popped from the stack
RET: not affected

Flags: CARRY flag: affected
BRANCH flag: set, if (TOS = 0)

X Y registers: not affected**Bytes used:** 3**See also:** D0= D0<> 0<>

Example:

```
: ZERO-EQUALS
  5 6          ( — 6 5          )
  0<>         ( 5 6 — 5 ; BRANCH flag SET )
  DROP 0      ( 5 — 0          )
  0=          ( 0 — ; BRANCH flag SET )
;
```

	1+	INC
--	-----------	------------

“One-plus”

Purpose:

Increments the 4-bit value on top of the stack (TOS) by 1.

Category (1+): Arithmetic/logical (single-length)
(INC): MARC4 mnemonic

MARC4 opcode: 14 hex

Stack effect: EXP (n — n+1)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag: not affected
BRANCH flag: set, if (TOS = 0)

X Y registers: not affected

Bytes used: 1

See also: 1- 1+!

Example:

```
VARIABLE PresetValue
VARIABLE Switch

: DownCounter      ( PresetValue — )
  BEGIN 1-        ( n — n-1 )
  UNTIL DROP
;

: UpCounter        ( PresetValue — )
  BEGIN 1+        ( n — n+1 )
  UNTIL DROP
;

: SelectDirection
  9 PresetValue !
  0 Switch !
  BEGIN
  PresetValue @
  Switch 1 TOGGLE ( Toggle between 1 <-> 0 )
  IF DownCounter
  ELSE UpCounter
  THEN
  PresetValue 1-!
  UNTIL
;
```

1+!

“One-plus-store”

Purpose:

Decrements the 4-bit contents of a specified memory location. 1+! requires the address of the variable on top of the stack.

Category:

Memory operation (single-length) / qFORTH macro

Library implementation:

```
CODE 1+!
      Y! [Y]@ 1+ [Y]! ( increment variable by 1 )
END-CODE
```

Stack effect:

```
EXP ( addr — )
RET ( — )
```

Stack changes:

```
EXP: 2 elements are popped from the stack
RET: not affected
```

Flags:

```
CARRY flag: not affected
BRANCH flag: set, if (TOS = 0)
```

X Y registers:

The address is stored into the Y register, then the value is fetched from RAM, the value is incremented on the stack and restored in the address indicated by the Y (or X) register.

Bytes used:

4

See also:

+! 1-!

Example:

VARIABLE State

```
: StateCounter
  5 State !           ( store 5 in the memory location      )
  6 0 DO             ( set loop index = 6                )
    State 1+!       ( increment contents of variable 'State' )
  LOOP
;
```

	1-	DEC
--	-----------	------------

“One-minus”

Purpose:

Decrements the 4-bit value on top of the stack (TOS) by 1.

Category (1-): Arithmetic/logical (single-length)
(DEC): MARC4 mnemonic

MARC4 opcode: 15 hex

Stack effect: EXP (n — n-1)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag not affected
BRANCH flag Set, if (TOS = 0)

X Y registers: not affected

Bytes used: 1

See also: 1+ 1-!

Example:

```
VARIABLE PresetValue
VARIABLE Switch

: DownCounter      ( PresetValue — )
  BEGIN 1-        ( n — n-1 )
  UNTIL DROP
;

: UpCounter        ( PresetValue — )
  BEGIN 1+        ( n — n+1 )
  UNTIL DROP
;

: SelectDirection
  9 PresetValue !
  0 Switch !
  BEGIN
  PresetValue @
  Switch 1 TOGGLE ( Toggle between 1 <-> 0 )
  IF DownCounter
ELSE UpCounter
THEN
  PresetValue 1-! ( UNTIL PresetValue = 0 )
  UNTIL
;
```


1-!

“One–minus–store”

Purpose:

Decrements the 4-bit contents of a specified memory location. 1-! requires the address of the variable on top of the stack.

Category:

Memory operation (single-length) / qFORTH macro

Library implementation:

```
CODE 1-!
      Y! [Y]@ 1- [Y]!    ( decrement variable by 1 )
END-CODE
```

Stack effect:

EXP (addr —)
RET (—)

Stack changes:

EXP: 2 elements are popped from the stack
RET: not affected

Flags:

CARRY flag: not affected
BRANCH flag: set, if (TOS = 0)

X Y registers:

The address is stored into the Y (or X) register, then the value is fetched from RAM , the value is decremented on the stack and re-stored in the address indicated by the Y register.

Bytes used:

4

See also:

+! 1+! !

Example:

VARIABLE PresetValue

VARIABLE Switch

```
: DownCounter          ( PresetValue —      )  
  BEGIN 1- UNTIL DROP ; ( n — n-1      )
```

```
: UpCounter           ( PresetValue —      )  
  BEGIN 1+ UNTIL DROP ; ( n — n+1      )
```

```
: SelectDirection  
  9 PresetValue !  
  0 Switch !  
  BEGIN  
    PresetValue @  
    Switch 1 TOGGLE ( Toggle between 1 <-> 0 )  
    IF DownCounter ELSE UpCounter THEN  
    PresetValue 1-! ( UNTIL PresetValue = 0 )  
  UNTIL  
;
```

2!

“Two-store”

Purpose:

Stores the 8-bit value on TOS into an 8-bit variable in RAM. The address of the variable is the TOS value.

Category: Arithmetic/logical (double-length) / qFORTH macro

Library implementation: CODE 2! Y! (nh nl addr — nh nl)
 SWAP (nh nl — nl nh)
 [Y]! (nl nh — nl)
 [+Y]! (nl —)
 END-CODE

Stack effect: EXP (n_h n_l RAM_addr —)
 RET (—)

Stack changes: EXP: 4 elements are popped from the stack
 RET: not affected

Flags: not affected

X Y registers: The contents of the Y or X register may be changed.

Bytes used: 4

See also: 2@ D+! D-!

Example 1:

```
: D-STORE
  13h 43h 2!      ( RAM [43] = 1 ; RAM [44] = 3      )
;                ( but normally variable names are used ! )
```

Example 2:

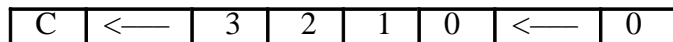
```
2VARIABLE Counter
: DoubleCount
  0 0 Counter 2!  ( initialize the 8-bit counter      )
  BEGIN
    0 1 Counter D+! ( increment the 8-bit counter      )
    Counter 2@ 20h D= ( until 20h is reached ...      )
  UNTIL
; 
```

	2*	SHL
--	----	-----

“Two–multiply”

Purpose:

Multiplies the 4-bit value on top of the stack by 2. SHL shifts TOS left into CARRY flag.



$$|C| \leftarrow |3|2|1|0| \leftarrow 0$$

Category:

(2*) : Arithmetic/logical (single-length) / qFORTH macro

(SHL) : MARC4 mnemonic / assembler instruction

MARC4 opcode:

10 hex (SHL)

Library implementation:

CODE 2* SHL END-CODE

Stack effect:

EXP (n — n*2)

RET (—)

Stack changes:

EXP: not affected

RET: not affected

Flags:

CARRY flag: the MSB of n is shifted into CARRY

BRANCH flag = CARRY flag

X Y registers:

not affected

Bytes used:

1

See also:

ROL ROR SHR 2/ D2* D2/

Example 1:

```
: MULT-BY-TWO      ( multiply a 4-bit number:      )
  7 2*             ( 7h — Eh                      )
                  ( multiply a 8-bit number: 'D2*' Macro : )
  18h SHL SWAP     ( 18h — 0 1h [CARRY flag set]    )
  ROL SWAP         ( 0 1h [CARRY] — 30h            )
;                  ( 18h * 2 = 30h ! use 'D2*' !      )
```

Example 2:

```
: BitShift
  SET_BCF 3        ( 3 = 0011b                      )
  ROR DROP        ( [CARRY] 3 — [CARRY] 9 = 1001b    )

  CLR_BCF 3        ( 3 = 0011b                      )
  ROR DROP        ( [no CARRY] 3 — [CARRY] 1 = 0001b  )

  SET_BCF 3        ( 3 = 0011b                      )
  ROL DROP        ( [CARRY] 3 — [no CARRY] 7 = 0111b  )

  CLR_BCF 3        ( 3 = 0011b                      )
  ROL DROP        ( [no CARRY] 3 — [no CARRY] 6 = 0110b )
  ( _____ )

  CLR_BCF Fh      ( —SHR—[no CARRY] F — [CARRY] 7 = 0111b )
  2/ DROP        ( 6 = 0110b                          )
  CLR_BCF 6       ( 6 = 0110b                          )
  2* DROP        ( —SHL— [no CARRY] 6 — [no C] C = 1100b )
; 
```

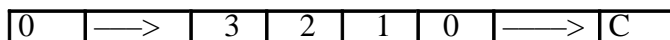
2/

SHR

“Two-divide”

Purpose:

Divides the 4-bit value on top of the stack by 2. SHR shifts TOS right into CARRY flag.



Category:

(2/) : arithmetic/logical (single-length) / qFORTH macro
 (SHR) : MARC4 mnemonic / assembler instruction

MARC4 opcode:

12 hex (SHR)

Library implementation:

CODE 2/ SHR END-CODE

Stack effect:

EXP (n — n/2)
 RET (—)

Stack changes:

EXP: not affected
 RET: not affected

Flags:

CARRY flag: the LSB of n is shifted into CARRY
 BRANCH flag = CARRY flag

X Y registers:

not affected

Bytes used:

1

See also:

2* SHL D2* D2/ ROR ROL

Example:

```
: TWO-DIVIDE
  10 2/          ( 10 — 5 )
                ( divide an 8-bit number: 'D2/' Macro : )
  30h SWAP SHR   ( 30h — 0 1h [CARRY flag set] )
    SWAP ROR     ( 0 1 [CARRY] — 18h )
;                ( 30h / 2 = 18h ! use 'D2/' ! )
```

For another example see '2*', 'SHL'.

2<ROT

“Two-left-rote”

Purpose:

Moves the top 8-bit value to the third byte position on the EXP stack, ie. performs an 8-bit left rotate.

Category: Stack operation (double-length) / qFORTH colon definition

Library implementation: : 2<ROT
 ROT >R (d1 d2 d3 — d1 d2h d3)
 ROT >R (d1 d2h d3 — d1 d3)
 ROT >R (d1 d3 — d1h d3)
 ROT R> (d1h d3 — d3 d1)
 R> R> (d3 d1 — d3 d1 d2)
 ;

Stack effect: EXP (d1 d2 d3 — d3 d1 d2)
 RET (—)

Stack changes: EXP: affected (changed order of elements)
 RET: use of 3 RET levels in between

Flags: not affected

X Y registers: not affected

Bytes used: 14

See also: ROT <ROT 2ROT

Example:

```
: TWO-LEFT-ROT
11h 22h 33h      ( — 11h 22h 33h      )
  2<ROT          ( 11h 22h 33h — 33h 11h 22h )
  ROT            ( 33h 11h 22h — 33h 12h 21h )
;
```

2>R

“Two-to-R”

Purpose:

Moves the top two 4-bit values from the expression stack and pushes them onto the top of the return stack.

2>R pops the EXP stack onto the RET stack. To avoid corrupting the RET stack and crashing the system, each use of 2>R MUST be followed by a subsequent 2R> (or an equivalent 2R@ and DROPR) within the same colon definition.

Category: Stack operation (double-length) / assembler instruction

MARC4 opcode: 28 hex

Stack effect: EXP (n1 n2 —)
RET (— u|n2|n1)

Stack changes: EXP: 2 elements are popped from the stack
RET: 1 (8-bit) entry is pushed onto the stack

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: I>R R> 2R@ – 2R> 3R@ 3>R 3R>

Example:

```
: 2SWAP          ( Swap 2nd byte with top          )
  >R <ROT        ( d1 d2 — n2_h d1                )
  R> <ROT        ( n2_h d1 — d2 d1                )
;

: 2OVER          ( Duplicate 2nd byte onto top      )
  2>R            ( d1 d2 — d1                      )
  2DUP           ( d1 — d1 d1                      )
  2R>            ( d1 d1 — d1 d1 d2                )
  2SWAP          ( d1 d1 d2 — d1 d2 d1            )
;
```

2@

“Two-fetch”

Purpose:

Copies the 8-bit value of a 2VARIABLE or 2ARRAY element to the stack. The MSN address of the selected variable is the TOS value.

Category: Arithmetic/logical (double-length) / qFORTH macro

Library implementation: CODE 2@ Y! (addr —)
 [Y]@ (— nh)
 [+Y]@ (nh — nh nl)
 END-CODE

Stack effect: EXP (RAM_addr — n_h n_l)
 RET (—)

Stack changes: EXP: The top two elements will be changed.
 RET: not affected

Flags: not affected

X Y registers: The contents of the Y or X register may be changed.

Bytes used: 3

See also: Other byte (double-length) qFORTH dictionary words,
 like
 D- D+ 2! D+! D-! D2/ D2* D< D> D<> D= D<= D>=
 D0= D0<>

Example 1:

```
: StoreFetch
  13h 43h 2!          ( RAM [43] = 1 ; RAM [44] = 3      )
                    ( use var name instead of address )
  43h 2@             ( — 1 3                          )
  2DROP
;
```

Example 2:

2VARIABLE Counter

```
: DoubleCount
  0 0 Counter 2!      ( initialize the 8-bit counter      )
  BEGIN
  0 1 Counter D+!    ( increment the 8-bit counter      )
  Counter 2@ 20h D=  ( until 20h is reached ...      )
  UNTIL
;
```

2ARRAY

“Two-ARRAY”

Purpose:

Allocates RAM memory for storage of a short double-length (8-bit / byte) array, using a 4-bit array index value. Therefore the number of 8-bit array elements is limited to 16.

The qFORTH syntax is as follows:

```
<number> 2ARRAY <identifier> [ AT <RAM-Addr> ]
```

At compile time, 2ARRAY adds <identifier> to the dictionary and ALLOTs memory for storage of <number> double-length values. At execution time, <identifier> leaves the RAM start address of the parameter field (<identifier> [0]) on the expression stack.

The storage area allocated by 2ARRAY is not initialized.

Category: Predefined data structure

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 0

See also: ARRAY LARRAY 2LARRAY 2VARIABLE VARIABLE

Example:

```
6 2ARRAY RawData ( RawData[0] ... RawData[5] )
3 CONSTANT STEP
```

```
: Init_RawData
  15h 6 0 DO ( RawData[0] := 15h )
    2DUP ( RawData[1] := 12h )
      I RawData INDEX 2! ( RawData[2] := 0Fh )
      STEP M- ( RawData[3] := 0Ch )
    LOOP ( RawData[4] := 09h )
  2DROP ( RawData[5] := 06h )
```

;

```
: INTO
  Init_RawData
  RawData [3] 2@
  50h D+
```

;

2CONSTANT

“Two-CONSTANT”

Purpose:

Creates a double-length (8-bit) constant definition. The qFORTH syntax is as follows:

```
<byte-constant> 2CONSTANT <identifier>
```

which assigns the 8-bit value <byte-constant> to <identifier>.

Category:	Predefined data structure
Stack effect:	EXP (— d) at execution time RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	0
See also:	ARRAY, CONSTANT

Example:

```
00h 2CONSTANT ZEROB ( define byte constants )
01h 2CONSTANT ONEB

: Emit_HexByte      \ Emit routine EXP: ( n1 n2 — )
  2DUP 3 OUT      ( Duplicate & emit lower nibble )
  SWAP 2 OUT
  SWAP
;

: FIBONACCI        ( Display 12 FIBONACCI nrs. )
  ZEROB ONEB      ( Set up for calculations )
  12 0 DO         ( Set up loop — 12 numbers )
    Emit_HexByte  ( Emit top 8-bit number )
    2SWAP 2OVER  ( Switch for addition )
    D+           ( Add top two 8-bit numbers )
  LOOP           ( Go back for next number )
  Emit_HexByte   ( Emit last number too )
  2DROP 2DROP   ( Clean up the stack )
;
```

2DROP

“Two-DROP”

Purpose:

Removes one double-length (8-bit) value or two single-length (4-bit) values from the expression stack.

Category: Stack operation (double-length) / qFORTH macro

Library implementation: CODE 2DROP
DROP DROP
END-CODE

Stack effect: EXP (n1 n2 —)
RET (—)

Stack changes: EXP: 2 elements are popped from the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 2

See also: DROP 3DROP

Example:

```
: TWO-DROP          ( Simple example for 2DROP      )
  19H 11H           (   — 19h 11h                  )
  2DROP            ( 19h 11h — 19h                  )
;
```

2DUP

“Two-doop”

Purpose:

Duplicates the double-length (8-bit) value on top of the expression stack.

Category:

Stack operation (double-length) / qFORTH macro

Library implementation:

```
CODE 2DUP
      OVER OVER
END-CODE
```

Stack effect:

```
EXP ( d — d d )
RET ( — )
```

Stack changes:

```
EXP: top 2 elements are pushed again onto the stack
RET: not affected
```

Flags:

not affected

X Y registers:

not affected

Bytes used:

2

See also:

DUP 3DUP

Example:

```
2VARIABLE CounterValue ( 8-bit counter value )

: Update_Byte ( addr — current value )
  2DUP ( addr — addr addr )
  2@ 2SWAP ( addr addr — d addr )
  18 2SWAP ( d addr — d 18 addr )
  D+! ( d 18 addr — d )
;

: Task_5
  CounterValue Update_Byte ( — d )
  3 OUT 2 OUT
;
```

2LARRAY

“Two-long-ARRAY”

Purpose:

Allocates RAM space for storage of a double-length (8-bit) long array, which has an 8-bit index to access the 8-bit array elements (more than 16 elements).

The qFORTH syntax is as follows

<number> 2LARRAY <identifier> [AT <RAM address>]

At compile time, 2LARRAY adds <identifier> to the dictionary and ALLOTs memory for storage of <number> double-length values. At execution time, <identifier> leaves the RAM start address of the array (<identifier> [0]) on the expression stack. The storage area ALLOTed by 2LARRAY is not initialized.

Category:	Predefined data structure
Stack effect:	EXP (—) RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	0
See also:	ARRAY 2ARRAY LARRAY

Example:

```
25 2LARRAY VarText
20 2CONSTANT StrLength
ROMCONST FixedText StrLength, ' Long string example ' ,
: TD- D- IF ROT 1- <ROT THEN ; ( subtract 8 from 12-bit. )
: TD+ D+ IF ROT 1+ <ROT THEN ; ( add 8-bit to a 12-bit value. )
: ROM_Byte@
  3>R 3R@ TABLE ;; ( keep ROMaddr on EXP; fetch char. )
( ***** )
: CopyString ( copy string from ROM into RAM array )
  FixedText ROM_Byte@ ( get string length. )
  2>R ( move length/index to return stack )
  2R@ TD+ ( length + start addr=ROMaddr [lastchar] )
  BEGIN ( )
    ROM_Byte@ ( get ASCII char )
    2R> 1 M- 2>R ( index := index - 1 )
    2R@ VarText INDEX 2! ( store char in array. )
    0 1 TD- ( ROMaddr := ROMaddr - 1 )
    2R@ D0= ( index = 0 ? )
  UNTIL ( )
  DROPR 3DROP ( skip count & ROMaddr. )
;
```


2NIP

“Two-NIP”

Purpose:

Removes the second double-length (8-bit) value from the expression stack.

Category: Stack operation (double-length) / qFORTH macro

Library implementation: CODE 2NIP
ROT DROP
ROT DROP
END-CODE

Stack effect: EXP (d1 d2 — d2)
RET (—)

Stack changes: EXP: 2 elements are popped from the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 4

See also: NIP SWAP DROP

Example:

```
: Cancel-2nd-Byte
  9 25 5Ah          ( — 9 19h 5Ah      )
  2NIP             ( 9 19h 5Ah — 9 5Ah  )
;
```

2OVER

“Two-OVER”

Purpose:

Copies the second double length (8-bit) value onto the top of the expression stack.

Category: Stack operation (double-length) / qFORTH colon definition

Library implementation: : 2OVER
 2>R (d1 d2 — d1)
 2DUP (d1 — d1 d1)
 2R> (d1 d1 — d1 d1 d2)
 2SWAP (d1 d1 d2 — d1 d2 d1)
 ;

Stack effect: EXP (d1 d2 — d1 d2 d1)
 RET (—)

Stack changes: EXP: 2 elements are pushed onto the stack
 RET: 3 levels are used in between

Flags: not affected

X Y registers: not affected

Bytes used: 8

See also: 2DUP 2SWAP OVER

Example:

```
: Double-2nd-Byte
  12H 19H      ( — 12h 19h      )
  2OVER       ( 12h 19h — 12h 19h 12h )
;
```

2R>

“Two-R-from”

Purpose:

Moves the double length (8-bit) value from the return stack onto the expression stack.
 2R@ , 2R> and 2>R allow use of the return stack as a temporary storage for 8-bit values.

2R> removes elements from the return stack onto the expression stack. To avoid corrupting the return stack and crashing the program, each use of 2R> MUST be preceded by a 2>R within the same colon definition.

Category: Stack operation (double-length) / qFORTH macro

Library implementation: CODE 2R>
 2R@ DROPR
 END-CODE

Stack effect: EXP (— n1 n2)
 RET (u|n2|n1 —)

Stack changes: EXP: 2 elements are pushed onto the stack
 RET: 1 (8-bit) entry is popped from the stack

Flags: not affected

X Y registers: not affected

Bytes used: 2

See also: I>R R> 2R@ 2>R – 3R@ 3>R 3R>

Example 1:

Library implementation: of +LOOP:

```
CODE +LOOP
  2R>          ( Move limit & index on EXP      )
  ROT + OVER
  CMP_LT      ( Check for index < limit        )
  2>R
  (S)BRA _$DO
_ $LOOP: DROPR ( Skip limit & index from RET      )
END-CODE
```

Example 2:

```
: 2OVER      ( Duplicate 2nd byte onto top      )
  2>R        ( d1 d2 — d1                      )
  2DUP       ( d1 — d1 d1                      )
  2R>        ( d1 d1 — d1 d1 d2                )
  2SWAP      ( d1 d1 d2 — d1 d2 d1            )
;
```

2R@

“Two-R-fetch”

Purpose:

Takes a copy of the 8-bit value on top of the return stack and pushes the double length (8-bit) value on the expression stack.

2R@ , 2R> and 2>R allow use of the return stack as a temporary storage for 8-bit values.

Category: Stack operation (double-length) / assembler instruction

MARC4 opcode: 2A hex

Stack effect: EXP (— n1 n2)
RET (u|n1|n2 — u|n1|n2)

Stack changes: EXP: 2 elements are pushed onto the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: I >R R> 2>R 2R> 3R@ 3>R 3R>

Example:

```
: 2OVER          ( Duplicate 2nd byte onto top      )
  2>R            ( d1 d2 — d1                      )
  2DUP          ( d1 — d1 d1                      )
  2R@ DROPR     ( d1 d1 — d1 d1 d2                )
  2SWAP        ( d1 d1 d2 — d1 d2 d1              )
;
```


2ROT

“Two-rote”

Purpose:

Moves the third double length (8-bit) value onto the top of the expression stack.

Category: Stack operation (double-length) / qFORTH colon definition

Library implementation: : 2ROT
 2>R 2SWAP (d1 d2 d3 — d2 d1)
 2R> 2SWAP (d2 d1 — d2 d3 d1)
 ;

Stack effect: EXP (d1 d2 d3 — d2 d3 d1)
 RET (—)

Stack changes: EXP: affected (changes order of elements)
 RET: affected (x levels are used in between)

Flags: not affected

X Y registers: not affected

Bytes used: 8

See also: 2<ROT <ROT ROT

Example:

```
: Rotate-Byte-to-Top
  11h 22h 33h      (   — 11h 22h 33h      )
  2ROT             ( 11h 22h 33h — 22h 33h 11h )
;
```

2SWAP

“Two-SWAP”

Purpose:

Exchanges the top two double-length (8-bit) values on the expression stack.

Category: Stack operation (double-length) / qFORTH colon definition

Library implementation: : 2SWAP
 >R <ROT(d1 d2 — d2h d1)
 R> <ROT(d2h d1 — d2 d1)
 ;

Stack effect: EXP (d1 d2 — d2 d1)
 RET (—)

Stack changes: EXP: affected (changes order of elements)
 RET: affected (1 level is used intermediately)

Flags: not affected

X Y registers: not affected

Bytes used: 8

See also: SWAP

Example:

```
: Swap-Bytes
  3 12h 19h      ( — 3 12h 19h      )
  2SWAP         ( 3 12h 19h — 3 19h 12h )
;
```

2TUCK

“Two-TUCK”

Purpose:

Tucks the top 8-bit (double-length) value under the second byte on the expression stack, the counterpart to 2OVER.

Category: Stack operation (double-length) / qFORTH colon definition

Library implementation: : 2TUCK

```

3>R      ( n1 n2 n3 n4 — n1      )
2R@     ( n1 — n1 n3 n4      )
ROT      ( n1 n3 n4 — n3 n4 n1  )
3R>     ( n3 n4 n1 — n3 n4 n1 n2 n3 n4 )
;        ( n3 n4 n1 n2 n3 n4 — d2 d1 d2 )
    
```

Stack effect: EXP (d1 d2 — d2 d1 d2)
RET (—)

Stack changes: EXP: Two 4-bit elements are pushed under the 2nd byte
RET: affected (1 level is used intermediately)

Flags: not affected

X Y registers: not affected

Bytes used: 6

See also: TUCK 2OVER

Example:

```
: Tuck-under-2nd-Byte
  11H 22H          ( — 11h 22h          )
  2TUCK           ( 11h 22h — 22h 11h 22h )
;
```

2VARIABLE

“Two-VARIABLE”

Purpose:

Allocates RAM space for storage of one double-length (8-bit) value.

The qFORTH syntax is as follows

```
2VARIABLE <name> [ AT <addr.> ] [ <number> ALLOT ]
```

At compile time, 2VARIABLE adds <name> to the dictionary and ALLOTs memory for storage of one double-length value. If AT <addr.> is appended, the variable/s will be placed at a specific address (i.e.: 'AT 40h'). If <number> ALLOT is appended, a total of $2 * (\text{<number>} + 1)$ 4-bit memory locations will be allocated. At execution time, <name> leaves the start address of the parameter field on the expression stack.

The storage ALLOTed by 2VARIABLE is not initialized.

Category:	Predefined data structure
Stack effect:	EXP (— d) at execution time RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	0
See also:	ALLOT VARIABLE 2ARRAY ARRAY

Example:

```
2VARIABLE NoKeyCounter          ( 8-bit variable          )
: No_KeyPressed
  0 0 NoKeyCounter 2!           ( initialise to $00           )
  NoKeyCounter 2@ 1 M+         ( increment by 1             )
  IF NoKeyOver THEN           ( 'call' NoKeyOver on overflow )
  NoKeyCounter 2!             ( store the result back       )
;
```


3!

“Three-store”

Purpose:

Store the top 3 nibbles into a 12-bit variable in RAM. The most significant digit address of that array has to be the TOS value.

Category: Memory operation (triple-length) / qFORTH colon definition

Library implementation: : 3! Y! (nh nm nl addr — nh nm nl)
 SWAP ROT (nh nm nl — nl nm nh)
 [Y]! [Y+]! (nl nm nh — nl)
 [+Y]! ; (nl —)

Stack effect: EXP (nh nm nl addr —)
 RET (—)

Stack changes: EXP: 5 element are popped from the stack
 RET: not affected

Flags: not affected

X Y registers: The contents of the Y register will be changed.

Bytes used: 7

See also: 3@ T+! T-! TD+! TD-!

Example:

```
3 ARRAY 3Nibbles AT 40h ( 3 nibbles at fixed locations. )

: Triples
  123h 3Nibbles 3! ( store 123h in the 3 nibbles array. )
  321h 3Nibbles T+! ( 123h + 321h = 444h )
  3Nibbles 3@ 3DROP ( fetch the result onto expression stack . )
  123h 3Nibbles T-! ( 444h - 123h = 321h )
;
```

3>R

“Three-to-R”

Purpose:

Removes the top 3 values from the expression stack and places them onto the return stack. 3>R unloads the EXP stack onto the RET stack. To avoid corrupting the RET stack and crashing the system, each use of 3>R MUST be followed by a subsequent 3R> within the same colon definition.

Category: Stack operation (triple-length) / assembler instruction

MARC4 opcode: 29 hex

Stack effect:

EXP	(n1	n2	n3	—)
RET	(—	n3 n2 n1)		

Stack changes:

EXP:	3 elements are popped from the top of the stack
RET:	1 entry (3 elements) is pushed onto the stack

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: I >R R> 2R@ 2>R 2R> 3R@ 3R>

Example:

```
: 2TUCK          \ TUCK top 8-bit value under the 2nd byte
  3>R            ( n1 n2 n3 n4 — n1                )
  2R@            ( n1 — n1 n3 n4                    )
  ROT            ( n1 n3 n4 — n3 n4 n1              )
  3R>            ( n3 n4 n1 — n3 n4 n1 n2 n3 n4    )
  ;              ( d1 d2 — d2 d1 d2                )
```

3@

“Three-fetch”

Purpose:

Fetch an 12-bit variable in RAM and push it on the expression stack. The most significant digit address of the variable must be on TOS.

Category: Memory operation (triple-length) / qFORTH macro

Library implementation: CODE 3@ Y! (addr —)
[Y]@ (— nh)
[+Y]@ (nh — nh nm)
[+Y]@ (nh nm — nh nm nl)
END-CODE

Stack effect: EXP (addr — nh nm nl)
RET (—)

Stack changes: EXP: 2 elements are popped from and 3 are pushed onto the expression stack.
RET: not affected

Flags: not affected

X Y registers: The contents of the Y or X register may be changed.

Bytes used: 5

See also: 3! T+! T-!

Example:

```
3 ARRAY 3Nibbles AT 40h ( 3 nibbles at fixed locations in RAM )
( ***** )
: Triples
  123h 3Nibbles 3!      ( store 123h in the 3 nibbles array. )
  321h 3Nibbles T+!    ( 123h + 321h = 444h )
  3Nibbles 3@ 3DROP    ( fetch the result onto expression stack. )
  123h 3Nibbles T-!    ( 444h - 123h = 321h )
  3Nibbles 3@ 3DROP    ( fetch the result onto expression stack. )
;
```

3DROP

“Three-DROP”

Purpose:

Removes one 12-bit or three 4-bit values from the expression stack.

Category:

Stack operation (triple-length) / qFORTH macro

Library implementation:

```
CODE 3DROP
DROP DROP DROP
END-CODE
```

Stack effect:

```
EXP ( n1 n2 n3 — )
RET ( — )
```

Stack changes:

```
EXP: 3 elements are popped from the stack
RET: not affected
```

Flags:

not affected

X Y registers:

not affected

Bytes used:

3

See also:

DROP 2DROP DROPR

Example:

```
: Skip-top-3-nibbles
  3 4 6 7          ( — 3 4 6 7 )
  3DROP          ( 3 4 6 7 — 3 )
;
```


3DUP

“Three-doop”

Purpose:

Duplicates the 12-bit address value on top of the expression stack.

Category: Stack operation (triple-length) / qFORTH colon definition

Library implementation: CODE 3DUP

3>R 3R@ 3R> (t — t t)

END-CODE

Stack effect: EXP (n1 n2 n3 — n1 n2 n3 n1 n2 n3)
RET (—)

Stack changes: EXP: 3 elements are pushed onto the stack
RET: affected (1 level is used intermediately)

Flags: not affected

X Y registers: not affected

Bytes used: 4

See also: DUP 2DUP

Example:

ROMCONST Message 5, ' Error ',

: Duplicate-ROMaddr

Message 3DUP (duplicate ROM address on stack)

ROMByte@ (fetch string length)

NIP (get string length as 4-bit value)

;

3R>

“Three-R-from”

Purpose:

Moves the top 3 nibbles from the return stack and puts them onto the expression stack. 3R> unloads the return stack onto the expression stack. To avoid corrupting the return stack and crashing the system, each use of 3R> MUST be preceded by a 3>R within the same colon definition.

Category: Stack operation (triple-length) / qFORTH macro

Library implementation: CODE 3R>
 3R@
 DROPR
 END-CODE

Stack effect: EXP (— n1 n2 n3)
 RET (n3|n2|n1 —)

Stack changes: EXP: 3 elements are pushed onto the stack
 RET: 1 element (3 nibbles) is popped from the stack

Flags: not affected

X Y registers: not affected

Bytes used: 2

See also: I >R R> 2R@ 2>R 2R> 3R@ 3>R

Example:

```
CODE 3DUP          ( Library implementation: of 3DUP    )
  3>R              ( t —                               )
  3R@              ( — t                               )
  3R>              ( t — t t                           )
END-CODE
```

```
ROMCONST StringExample 6, 'String',
```

```
: Duplicate-ROMAddr
  StringExample 3DUP ( duplicate ROM address on stack  )
  0 DTABLE@         ( fetch 1st value of ROM string   )
  NIP               ( get string length as 4-bit value )
;
```

3R@

“Three-R-fetch”

Purpose:

Copies the top 3 values from the return stack and leaves the 3 values on the expression stack.

3R@ fetches the topmost value on the return stack. 3R@, 3R> and 3>R allow use of the return stack as a temporary storage for values within a colon definition.

Category: Stack operation (triple-length) / assembler instruction

MARC4 opcode: 2B hex

Stack effect: EXP (— n1 n2 n3)
RET (n3|n2|n1 — n3|n2|n1)

Stack changes: EXP: 3 elements are pushed onto the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: I >R R> 2R@ 2>R 2R> – 3>R 3R>

Example 1:

```
CODE 3DUP          ( Library implementation: of 3DUP  )
  3>R              ( t —                               )
  3R@              ( — t                               )
  3R>              ( t — t t                           )
END-CODE
```

Example 2:

```
:ROM_Byte@        ( ROM_addr — ROM_addr ROM_byte )

3>R               ( ROM_addr —                       )
3R@               ( — ROM_addr                       )
TABLE             ( ROM_addr — ROM_addr ROM_byte )
;;               ( back to 'CALL', implizit EXIT      )
```



“Colon”

Purpose:

Begins compilation of a new colon definition, i.e. defines the entry point of a new subroutine. Used in the form

: <name> ... <words> ... ;

’:’ creates a new dictionary entry for <name> and compiles the sequence between <name> and ’:’ into this new definition. If no errors are encountered during compilation, the new colon definition may itself be used in subsequent colon definitions.

On execution of a colon definition, the current program counter is pushed onto the return stack.

Category:	Predefined data structure	
Stack effect:	EXP	(—)
	RET	(— ReturnAddress)
Stack changes:	EXP:	not affected
	RET:	The return address to the word which executes this colon definition is pushed onto the stack.
Flags:	not affected	
X Y registers:	not affected	
Bytes used:	0	
See also:	; INT0 .. INT7	

Example:

```
: COLON-Example      ( BEGIN a colon definition      )
  3 BEGIN            ( 3 = initial start value      )
    1+ DUP 9 =      ( increment count 3 -> 9        )
  UNTIL             ( continue until condition is true )
;                   ( END of DEFINITION with a SEMICOLON )
```


; ;; **EXIT RTI**

“Semicolon”

Purpose:

Terminates a qFORTH colon definition, i.e. exits from the current colon definition.

';' compiles to EXIT at the end of a normal colon definition. It then marks the new definition as having been successfully compiled so that it can be found in the dictionary.

';;' compiles to RTI which automatically sets the I_ENABLE flag at the end of an INT0 .. INT7 or \$RESET definition.

When EXIT is executed, program control passes out of the current definition. EXIT may NOT be used inside any iterative DO loop structure, but it may be used in control structures, like:

```
BEGIN ... WHILE, REPEAT, ... UNTIL, ... AGAIN, and
IF ... [ ELSE ... ] THEN
```

Note: ';;' suppresses the implementation of an EXIT or RTI in the code generation. This is used after any TABLE instruction and \$AUTOSLEEP, which exits automatically.

Category:	(;)	: Predefined data structure (RTI/EXIT): MARC4 mnemonic
MARC4 opcode:	EXIT : 25 hex	RTI : 1D hex
Stack effect:	EXP RET	(—) (Return address —)
Stack changes:	EXP: RET:	not affected The address on top of the stack is moved into the PC.
Flags:	I_ENABLE flag Set after execution of the RTI instruction at the end of an INTx (x=0..7) or \$RESET definition. CARRY and BRANCH flags are not affected	
X Y registers:	not affected	
Bytes used:	1	
See also:	: ?LEAVE -?LEAVE	

Example:

```
: Colon-Def
  3 BEGIN          ( 3 is the initial start value          )
    1+ DUP 9 =    ( increment count from 3 -> 9          )
    UNTIL        ( Repeat UNTIL condition is TRUE        )
;                ( EXIT from the colon def. with ';'      )

: INT5           ( Register contents saved automatically. )
  DI             ( disable Interrupts                    )
  Colon-Def     ( execute 'colon def'                    )
;               ( RTI - enables all interrupts            )
```

< **CMP_LT**

“Less-than”

Purpose:

'Less-than' comparison of the top two 4-bit values on the stack. If the second value on the stack is less than the top of stack value, then the BRANCH flag in the CCR is set. Unlike standard FORTH, whereby a BOOLEAN value (0 or 1), depending on the comparison result, is pushed onto the stack.

Category: (<) : Comparison (single-length) / qFORTH macro
 (CMP_LT) : MARC4 mnemonic

Library implementation:

```
CODE < CMP_LT        ( n1 n2 — n1 [BRANCH flag])
                      DROP        ( n1 — )
END-CODE
```

MARC4 opcode: 08 hex (CMP_LT)

Stack effect: < EXP (n1 n2 —)
 CMP_LT EXP (n1 n2 — n1)
 both RET (—)

Stack changes: EXP: < 2 elements are popped from the stack
 CMP_LT top element is popped from the stack
 RET: not affected

Flags: CARRY flag affected
 BRANCH flag Set, if (n1 < n2)

X Y registers: not affected

Bytes used: 1 – 2

See also: <> = <= >= > <> D<> D>= D<=

Example:

```
: LESS-THAN      ( Check 5 < 7 and 8 < 6 and   5 < 5      )
  5 7 CMP_LT     ( 5 7 — 5 [ BRANCH and CARRY set ]    )
  8 6 <          ( 5 8 6 — 5 [ BRANCH is NOT set ]     )
  5 CMP_LT       ( 5 5 — 5                             )
  DROP
;
```

<=
CMP_LE

“Less-than-equal”

Purpose:

'Less-than-or-equal' comparison of the top two 4-bit values on the stack. If the 2nd value on the stack is less than, or equal to the top of stack value, then the BRANCH flag in the condition code register (CCR) is set. Unlike standard FORTH, whereby a BOOLEAN value (0 or 1), depending on the comparison result, is pushed onto the stack.

Category: (<=) : Comparison (single-length) / qFORTH macro
 (CMP_LE) : MARC4 mnemonic

Library implementation: CODE <= CMP_LE (n1 n2 — n1 [BRANCH flag])
 DROP (n1 —)
 END-CODE

MARC4 opcode: 09 hex (CMP_LE)

Stack effect: <= EXP (n1 n2 —)
 CMP_LE EXP (n1 n2 — n1)
 both RET (—)

Stack changes: EXP: <= 2 elements are popped from the stack
 CMP_LE top element is popped from the stack
 RET: not affected

Flags: CARRY flag affected
 BRANCH flag set, if (n1 <= n2)

X Y registers: not affected

Bytes used: 1 – 2

See also: D<=

Example:

```
: LESS-EQUALS      ( show 7 <= 5 and 7 <= 7 and 8 <= 9      )
  7 5 CMP_LE      ( 7 5 — 7 [ BRANCH flag NOT set ]      )
  7 <=          ( 7 7 — [ BRANCH flag set ]      )
  8 9 <=        ( 8 9 — [ BRANCH and CARRY flag set ] )
;
```

⟨⟩ **CMP_NE**

“Not-equal”

Purpose:

Inequality test for the top two 4-bit values on the stack. If the 2nd value on the stack is NOT equal to the top of stack value, then the BRANCH flag in the CCR is set. Unlike standard FORTH, whereby a BOOLEAN value (0 or 1), depending on the comparison result, is pushed onto the stack.

Category:

⟨⟩ : Comparison (single-length) / qFORTH macro
 (CMP_NE) : MARC4 mnemonic

Library implementation:

```
CODE  ⟨⟩  CMP_NE    ( n1 n2 — n1 [BRANCH flag])
                DROP  ( n1 — )
END-CODE
```

MARC4 opcode:

07 hex (CMP_NE)

Stack effect:

```
⟨⟩          EXP    ( n1 n2 — )
CMP_NE      EXP    ( n1 n2 — n1 )
both RET    ( — )
```

Stack changes:

```
EXP:        ⟨⟩      2 elements are popped from the stack
CMP_NE      top element is popped from the stack
RET:        not affected
```

Flags:

```
CARRY flag  affected
BRANCH flag set, if (n1 <> n2)
```

X Y registers:

not affected

Bytes used:

1 – 2

See also:

0⟨⟩ 0= D⟨⟩

Example:

```
: NOT-EQUALS          ( show 7 <> 5 and 7 <> 7 and 8 <> 9      )
  7 5 CMP_NE          ( 7 5 — 7      [ BRANCH flag set ]      )
  7 <>                 ( 7 7 —      [ BRANCH flag NOT set ]    )
  8 9 <>               ( 8 9 — [ BRANCH and CARRY flag set ] )
;
```


<ROT

“Left-rote”

Purpose:

MOVE the TOS value to the third stack position. i.e.: performs a LEFTWARD rotation

Category: Stack operation (single-length) / qFORTH macro

Library implementation: CODE <ROT
ROT ROT
END-CODE

Stack effect: EXP (n1 n2 n3 — n3 n1 n2)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 2

See also: ROT 2<ROT 2ROT

Example:

```
: TD+          \ Add an 8-bit offset to a 12-bit value
  D+           \ Add the lower 8-bits  ( t1 d2 — n1 d3  )
  IF          \ IF an overflow to the 9th bit occurs, THEN
    ROT       ( n1 d3 — d3 n1                )
    1+        ( d3 n1 — d3 n1+1              )
    <ROT      ( d3 n1+1 — n1+1 d3            )
  THEN        ( n3 d3 — t3                  )
;
```

= **CMP_EQ**

“Equal”

Purpose:

Equality test for the top two 4-bit values on the stack. If the 2nd value on the stack is equal to the top of stack value, then the BRANCH flag in the CCR is set. Unlike standard FORTH, whereby a BOOLEAN value (0 or 1), depending on the comparison result, is pushed onto the stack.

Category: (=) : Comparison (single-length) / qFORTH macro
(CMP_EQ) : MARC4 mnemonic

Library implementation: CODE = CMP_EQ (n1 n2 — n1 [BRANCH flag])
DROP (n1 —)
END-CODE

MARC4 opcode: 06 hex (CMP_EQ)

Stack effect: =
EXP (n1 n2 —)
CMP_EQ EXP (n1 n2 — n1)
=,CMP_EQ RET (—)

Stack changes: EXP: = 2 elements are popped from the stack
CMP_EQ top element is popped from the stack
RET: not affected

Flags: CARRY flag affected
BRANCH flag set, if (n1 = n2)

X Y registers: not affected

Bytes used: 1 – 2

See also: 0<> 0= D<> D=

Example:

```
: EQUAL-TO          ( show 7 = 5 and 7 = 7 and 8 = 9      )
  7 5 CMP_EQ        ( 7 5 — 7 [ BRANCH flag NOT set ]    )
  7 =                ( 7 7 — [ BRANCH flag set ]          )
  8 9 =              ( 8 9 — [ CARRY flag set ]           )
;
```

>
CMP_GT

“Greater-than”

Purpose:

‘Greater-than’ comparison of the top two 4-bit values on the stack. If the 2nd value on the stack is greater than the top of stack value, then the BRANCH flag in the CCR is set. Unlike standard FORTH, whereby a BOOLEAN value (0 or 1), depending on the comparison result, is pushed onto the stack.

Category: (>) : Comparison (single-length) / qFORTH macro
(CMP_GT) : MARC4 mnemonic

Library implementation: CODE > CMP_GT (n1 n2 — n1 [BRANCH flag])
DROP (n1 —)
END-CODE

MARC4 opcode: 0A hex (CMP_GT)

Stack effect: > EXP (n1 n2 —)
CMP_GT EXP (n1 n2 — n1)
both RET (—)

Stack changes: EXP: > 2 elements are popped from the stack
CMP_GT top element is popped from the stack
RET: not affected

Flags: CARRY flag affected
BRANCH flag set, if (n1 > n2)

X Y registers: not affected.

Bytes used: 1 – 2

See also: < <> = >= <> D> D>= D<> D<

Example:

```
: GREATER-THAN      ( check 5 > 7 and 8 > 6 and 5 > 5      )
  5 7 CMP_GT        ( 5 7 — 5 [ CARRY set ]                )
  8 6 >             ( 5 6 8 — 5 [ BRANCH set ]              )
  5 CMP_GT DROP     ( 5 5 —                                )
;
```

>= **CMP_GE**

“Greater-or-equal”

Purpose:

‘Greater-than-or-equal’ comparison of the top two 4-bit values on the stack. If the 2nd value on the stack is greater than, or equal to the top of stack value, then the BRANCH flag in the condition code register (CCR) is set. Unlike standard FORTH, whereby a BOOLEAN value (0 or 1), depending on the comparison result, is pushed onto the stack.

Category:

(>=) : Comparison (single-length) / qFORTH macro
(CMP_GE) : MARC4 mnemonic

Library implementation:

```
CODE >= CMP_GE      ( n1 n2 — n1 [BRANCH flag])
                   DROP      ( n1 — )
END-CODE
```

MARC4 opcode:

0B hex (CMP_GE)

Stack effect:

```
>=      EXP      ( n1 n2 — )
CMP_GE  EXP      ( n1 n2 — n1 )
both    RET      ( — )
```

Stack changes:

```
EXP:      >=      2 elements are popped from the stack
          CMP_GE  top element is popped from the stack
RET:      not affected
```

Flags:

```
CARRY flag  affected
BRANCH flag set, if (n1 >= n2)
```

X Y registers:

not affected.

Bytes used:

1 – 2

See also:

<> = <= < > <> D<> D>= D<=

Example:

```
: GREATER-THAN-EQUALS      ( show 7 >= 5 and 7 >= 7 and 8 >= 9    )
  7 5 CMP_GE                ( 7 5 — 7 [ BRANCH flag set ]        )
  7 >=                       ( 7 7 — [ BRANCH flag set ]        )
  8 9 >=                     ( 8 9 — [ CARRY flag set ]          )
;
```


>R

“To-R”

Purpose:

Moves the top 4-bit value from the expression stack and pushes it onto the return stack.

>R pops the EXP stack onto the RET stack. To avoid corrupting the RET stack and crashing the program, each use of >R must be followed by a subsequent R> within the same colon definition.

Category:	Stack operation / assembler instruction
MARC4 opcode:	22 hex
Stack effect:	EXP (n1 —) RET (— u u n1)
Stack changes:	EXP: top element is popped from the stack RET: One element is pushed onto the stack
Flags:	not affected
X Y registers:	not affected
Bytes used:	1
See also:	I – R> 2R@ 2>R 2R> 3R@ 3>R 3R>

Example:

The sequence 3 1 (— 3 1)
 >R 1- R> (3 1 — 2 1)

temporarily moves the top value on the stack to the return stack so that the second value on the stack can be decremented.

```
: 2<ROT                    \ Move top byte to 3rd position on stack
  ROT >R                    ( d1 d2 d3 — d1 d2h d3                    )
  ROT >R                    ( d1 d2h d3 — d1 d3                        )
  ROT >R                    ( d1 d3 — d1h d3                         )
  ROT R>                    ( d1h d3 — d3 d1                         )
  R> R>                    ( d3 d1 — d3 d1 d2                       )
```

;

```
: JUGGLE-BYTES
  11h 22h 33h              ( — 11h 22h 33h                        )
  2<ROT                    ( 11h 22h 33h — 33h 11h 22h            )
  2SWAP                    ( 33h 11h 22h — 33h 22h 11h            )
  2OVER                    ( 33h 22h 11h — 33h 22h 11h 22h       )
```

;

?DO

“Query-DO”

Purpose:

Indicates the start of a (conditional) iterative loop.

?DO is used only within a colon definition in a pair with LOOP or +LOOP. The two numbers on top of the stack at the time ?DO is executed determine the number of times the loop repeats. The value on top is the initial loop index and the next value is the loop limit. If the initial loop index is equal to the limit, the loop is not executed (unlike DO). The control is transferred to the statement directly following the LOOP or +LOOP statement and the two values are popped from the stack.

Category: Control structure / qFORTH macro

Library implementation:

```
CODE ?DOOVER CMP_EQ 2>R
(S)BRA_$LOOP
_$DO:
END-CODE
```

Stack effect:

EXP	(limit index —)	
RET	(— u limit index)	if LOOP is executed
RET	(—)	if LOOP is not executed

Stack changes:

EXP: 2 elements are popped from the stack
 RET: 1 element is pushed onto the stack, if loop is executed
 not affected, if loop is not executed

Flags:

CARRY flag affected
 BRANCH flag set, if (limit = index)

X Y registers: not affected

Bytes used: 5

See also: DO LOOP +LOOP

Example:

VARIABLE Counter

```
: QUERY-DO
  0 Counter !           ( Counter := 0           )
  6 0 DO               ( repeat 6 times         )
    I 0                ( copy limit, index start = 0 )
    ?DO Counter 1+! LOOP ( first time not executed )
    Counter @ 10 >= ?LEAVE ( repeat,til count. >= 10 )
  LOOP
;
```

?DUP

“Query-doop”

Purpose:

Duplicates the top value on the stack only if it is non zero. ?DUP is equivalent to the standard FORTH sequence

DUP IF DUP THEN

but executes faster. ?DUP can simplify a control structure when it is used just before a conditional test (IF, WHILE or UNTIL).

Category: Stack operation (single-length) / qFORTH macro

Library implementation: CODE ?DUPDUP OR
(S)BRA_\$ZERO
DUP
_\$ZERO:
END-CODE

Stack effect: IF TOS = 0 THEN EXP (0 — 0)
ELSE EXP (n — n n)
RET (—)

Stack changes: EXP: A copy of the non zero top value is pushed onto the stack
RET: not affected

Flags: CARRY flag affected
BRANCH flag set, if (TOS = 0)

X Y registers: not affected

Bytes used: 4 – 5

See also: DUP 0= 0<>

Example:

```
: ShowByte          ( b_high b_low — b_high b_low      )
  DUP 3 OUT         ( show a byte value as hexadecimal  )
  SWAP              ( b_high b_low — b_low b_high      )
  ?DUP              ( DUP and write only if non zero    )
  IF 2 OUT THEN     ( suppress leading zero display    )
  SWAP              ( restore nibble sequence          )
;
```

?LEAVE

“Query-leave”

Purpose:

Conditional exit from within a control structure, if the previous tested condition was TRUE (ie. BRANCH flag is SET). ?LEAVE is the opposite to -?LEAVE (condition FALSE).
The standard FORTH word sequence

IF LEAVE ELSE word .. THEN
is equivalent to the qFORTH sequence

?LEAVE word ...

?LEAVE transfers control just beyond the next LOOP, +LOOP or #LOOP or any other loop structure like BEGIN ... UNTIL, WHILE
. REPEAT or BEGIN ... AGAIN, if the tested condition is TRUE.E

Category: Control structure / qFORTH macro

Library implementation: CODE
?LEAVE (S)BRA _\$LOOP (Exit LOOP if BRANCH set)
END-CODE

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

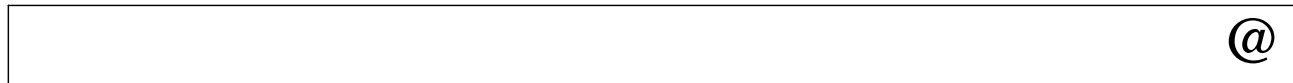
X Y registers: not affected

Bytes used: 1 – 2

See also: -?LEAVE

Example:

```
: QUERY-LEAVE
  3 BEGIN          ( 3 = initial start value          )
    1+            ( increment count 3 -> 9          )
    DUP           ( keep current value on the stack   )
    9 = ?LEAVE    ( when stack value = 9 then exit loop )
  AGAIN          ( Indefinite repeat loop           )
  DROP           ( the index                          )
;
```

“Fetch”

Purpose:

Copies the 4-bit value at a specified memory location to the top of the stack.

Category: Memory operation (single-length) / qFORTH macro

Library implementation: CODE @ Y! (addr —)
[Y]@ (— n)
END-CODE

Stack effect: EXP (RAM_addr — n)
RET (—)

Stack changes: EXP: 1 element is popped from the stack
RET: not affected

Flags: not affected

X Y registers: The contents of the Y or X register may be changed.

Bytes used: 2

See also: 2@ 3@ !

Example:

VARIABLE DigitPosition

8 ARRAY Result

```
: DisplayResult      ( write ARRAY 'Result' [7]..[0] to ports 7..0)
  7 DigitPosition !  ( initialize position )
  BEGIN DigitPosition @ Fh <> ( REPEAT, 'til index = Fh )
  WHILE DigitPosition @ DUP ( get digit pos: 7 .. 0 )
    Result INDEX @ ( get digit [7] .. [0] )
    OVER ( DPos val — DPos val DPos )
    OUT ( data port — Display digit )
    1- DigitPosition ! ( decrement digit & store )
  REPEAT ( REPEAT always;stop at WHILE )
```

;

```
: Display      ( write 0 .. 7 to ARRAY 'Result' [0.] .. [7] )
  8 0 DO
  I DUP Result INDEX !
  LOOP
  DisplayResult ( 'call' display routine. )
```

;

AGAIN

“AGAIN”

Purpose:

Part of the (infinite loop) BEGIN ... AGAIN control structure.

AGAIN causes an unconditional branch in program control to the word following the corresponding BEGIN statement.

Category: Control structure / qFORTH macro

Library implementation: CODE AGAIN
SET_BCF (execute an unconditional branch)
(S)BRA _\$BEGIN
END-CODE

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag set
BRANCH flag set

X Y registers: not affected

Bytes used: 2 – 3

See also: BEGIN UNTIL WHILE REPEAT

Example:

```
: INFINITE-LOOP
  3 BEGIN          ( 3 = initial start value          )
    1+            ( increment count 3 -> 9          )
    DUP           ( keep current value on the stack   )
    9 = ?LEAVE    ( when stack value = 9 then exit loop )
  AGAIN          ( repeat unconditional              )
;
```

ALLOT

“ALLOT”

Purpose:

Allocate (uninitialized) RAM space for the two stacks and global data of the type VARIABLE or 2VARIABLE.

Category: Predefined data structure

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 0

See also: VARIABLE 2VARIABLE

Example:

```
VARIABLE Limits 8 ALLOT      ( allocates 8 nibbles for the      )  
                             ( variable LIMITS                          )  
  
VARIABLE R0 21 ALLOT        ( allocate space for RETURN stack  )  
VARIABLE S0 9 ALLOT        ( allot 10 nibbles for EXP stack   )
```

AND

“AND”

Purpose:

Bitwise AND of the top two 4-bit stack elements leaving the 4-bit result on top of the expression stack.

Category: Arithmetic/logical (single-length) / assembler instruction

MARC4 opcode: 05 hex

Stack effect: EXP (n1 n2 — n1^n2)
RET (—)

Stack changes: EXP: top element is popped from the stack.
RET: not affected

Flags: CARRY flag not affected
BRANCH flag set, if (TOS = 0)

X Y registers: not affected

Bytes used: 1

See also: NOT OR XOR

Example:

```
: ERROR          ( what shall happen in error case:      )
  3R@
  3#DO           ( show PC, where CPU fails                )
    I OUT [ E 0 ] ( suppress compiler warnings.                          )
  #LOOP
;
```

```
: Logical
  1001b 1100b
    AND
  1000b <> IF ERROR THEN
  1010b 0011b
    AND
  0010b <> IF ERROR THEN
  1001b 1100b
    OR
  1101b <> IF ERROR THEN
  1010b 0011b
    OR
  1011b <> IF ERROR THEN
  1001b 1100b
    XOR
  0101b <> IF ERROR THEN
  1010b 0011b
    XOR
  1001b <> IF ERROR THEN
;
```


ARRAY

“ARRAY”

Purpose:

Allocates RAM space for storage of a short single-length (4-bit / nibble) array, using a 4-bit array index value. Therefore the number of 4-bit array elements is limited to 16.

The qFORTH syntax is as follows:

```
<number> ARRAY <name> [ AT <RAM-Addr> ]
```

At compile time, ARRAY adds <name> to the dictionary and ALLOTs memory for storage of <number> single-length values. At execution time, <name> leaves the RAM start address of the parameter field (<name> [0]) on the expression stack.

The storage ALLOTed by an ARRAY is not initialized.

Category:	Predefined data structure
Stack effect:	EXP (—) RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	0
See also:	2ARRAY LARRAY Index ERASE

Example:

6 ARRAY RawDATA AT 1Eh (RawDATA[0]...RawDATA[5])

: Init_ARRAY

5 (set initial value := 5)

6 0 DO (array index from 0 ... 5)

DUP I RawDATA INDEX ! (indexed store)

1- (decrement store value)

LOOP

DROP

;

(The result is: RawDATA[0] := 5 stored in RAM location 1E)

(RawDATA[1] := 4 stored in RAM location 1F)

(RawDATA[2] := 3 stored in RAM location 20)

(RawDATA[3] := 2 stored in RAM location 21)

(RawDATA[4] := 1 stored in RAM location 22)

(RawDATA[5] := 0 stored in RAM location 23)

AT

“AT”

Purpose:

Specifies the ABSOLUTE memory location AT where either a variable will be placed in RAM, a L/U table, string or a qFORTH word (subroutine / interrupt service routine) is forced to be placed in the ROM area.

Category: Predefined data structure

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 0

See also: VARIABLE ARRAY ROMCONST

Example:

VARIABLE State AT 3

: CheckState

State @ (fetch current state from RAM loc. 3)

CASE

0 OF State 1+! (increment contents of variable state)

ENDOF

15 OF State 1-!

ENDOF

ENDCASE

[Z] ; Test_Status (force placement in ZERO page)

: INTO

Fh State !

BEGIN

CheckState

State 1-!

UNTIL

; AT 400h (force placement at ROM address 400h)

BEGIN

“BEGIN”

Purpose:

Indicates the start of one of the following control structures:

```
BEGIN ... UNTIL
BEGIN ... AGAIN
BEGIN ... WHILE .. REPEAT
```

BEGIN marks the start of a sequence that may be repetitively executed. It serves as a branch destination (`_$BEGINxx:`) for the corresponding UNTIL, AGAIN or REPEAT statement.

Category: Control structure

Library implementation: CODE BEGIN
 _\$BEGIN: [E 0 R 0]
 END-CODE

Stack effect: EXP (—)
 RET (—)

Stack changes: EXP: not affected
 RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 0

See also: UNTIL AGAIN REPEAT WHILE ?LEAVE -?LEAVE

Example:

```
: BEGIN-UNTIL
  3 BEGIN          ( increment value from 3 until 9          )
    1+ DUP 9 =    ( DUP the current value because the      )
    UNTIL         ( comparison will DROP it                )
    DROP          ( BRANCH and CARRY flags will be set    )
;
: BEGIN-AGAIN     ( do the same with an infinite loop      )
  3 BEGIN
    1+ DUP
    9 = ?LEAVE
    AGAIN
  DROP
;
: BEGIN-WHILE-REPEAT ( do the same with a WHILE-REPEAT loop )
  3 BEGIN
    DUP 9 <>
    WHILE          ( REPEAT increment while not equal 9    )
      1+
    REPEAT
  DROP
;
```

CASE

“CASE”

Purpose:

Indicates the start of a CASE ... OF ... ENDOF ... ENDCASE control structure. Using a 4-bit index value on TOS, CASE compares it sequentially with each value in front of an OF ... ENDOF pair until a match is found. When the index value equals one of the 4-bit OF values, the sequence between that OF and the corresponding ENDOF is executed. Control then branches to the word following ENDCASE.

If no match is found, the ENDCASE will DROP the index value from the EXP stack. The 'otherwise' case may be handled by qFORTH words placed between the last ENDOF and ENDCASE.

NOTE: However, the 4-bit index value must be perserved across the 'otherwise' sequence so that ENDCASE can drop it !

Category:	Control structure
Library implementation:	CODE CASE _\$CASE: [E 0 R 0] END-CODE
Stack effect:	EXP (n — n) RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	0
See also:	OF ENDOF ENDCASE

Example:

```
5h CONSTANT Keyboard
1  CONSTANT TestPort1

: ONE  1 TestPort1 OUT ;           ( write 1 to the 'TestPort1' )
: TWO  2 TestPort1 OUT ;           ( write 2 to the 'TestPort1' )
: THREE 3 TestPort1 OUT ;          ( write 3 to the 'TestPort1' )
: ERROR DUP TestPort1 OUT ;        ( dump wrong input to the port )

( duplicate value for the following ENDCASE; it drops one )

: CASE-Example
  KeyBoard IN                      ( request 1-digit keyboard input )
  CASE                              ( depending of the input value, )
    1 OF ONE  ENDOF                 ( one of these words will be acti- )
    2 OF TWO  ENDOF                 ( vated. )
    3 OF THREE ENDOF
    ERROR                            ( otherwise ... )
  ENDCASE                          ( n — )
;
```


CCR!

“CCR-store”

Purpose:

Store the 4-bit TOS value in the condition code register (CCR).

NOTE: All flags will be altered by this command !

Category: Assembler instruction

MARC4 opcode: 0E hex

Stack effect: EXP (n —)
RET (—)

Stack changes: EXP: 1 element is popped from the stack
RET: not affected

Flags: CARRY flag set, if bit 3 of TOS was set
BRANCH flag set, if bit 1 of TOS was set
I_ENABLE flag set, if bit 0 of TOS was set

X Y registers: not affected

Bytes used: 1

See also: EI DI CCR@ SET_BCF CLR_BCF

Example 1:

```
: INT5          ( timer interrupt service routine      )
  CCR@          ( save the current condition codes    )
  Inc_Time      ( call procedure.                    )
  CCR!          ( restore CCR status                  )
;              ( RTI & Enable interrupts              )
```

NOTE: CCR@/! and X/Y@/! will be inserted in INTx-routines by the compiler automatically.

Example 2:

```
CODE EI          ( enable all interrupts              )
  0001b CCR!
END_CODE
```

CCR@

“CCR-fetch”

Purpose:

Save the contents of the condition code register on TOS.

Category: Assembler instruction

MARC4 opcode: 0D hex

Stack effect: EXP (— n)
RET (—)

Stack changes: EXP: 1 element is pushed onto the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: CCR! EI DI

Example:

```
1 CONSTANT Port1
: ?ERROR          ( error routine: are the numbers equal      )
  <> IF           ( if unequal, then write Fh to port1.      )
    Fh Port1 OUT
    THEN          ( two digits are dropped from the        )
;

: ADD_ADDC_TEST  ( add up to 8-bit numbers                    )
  Ah Ch +        ( 10 12 — 6 + CARRY flag set              )
  CCR@ SWAP      ( 6 — [C-BI flags] 6 )
  6 ?ERROR       ( check correct result ( 6 6 — )          )
  CCR!           ( restore CARRY flag setting                )
  Dh 6h +C       ( 13 6 [CARRY] — 4 + CARRY flag set      )
  4 ?ERROR       ( check correct result ( 4 4 — )          )
;
```

CLR_BCF

“Clear BRANCH- and CARRY-Flag”

Purpose:

Clear the BRANCH and CARRY flag in the condition code register.

Category: qFORTH macro

Library implementation: CODE CLR_BCF
0 ADD (reset CARRY & BRANCH flag)
END-CODE

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag reset
BRANCH flag reset

X Y registers: not affected

Bytes used: 2

See also: SET_BCF TOG_BF

Example:

```
8 ARRAY Result      ( 8 digit BCD number array definition )
: DIG+              ( add 1 digit to an 8 digit BCD number )
  Y! CLR_BCF        ( digit LSD_addr — digit ; clear flgs )
  8 #DO             ( loop maximal 8 times. )
    [Y]@ +C DAA     ( add digit & do a decimal adjust. )
    [Y-]! 0         ( store; add 0 to the next digit. )
    -?LEAVE        ( if no more carry, then leave loop. )
  #LOOP
  DROP              ( last 0 isn't used. )
;                  ( EXIT – return )

: ADD-UP-NUMBERS
  Result 8 ERASE    ( clear the array. )
  15 #DO           ( loop 15 times. )
    9 Result [7]   ( put address of last nibble to TOS,-1 )
  DIG+            ( add 15 times 9 to RESULT )
  #LOOP           ( BRANCH conditionally to begin of loop )
;                ( result: 9 * 15 = 135 )
```

CODE

“CODE”

Purpose:

Begins a qFORTH macro definition where both MARC4 assembler instructions and qFORTH words may be included.

Macros defined as CODE ... END-CODE are executed identically to words created as colon definitions (i.e. : ... ;) – except that no CALL and EXIT is placed in the ROM.

The macro bytes are placed from the compiler in the ROM to every program sequence, where they should be activated. MACROs are often used to improve run-time optimization, as long as the macro is used by the program not too often.

NOTE: qFORTH word definitions that change the return stack level (>R, 2>R, ... 3R>, DROPR) require CODE ... END-CODE implementations, because the return address would no more be available.

Category:	Predefined structure
Stack effect:	EXP (—) RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	0
See also:	END-CODE colon definition (:) EXIT (;

Example:

```
5 CONSTANT Port5
3 ARRAY ReceiveData    ( 12-bit data item          )
(----- CODE to shift right a 12-bit data word  )
CODE ShiftRDBits
  ReceiveData Y!
  [Y]@ ROR [Y]!
  [+Y]@ ROR [Y]!      ( rotate thru CARRY          )
  [+Y]@ ROR [Y]!
END-CODE
```

```
: Receive_Bit
  ReceiveDate Y!      ( write data to the array:          )
  5 [Y]! Ah [+Y]! 1 [+Y]!
  Port5 IN SHL        ( Read input from IP53          )
  ShiftRDBits         ( shift 'ReceiveData' 1 bit right )
;
```


\$ INCLUDE \$RAMSIZE \$ROMSIZE

'COMPILER DIRECTIVES'

"Dollar-include" "Dollar-RAMSize" ...

Purpose:

The MARC4 qFORTH compiler's behavior during compilation may be controlled by including \$-Sign directives within the source code file. These \$-sign directives consist of one keyword which may be followed by at least one parameter.

Category: Compiler word / directives

\$INCLUDE: Include another source file (up to 4 levels) at compilation time.

Note: To avoid a warning from the compiler, set all of the following directives. The default values could be changed with up-dated compiler versions.

\$RAMSIZE: Specifies the RAM size of the target processor. Default size is 255 nibbles (from \$00 .. \$FF). Some processors contain 253 nibbles only.

\$ROMSIZE: Specifies the ROM size of the target processor. Default size is 4.0K (from \$000 .. \$FFF) ;
The constants are as follows: 1.0K = 3Fh, 2.5K = 9Fh and 4.0K = FFh. With '\$ROMSIZE' you can access this constant in your source program.

Example:

```
$INCLUDE Timer.INC
( Predefined constants:                                )
255 2CONSTANT $RAMSIZE                               ( for 253 RAM nibbles [3 auto sleep] )
1.5k 2CONSTANT $ROMSIZE                              ( 1535 ROM bytes – 2 b. for check sum )
                                                ( resulting constant [$ROMSIZE] = 5Fh )
VARIABLE R0 27 ALLOT                                ( return stack: 28 nibbles for 7 level )
VARIABLE S0 19 ALLOT                                ( data stack: 20 nibbles )
```

CONSTANT

“CONSTANT”

Purpose:

Creates a 4-bit constant; implemented in a qFORTH program as:

n CONSTANT <name>

with $0 \leq n \leq 15$ or $0 \leq n \leq Fh$.

Creates a dictionary entry for <name>, so that when <name> is later ‘executed’, the value n is left on the stack. This is similar to an assembler EQUate statement, in that it assigns a value to a symbol.

Category: Predefined data structure

Stack effect: EXP (— n) on runtime.
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 0

See also: 2CONSTANT VARIABLE 2VARIABLE

Example:

```
4h CONSTANT #Nibbles           ( value of valid bits           )
20 2CONSTANT Nr_of_Apples      ( value > 15 [Fh]             )
03h 2CONSTANT Nr_of_Bananas
8  CONSTANT NumberOfBits      ( hexadecimal, decimal or ...  )
0011b CONSTANT BitMask        ( binary.                       )
```

Example:

```
  Nr_of_Apples Nr_of_Bananas
  D+           ( calculate nr of fruits           )
  DUP BitMask AND DROP      ( lower nibble: odd or even ?   )
  IF
    NumberOfBits           ( do it with every bit:        )
    #DO ... #LOOP
  THEN
;
```

D+

“D-plus”

Purpose:

D+ adds the top two 8-bit values on the stack and leaves the result on the expression stack.

Category:

Arithmetic/logical (double-length) / qFORTH colon definition

Library implementation:

```
: D+ ROT      ( d1h d1l d2h d2l — d1h d2h d2l d1l)
      ADD      ( d1h d2h d2l d1l — d1h d2h d3l   )
      <ROT     ( d1h d2h d3l — d3l d1h d2h     )
      ADDC    ( d3l d1h d2h — d3l d3h         )
      SWAP    ( d3l d3h — d3                  )
;

```

Stack effect:

EXP (d1 d2 — d_sum)
RET (—)

Stack changes:

EXP: 2 elements are popped from the stack
RET: not affected

Flags:

CARRY flag set on overflow on higher nibble
BRANCH flag = CARRY flag

X Y registers:

not affected

Bytes used:

7

See also:

D- 2! 2@ D+! D-! D2/ D2*

Example:

```
: DC Double Add
  10h 0 5 D+      ( result: — 15 ; no flags   )
  18h D+          ( result: — 2D ; no flags   )
  14h D+          ( result: — 41 ; no flags   )
  C0h D+ 2DROP   ( result: — 01 ; C & B flag )
;
```

D+!

“D-plus-store”

Purpose:

ADD the TOS 8-bit value to an 8-bit variable in RAM and store the result in that variable. On function entry, the higher nibble address of the variable is the TOS value.

Category: Arithmetic/logical (double-length) / qFORTH colon definition

Library implementation:

```

: D+!   Y!           ( nh nl address — nh nl)
  [+Y]@ +           ( nh nl — nh nl'  )
  [Y-]!           ( nh nl' — nh    )
  [Y]@ +c         ( nh — nh'    )
  [Y]!           ( nh' —      )

```

;

Stack effect: EXP (d RAM_addr —)
RET (—)

Stack changes: EXP: 4 elements are popped from the stack
RET: not affected

Flags: CARRY flag set on overflow on higher nibble
BRANCH flag = CARRY flag

X Y registers: The contents of the Y register will be changed.

Bytes used: 8

See also: The other double-length qFORTH dictionary words, like
D- D+ 2! 2@ D-! D2/ D2* D< D> D<> D= D<= D>= D0=
D0<>

Example:

2VARIABLE count AT 43h

: DCompAri

13h count 2! (RAM [43] = 1 ; RAM [44] = 3)

count 2@ (— 1 3)

2DROP

55h count D+! (68 in the RAM ; no flags)

b5h count D+! (1D in the RAM ; C & B flag)

;

D-

“D-minus”

Purpose:

D- subtracts the top two 8-bit values on the EXP stack and leaves the result on the EXP stack.

Category:

Arithmetic/logical (double-length) / qFORTH colon definition

Library implementation:

```
: D- ROT      ( d1h d1l d2h d2l — d1h d2h d2l d1l )
  SWAP      ( d1h d2h d2l d1l — d1h d2h d1l d2l )
  SUB       ( d1h d2h d1l d2l — d1h d2h d3l   )
  <ROT      ( d1h d2h d3l — d3l d1h d2h   )
  SUBB      ( d3l d1h d2h — d3l d3h     )
  SWAP      ( d3l d3h — d3              )
```

;

Stack effect:

```
EXP ( d1 d2 — d1-d2   )
RET ( —              )
```

Stack changes:

EXP: 2 elements are popped from the stack
RET: not affected

Flags:

CARRY flag set on arithmetic underflow
BRANCH flag = CARRY flag

X Y registers:

not affected

Bytes used:

8

See also:

D+ 2! 2@ D+! D-! D2/ D2* D< D>

Example:

```
: DCompAri
  15h 13h
  D- 2DROP          ( result: — 02 ; no flags   )
  13h 15h
  D- 2DROP          ( result: — FE ; C & B flag )
  18h 18h D- 2DROP ( result: — 00 ; no flags   )
;
```

D-!

“D-minus-store”

Purpose:

Subtract the top 8-bit value from an 8-bit variable in RAM and store the result in that variable. The address of the variable is the TOS value.

Category: Arithmetic/logical (double-length) / qFORTH colon definition

Library implementation:

```

: D-!   Y!           ( nh nl address — nh nl   )
[+Y]@   ( nh nl — nh nl @RAM[Y] )
SWAP -   ( nh nl @RAM[Y] — nh nl )
[Y-]!   ( nh nl' — nh         )
[Y]@    ( nh — nh @RAM[Y+1]   )
SWAP -c ( nh @RAM[Y+1] — nh'   )
[Y]!    ( nh' —                )

```

;

Stack effect: EXP (d RAM_addr —)
RET (—)

Stack changes: EXP: 4 elements are popped from the stack
RET: not affected

Flags: CARRY flag set on arithmetic underflow
BRANCH flag = CARRY flag

X Y registers: The contents of the Y register will be changed.

Bytes used: 10

See also: D- D+ 2! 2@ D+!

Example:

```
2VARIABLE Fred
: DCompAri
  13h Fred 2!
  11h Fred D-!      ( 02 in the RAM ; no flags      )
  11h Fred D-!      ( F1 in the RAM ; C & B flag    )
;
```

D0<>

“D-zero-not-equal”

Purpose:

Compares the 8-bit value on top of the stack to zero. Instead of pushing a Boolean TRUE flag on the stack if the byte on top of the stack is non zero, ‘D0<>’ sets the BRANCH flag in the CCR.

Category: Arithmetic/logical (double-length) / qFORTH macro

Library implementation: CODE D0<> OR (n1 n2 — n3 [if 0 then BRANCH flag])
 DROP (n3 —)
 TOG_BF (Toggle BRANCH flag)
 END-CODE

Stack effect: EXP (d —)
 RET (—)

Stack changes: EXP: 2 elements will be popped from the stack
 RET: not affected

Flags: CARRY flag not affected
 BRANCH flag Set, if (d <> 0)

X Y registers: not affected

Bytes used: 3

See also: D- D+ D< D> D<> D= D<= D>= D0=

Example:

```
1 CONSTANT true
0 CONSTANT false

: DCompare
  12h D0<>
  IF true ELSE false THEN DROP ( result is 'true' )
  0 0 D0<>
  IF true ELSE false THEN DROP ( result is 'false' )
;
```

D0=

“D-zero-equal”

Purpose:

Compare the 8-bit value on top of the stack to zero. Instead of pushing a Boolean TRUE flag on the stack if the byte on top of the stack is zero, ‘D0=’ sets the BRANCH flag.

Category: Arithmetic/logical (double-length) / qFORTH macro

Library implementation: CODE D0= OR (n1 n2 — n3 [BRANCH flag re/set])
 DROP (n3 — [BRANCH flag])
 END-CODE

Stack effect: EXP (d —)
 RET (—)

Stack changes: EXP: 2 elements will be popped from the stack
 RET: not affected

Flags: CARRY flag not affected
 BRANCH flag set, if (d = 0)

X Y registers: not affected

Bytes used: 2

See also: D- D+ D< D> D<> D= D<= D>= D0<>

Example:

```
1 CONSTANT true
0 CONSTANT false
```

```
: DCompare
```

```
  12h D0=
```

```
  IF true ELSE false THEN DROP      ( result is 'false' )
```

```
  0 0 D0=
```

```
  IF true ELSE false THEN DROP      ( result is 'true' )
```

```
;
```


D2*

“D-two-multiply”

Purpose:

Multiplies the 8-bit value on top of the stack by 2.

Category:

Arithmetic/logical (double-length) / qFORTH macro

Library implementation:

```
CODE D2*  SHL      ( dh dl — dh dl*2  )
          SWAP    ( dh dl*2 — dl*2 dh  )
          ROL     ( dl*2 dh — dl*2 dh*2 )
          SWAP    ( dl*2 dh*2 — d*2   )
END-CODE
```

Stack effect:

```
EXP      ( d — d*2  )
RET      ( —       )
```

Stack changes:

EXP: not affected
RET: not affected

Flags:

CARRY flag set on arithmetic overflow
BRANCH flag = CARRY flag

X Y registers:

not affected

Bytes used:

4

See also:

D- D+ D2/ D<> D= D<= D>= D0= D0<>

Example:

```
: DMultiply
  0 3
  D2*          ( 03h -> 06h and no flags )
  D2*          ( 06h -> 0Ch and no flags )
  D2* 2DROP    ( 0Ch -> 18h and no flags )
  90h D2* 2DROP ( 90h -> 20h and C & B flag)
;              ( [90h *2 = 120h]          )
```

D2/

“D-two-divide”

Purpose:

Divides the 8-bit value on top of the stack by 2.

Category:

Arithmetic/logical (double-length) / qFORTH macro

Library implementation:

```
CODE D2/  SWAP      ( dh dl/2 — dl dh          )
          SHR       ( dl dh — dl dh/2 [CARRY flag] )
          SWAP     ( dl dh/2 [CARRY flag] — dh/2 dl )
          ROR      ( dh/2 dl [CARRY flag] — d/2    )
END-CODE
```

Stack effect:

```
EXP      ( d — d/2 )
RET      ( —      )
```

Stack changes:

EXP: not affected
RET: not affected

Flags:

CARRY flag set, if LSB of byte on TOS was set
BRANCH flag = CARRY flag

X Y registers:

not affected

Bytes used:

4

See also:

D- D+ D+! D-! D2* D<> D=

Example:

```
: D2Divide
  13h
  D2/          ( 13h -> 09h and C & B flag)
  D2/          ( 09h -> 04h and C & B flag)
  D2/          ( 04h -> 02h and no flags )
  D2/          ( 02h -> 01h and no flags )
  D2/ 2DROP   ( 01h -> 00h and C & B flag)
;
```

D<

“D-less-than”

Purpose:

‘Less-than’ comparison for the top two unsigned 8-bit values. Instead of pushing a Boolean TRUE flag onto the stack if the 2nd value on the stack is ‘less-than’ the TOS value, ‘D<’ sets the BRANCH flag.

Category:

Arithmetic/logical (double-length) / qFORTH colon definition

Library implementation:

```
: D<          ROT      ( d1 d2 — d1h d2 d1l      )
                2>R     ( d1h d2h d2l d1l — d1h d2h      )
                OVER    ( d1h d2h — d1h d2h d1h      )
                CMP_LT  ( d1h d2h d1h — d1h d2h [B-flag] )
BRA  _BIGGER   ( jump if upper nibble is bigger      )
                CMP_LT  ( d1h d2h — d1h [BRANCH flag] )
BRA  _IS_LESS  ( jump if upper nibble is smaller     )
                2R@     ( d1h — d1h d2l d1l          )
                CMP_LE  ( d1h d2l d1l — d1h d2l      )
_BIGGER: TOG_BF ( correct the BRANCH flag           )
                DROP    ( d1h d2l — d1h              )
_IS_LESS: DROP  ( d1h —                              )
                DROPR   ( skip lower nibbles from RET stack )

[ E -4 R 0 ]
;
```

Stack effect:

```
EXP      ( d1 d2 — )
RET      ( —      )
```

Stack changes:

EXP: 4 elements are popped from the stack
RET: not affected

Flags:

CARRY flag affected
BRANCH flag = set, if (d1 < d2)

X Y registers:

not affected

Bytes used:

16

See also:

D> D<> D= D<= D>= D0= D0<>

Example:

```
1 CONSTANT true
0 CONSTANT false

: DCompare
  12h 15h D<
  IF true ELSE false THEN DROP      ( result is 'true' )
  15h 12h D<
  IF true ELSE false THEN DROP      ( result is 'false' )
  18h 18h D<
  IF true ELSE false THEN DROP      ( result is 'false' )
;
```


Example:

```
1 CONSTANT true
0 CONSTANT false

: DCompare
  12h 15h D<=
  IF true ELSE false THEN DROP      ( result is 'true' )
  15h 12h D<=
  IF true ELSE false THEN DROP      ( result is 'false' )
  18h 18h D<=
  IF true ELSE false THEN DROP      ( result is 'true' )
;
```


D<>

“D-not-equal”

Purpose:

Inequality test for the top two 8-bit values. Instead of pushing a Boolean true flag onto the stack if the 2nd value on the stack is 'not-equal' to the TOS value, 'D<>' sets the BRANCH flag.

Category: Arithmetic/logical(double-length) / qFORTH colon definition

Library implementation:

```

: D<>  ROT          ( d1h d1l d2h d2l — d1h d2h d2l d1l )
  CMP_NE ( d1h d2h d2l d1l — d1h d2h d2l )
  DROP   ( d1h d2h d2l — d1h d2h )
  BRA _NOT_EQ ( jump if lower nibbles not equal )
  CMP_NE ( d1h d2h — d1h [BRANCH flag] )
  DUP    ( d1h — d1h d1h )
  _NOT_EQ: 2DROP ( d1h d2h — )
[ E -4 R 0 ]
;
```

Stack effect:

```

EXP ( d1 d2 — )
RET ( — )
```

Stack changes:

```

EXP: 4 elements are popped from the stack
RET: not affected
```

Flags:

```

CARRY flag affected
BRANCH flag = set, if (d1 <> d2)
```

X Y registers: not affected

Bytes used: 10

See also: D< D> D= D<= D>= D0= D0<>

Example:

```
1 CONSTANT true
0 CONSTANT false
```

```
: DCompare
  12h 15h D<>
  IF true ELSE false THEN DROP      ( result is 'true' )
  15h 12h D<>
  IF true ELSE false THEN DROP      ( result is 'true' )
  18h 18h D<>
  IF true ELSE false THEN DROP      ( result is 'false' )
;
```

D=

“D-equal”

Purpose:

Equality test for the top two 8-bit values. Instead of pushing a Boolean TRUE flag onto the stack if the 2nd value is ‘equal’ to the TOS value, ‘D=’ sets the BRANCH flag. This macro uses the colon definition ‘D<>’.

Category: Arithmetic/logical(double-length) / qFORTH macro

Library implementation: CODE D=
 D<>
 TOG_BF
 END-CODE

Stack effect: EXP (d1 d2 —)
 RET (—)

Stack changes: EXP: 4 elements are popped from the stack
 RET: not affected

Flags: CARRY flag affected
 BRANCH flag = set, if (d1 = d2)

X Y register: not affected

Bytes used: 3 + ‘D<>’

See also: D< D> D<> D<= D>= D0= D0<>

Example:

```
1 CONSTANT true
0 CONSTANT false

: DCompare
  12h 15h D=
  IF true ELSE false THEN DROP      ( result is 'false' )
  15h 12h D=
  IF true ELSE false THEN DROP      ( result is 'false' )
  18h 18h D=
  IF true ELSE false THEN DROP      ( result is 'true' )
;
```

D>

“D-greater-than”

Purpose:

'Greater-than' comparison for the top two 8-bit values. Instead of pushing a Boolean TRUE flag onto the stack if the 2nd value is 'greater-than' to the TOS value, 'D>' sets the BRANCH flag.

Category: Arithmetic/logical(double-length) / qFORTH colon definition

Library implementation:

```

: D>  ROT          ( d1 d2 — d1h d2 d1l          )
      2>R          ( d1h d2h d2l d1l — d1h d2h    )
      OVER        ( d1h d2h — d1h d2h d1h        )
      CMP_GT      ( d1h d2h d1h — d1h d2h [B-flag] )
      BRA _SMALLER ( jump if upper nibble is smaller )
      CMP_GT      ( d1h d2h — d1h [BRANCH flag]    )
      BRA _IS_HUGH ( jump if upper nibble is bigger )
      2R@         ( d1h — d1h d2l d1l            )
      CMP_GE      ( d1h d2l d1l — d1h d2l        )
      _SMALLER: TOG_BF ( correct the BRANCH flag )
      DROP        ( d1h d2? — d1h                )
      _IS_HUGH: DROP ( d1h —                    )
      DROPR       ( skip lower nibbles from RET stack )
      [ E -4 R 0 ]
      ;

```

Stack effect:

```

EXP      ( d1 d2 — )
RET      ( —      )

```

Stack changes:

```

EXP: 4 elements are popped from the stack
RET: not affected

```

Flags:

```

CARRY flag affected
BRANCH flag = set, if (d1 > d2)

```

X Y registers: not affected

Bytes used: 16

See also: D< D<> D= D<= D>= D0= D0<>

Example:

```
1 CONSTANT true
0 CONSTANT false

: DCompare
  12h 15h D>
  IF true ELSE false THEN DROP      ( result is 'false' )
  15h 12h D>
  IF true ELSE false THEN DROP      ( result is 'true' )
  18h 18h D>
  IF true ELSE false THEN DROP      ( result is 'false' )
;
```

D>=

“D-greater-equal”

Purpose:

'Greater-than-or-equal' comparison for the top two 8-bit values. Instead of pushing a Boolean TRUE flag onto the stack if the 2nd value is 'greater-than-or-equal' to the TOS value, 'D>=' sets the BRANCH flag.

Category: Arithmetic/logical(double-length) / qFORTH colon definition

Library implementation: CODE D>= D<
TOG_BF
END-CODE

Stack effect: EXP (d1 d2 —)
RET (—)

Stack changes: EXP: 4 elements are popped from the stack
RET: not affected

Flags: CARRY flag affected
BRANCH flag = set, if (d1 >= d2)

X Y registers: not affected

Bytes used: 3 + 'D<'

See also: D< D> D<> D= D<= D0= D0<>

Example:

```
1 CONSTANT true
0 CONSTANT false

: DCompare
  12h 15h D>=
  IF true ELSE false THEN DROP      ( result is 'false' )
  15h 12h D>=
  IF true ELSE false THEN DROP      ( result is 'true' )
  18h 18h D>=
  IF true ELSE false THEN DROP      ( result is 'true' )
;
```


D>S**NIP**

“Double-to-single” “NIP”

Purpose: Transform a 8-bit value to an 4-bit value. Drops second 4-bit value from the stack.

Category: Stack operation (single-length) / qFORTH macro

Library implementation: `CODE D>S | NIP SWAP (d — n 0) or (n1 n2 — n2 n1)`
`DROP (n 0 — n) (n2 n1 — n2)`
`END-CODE`

Stack effect: `EXP (d — n) or EXP (n1 n2 — n2)`
`RET (—)`

Stack changes: `EXP`: 1 element is popped from the stack
`RET`: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 2

See also: `S>D 2NIP`

Example 1:

```
: NIP_Example
10H          (  — 1 0          )
  3          ( 1 0 — 1 0 3    )
  NIP        ( 1 0 3 — 1 3    )
  2DROP      ( 1 3 —          )
;
```

Example 2:

```
( ***** )
: StoDDtoS
  4          (  — 4          )
  S>D        (  4 — 0 4      )
  D>S        (  0 4 — 4      )
  DROP       (  4 —          )
;
```

Example 3:

```
Library implementation: of 'DEPTH'
: DEPTH  SP@ S0 D- ( — SPh SP1 S0h S0l — diff )
  NIP 1- ( diff — n )
;
```

DAA

“D-A-A”

Purpose:

Decimal-arithmetic-adjust for BCD arithmetic, if the digit on top of stack is greater than 9 or the CARRY flag is set.

Category: Assembler instruction

MARC4 opcode: 16 hex

Stack effect: IF TOS > 9 or CARRY-in = 1
 THEN EXP (n — n+6)
 ELSE EXP (n — n)
 RET (—)

Stack changes: EXP: not affected
 RET: not affected

Flags: CARRY flag set, if (TOS > 9) or (CARRY-in = 1)
 BRANCH flag = CARRY flag

X Y registers: not affected

Bytes used: 1

See also: +C DAS SET_BCF

Example:

```
: DAA_Example
  2 SET_BCF DAA      ( result is: 2 — 8 & C flag      )
  2 CLR_BCF DAA      ( result is: 2 — 2 no flag        )
  11 DAA             ( result is: B — 1 & C flag [Bh = 11] )
  Bh SET_BCF DAA     ( result is: B — 1 & C flag        )
  2DROP 2DROP
;
```

Another example for DAA, see DAS–entry (next page).

DAS

“D–A–S”

Purpose:

Decimal arithmetic for BCD subtraction, computes a 9's complement.

Category: qFORTH macro

Library implementation: CODE DAS
 NOT 10 +c (n — 9–n)
 END–CODE

Stack effect: EXP (n — 9–n)
 RET (—)

Stack changes: EXP: not affected
 RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 3

See also: NOT +C DAA

Example:

```
(----- CONSTANT definition part -----)
8 CONSTANT BCD#      ( number of BCD digits      )

----- GLOBAL RAM variables -----)
BCD#  ARRAY input2
BCD#  ARRAY input1

( ===== )
: DIG-      ( digit count LSD_Addr ---      )
Y! SWAP DAS SWAP  ( generate 9's complement      )
  #DO      ( digit count --- digit      )
    [Y]@ + DAA [Y-]! 10 ( transfer CARRY on stack      )
    -?LEAVE ( exit LOOP, if NOT CARRY      )
  #LOOP    ( repeat until index = 0      )
  DROP ;   ( skip TOS overflow digit      )
( ===== )
: BCD-      ( count LSD_Addr1 LSD_Addr2 ---      )
Y! X! SET_BCF ( set CARRY and pointer reg's      )
  #DO
    [Y]@ [X-]@ DAS ( 9's complement generation      )
    + DAA [Y-]!
  #LOOP ;

: int0 ( ===== )
  BCD# Input2 [7] Input1 [7] BCD-      ( Inp1:=Inp1-Input2      )
  3 BCD# Input1 [7] DIG-      ( Input1 := Input1 - 3      )
;
```

DECR

“Dec-R”

Purpose:

Decrements the lowest nibble (i.e. the loop index) on the return stack.

Category: Assembler instruction

MARC4 opcode: 1C hex

Stack effect: EXP (—)
RET (u|u|n — u|u|n-1)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag not affected
BRANCH flag = set, if (n-1 <> 0)

X Y registers: not affected

Bytes used: 1

See also: #DO .. #LOOP (#LOOP uses DECR)

Example 1:

```

: DECR_Example
  3 >R
  DECR          ( RET stack: u|u|3 — u|u|2 & B flag   )
I DROP         ( EXP stack: — 2 —                    )
  DECR          ( RET stack: u|u|2 — u|u|1 & B flag   )
  DECR          ( RET stack: u|u|1 — u|u|0 no flag    )
  DECR          ( RET stack: u|u|0 — u|u|F & B flag   )
  DECR          ( RET stack: u|u|F — u|u|E & B flag   )
  DECR          ( RET stack: u|u|E — u|u|D & B flag   )
  DECR          ( RET stack: u|u|D — u|u|C & B flag   )
  DROPR        ( pop “one element” from return stack )
;              ( “one element” = 3 nibbles           )

```

Example 2:

Library implementation: of #LOOP :

```

( Purpose: #LOOP – macro is used to terminate a #DO loop. )
( On each iteration of a #DO loop, #LOOP decrements the )
( loop index on the return stack. It then compares the index )
( to zero and determine whether the loop should terminate. )
( If the new index is decremented to zero, the loop is )
( terminated and the loop index is discarded from the return )
( stack. Otherwise, control jumps back to the point just after )
( the corresponding begin of the #DO macro. )
\   IF   Index-1 > 0
\   THEN RET      ( u|u|Index — u|u|Index-1 )
\   ELSE RET      ( u|u|Index — )
CODE #LOOP DECR  ( RET: u|u|Index — u|u|Index )
    BRA   _$#DO   ( IF Index > 0, loop again )
    _$LOOP: DROPR ( forget index on return stack )
END-CODE

```


DEPTH

“DEPTH”

Purpose:

Leaves the currently used depth of the expression stack on top of the stack.

Category:

Stack operation / interrupt handling / qFORTH colon definition

Library implementation:

```
: DEPTH  SP@ S0 D- ( — SPh SP1 S0h S0l — diff )
          NIP 1-      ( diff — n      )
          ;
```

Stack effect:

EXP (— n) (n <= Fh)
RET (—)

Stack changes:

EXP: 1 element will be pushed onto the stack
RET: not affected

Flags:

CARRY flag affected
BRANCH flag set, if (depth = 0)

X Y registers:

not affected

Bytes used:

9

See also:

RFREE RDEPTH

Example:

```
( ***** )
: DEPTH-Example
DEPTH          ( [*1] — 5 )
1 2            ( 5 — 5 1 2 )
DEPTH          ( 5 1 2 — 5 1 2 8 )
3 4            ( 5 1 2 8 — 5 1 2 8 3 4 )
DEPTH          ( 5 1 2 8 3 4 — [*1] 5 1 2 8 3 4 b )
2DROP 2DROP   ( drop the 4 nibbles and )
2DROP DROP    ( the 3 result values. )
;

: INT0
(CCR@/! X@/! Y@/! is – if the registers are used – automatically )
( implemented by the compiler [*1]. )
DEPTH-Example
;
```

“Disable–Interrupt” or “D–I”

Purpose:

Disable all interrupts. Use EI/DI only, if different tasks use the same resources; i.e. two tasks both use the external RAM (e6111) – without semaphore handling.

Note: Under normal circumstances, the programmer will not need to disable or enable interrupts – every task will have just the right interrupt level.

Category:	Interrupt handling / assembler instruction
MARC4 opcode:	1A hex
Stack effect:	EXP (—) RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	I_ENABLE flag reset CARRY and BRANCH flags are not affected
X Y registers:	not affected
Bytes used:	1
See also:	EI RTI INT0 .. INT7 SWI0 .. SWI7

Example 1:

```
: DIS-ENABLE-INTERRUPT
  DI          ( disable interrupt flag - CCR: x-x-      )
  5 ROLL      ( re-order EXP stack values.            )
  EI          ( set interrupt flag - CCR: x-xI         )
;
```

Example 2:

Library implementation: of ROLL:

```
: ROLL
  ?DUP
  IF
    CCR@ DI >R      ( save current I-flag setting on RET st.      )
    1+ PICK >R     ( do a PICK, move PICKed value on RET st.      )
    Y@             ( move ptr from Y -> X reg.                    )
    1 M+ X!        ( adjust X reg. pointer                        )
    #DO [+X]@ [+Y]! #LOOP      ( shift data values one down      )
    DROP R>
    R> CCR!        ( restore I-flag setting in CCR                )
  ELSE DROP THEN
;
```

DMAX

“D-max”

Purpose: Leaves the greater of two 8-bit unsigned values on the stack.

Category: Stack operation (double-length) / qFORTH colon definition

Library implementation: (will be improved / changed soon; using D<=, D>)

```

: DMAX 2>R          ( d1 d2 — d1          )
        2DUP        ( d1 — d1 d1          )
        2R@         ( d1 d1 — d1 d1 d2     )
        ROT         ( d1 d1h d1l d2h d2l — d1 d1h d2h d2l d1l )
        2>R         ( d1 d1h d2h d2l d1l — d1 d1h d2h      )
        OVER        ( d1 d1h d2h — d1 d1h d2h d1h        )
        CMP_LT      ( d1 d1h d2h d1h—d1 d1h d2h[B-flag] )
        BRA _DMAX1  ( jump if d2 < d1 in higher nibble     )
        <>          ( d1 d1h d2h — d1 [BRANCH flag]       )
        BRA _DMAX3  ( jump if d2 > d1 in higher nibble     )
        2R@         ( d1 — d1 d2l d1l        )
        <           ( d1 d2l d1l — d1        )
        BRA _DMAX2  ( jump, if d2l < d1l          )
_DMAY3: DROPR      ( skip compare values from RET stack )
        2DROP       ( d1 —                    )
        2R>         ( — d2                    )
        EXIT

_DMAY1: 2DROP      ( skip compare values from EXP stack )
_DMAY2: DROPR      ( skip values from RET stack         )
DROPR      ( — d1                                     )
[ E -2 R 0 ]
;

```

Stack effect:

```

IF d1 > d2
THEN EXP ( d1 d2 — d1 )
ELSE EXP ( d1 d2 — d2 )
RET      ( —          )

```

Stack changes: EXP: 2 elements will be popped from the stack
RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y register: not affected

Bytes used: 30

See also: DMIN MAX MIN

Example:

```
: DMAX-Example  
  ABh 25h DMAX 2DROP ( — A B 2 5 — A B — )  
  ABh ABh DMAX 2DROP ( — A B A B — A B — )  
  25h ABh DMAX 2DROP ( — 2 5 A B — A B — )  
;
```

DMIN

“D–min”

Purpose: Leaves the smaller of two 8-bit unsigned values on the stack.

Category: Stack operation (double-length) / qFORTH colon definition

Library implementation:

```

                ( will be improved / changed soon; using D<=, D> )
: DMIN          2>R          ( d1 d2 — d1 )
                2DUP        ( d1 — d1 d1 )
                2R@         ( d1 d1 — d1 d1 d2 )
                ROT         ( d1 d1h d1l d2h d2l — d1 d1h d2h d2l d1l )
                2>R        ( d1 d1h d2h d2l d1l — d1 d1h d2h )
                OVER        ( d1 d1h d2h — d1 d1h d2h d1h )
                CMP_GT      ( d1 d1h d2h d1h — d1 d1h d2h [BRANCH flag] )
                BRA _DMIN1  ( jump if d2 < d1 in higher nibble )
                <>          ( d1 d1h d2h — d1 [BRANCH flag] )
                BRA _DMIN3  ( jump if d2 > d1 in higher nibble )
                2R@         ( d1 — d1 d2l d1l )
                >           ( d1 d2l d1l — d1 )
                BRA _DMIN2  ( jump, if d2l < d1l )
_DMIN3:        DROPR       ( skip compare values from RET stack )
                2DROP       ( d1 — )
                2R>        ( — d2 )
                EXIT
_DMIN1:        2DROP       ( skip compare values from EXP stack )
_DMIN2:        DROPR       ( skip values from RET stack )
                DROPR       ( — d1 )
                [ E -2 R 0 ]
;

```

Stack effect: IF d1 < d2
THEN EXP (d1 d2 — d1)
ELSE EXP (d1 d2 — d2)
RET (—)

Stack changes: EXP: 2 elements will be popped from the stack
RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 30

See also: DMAX MIN MAX

Example:

: DMIN-example

```
ABh 25h DMIN 2DROP ( — A B 2 5 — 2 5 — )
25h 25h DMIN 2DROP ( — 2 5 2 5 — 2 5 — )
25h ABh DMIN 2DROP ( — 2 5 A B — 2 5 — )
;
```


DNEGATE

“D-negate”

- Purpose:** 2's complement of the top 8-bit value.
- Category:** Arithmetic/logical(double-length) / qFORTH colon definition
- Library implementation:** `: DNEGATE 0 SWAP - (dh dl — dh -dl)
0 ROT -c(dh -dl — -dl -dh)
SWAP ; (-dl -dh — -d)`
- Stack effect:** EXP (d — -d)
RET (—)
- Stack changes:** EXP: not affected
RET: not affected
- Flags:** CARRY and BRANCH flags are affected
- X Y registers:** not affected
- Bytes used:** 8
- See also:** NEGATE

Example:

```
1 CONSTANT true
0 CONSTANT false
( ***** )
: D_Negate
  18h 12h D-      ( 18h - 12h = 06h          )
  12h DNEGATE 18h ( 2's complement of 12h add to 18h )
  D+             ( 18h + [-12h] = 06h ?      )
  D= IF true     ( is the result equal ?      )
  ELSE false     ( 'true' = 1 = YES !        )
  THEN          ( end of test: return.       )
;
```

DO

“DO”

Purpose:

Indicates the start of an iterative loop.

DO is used only within a colon definition and only in a pair with LOOP or +LOOP. The two numbers on top of the stack at the time DO is executed determine the number of times the loop repeats.

The topmost number on the stack is the initial loop index. The next number on the stack is the loop limit. The loop terminates when the loop index is incremented past the boundary between limit-1 and limit (if limit is reached).

A DO loop is always executed at least once, even if the loop index initially exceeds the limit.

Category: Control structure / qFORTH macro

Library implementation: CODE DO (EXP: limit index —)
 2>R (RET: — u|limit|index)
 _\$DO: [E -2 R 1] (DO LOOP backpatch label)
 END-CODE

Stack effect: EXP (limit index —)
 RET (— u|limit|index)

Stack changes: EXP: 2 elements will be popped from the stack
 RET: 1 element (2 nibbles) will be pushed onto the stack

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: LOOP #DO #LOOP ?DO +LOOP ?LEAVE -?LEAVE

Example:

```
: DoLoop
  6 2          ( limit and start on the stack.      )
  DO          (                                     )
    I         ( copy the index from the return stack. )
    TestPort1 OUT ( write 2, 3, 4, 5 to the 'TestPort1'. )
  LOOP       ( loop until limit = index.           )
            (                                     )
  9 2        (                                     )
  DO         (                                     )
    I         (                                     )
    TestPort1 OUT ( write 2, 4, 6, 8 to the 'TestPort1'. )
  2 +LOOP    (                                     )
;           (                                     )
```

DROP

“DROP”

Purpose:

Removes one 4-bit value from top of the expression stack, i.e.: decrements the expression stack pointer.

Category: Stack operation (single-length) / assembler instruction

MARC4 opcode: 2E hex

Stack effect: EXP (n1 —)
RET (—)

Stack changes: EXP: 1 element is popped from the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: 2DROP 3DROP

Example:

```
: DROP_Example
  19H      (      — 1 9      )
  11H      (   1 9 — 1 9 1 1   )
  DROP     ( 1 9 1 1 — 1 9 1   )
  2DROP    (   1 9 1 — 1       )
  19h      (      1 — 1 1 9    )
  3DROP    (   1 1 9 —         )
;
```

DROPR

“DROP-R”

Purpose:

Decrements the return stack pointer. Removes one entry (= 3 nibbles) from the return stack.

Category: Assembler instruction

MARC4 opcode: 2F hex

Stack effect: EXP (—)
RET (x|x|x —)

Stack changes: EXP: not affected
RET: 1 element (12-bits) is popped from the return stack

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: >R I R> 2>R 2R> 3>R 3R> EXIT

Example 1:

```
: DROPR_Example
  1 2 3 3>R          ( RET: — 1|2|3          )
  DROPR             ( RET: 1|2|3 —          )
;
```

Example 2:

Library implementation: of #LOOP :

```
( #LOOP – Macro _____ )
( Purpose: #LOOP is used to terminate a #DO loop. )
( )
( On each iteration of a #DO loop, #LOOP decrements the )
( loop index on the return stack. It then compares the index )
( to zero and determine whether the loop should terminate. )
( If the new index is decremented to zero, the loop is )
( terminated and the loop index is discarded from the return )
( stack. Otherwise, control jumps back to the point just after )
( the corresponding begin of the #DO macro. )

\ IF Index-1 > 0
\ THEN RET      ( u|u|Index — u|u|Index-1  )
\ ELSE RET      ( u|u|Index —              )

CODE #LOOP DECR ( RET: u|u|Index — u|u|Index )
  BRA _$#DO     ( IF Index > 0, Loop again )
  _$LOOP: DROPR ( forget Index on RET stack )
  [ E 0 R -1 ]
END-CODE
```


DTABLE@

“D-TABLE-fetch”

Purpose: Fetches an 8-bit constant from a ROMCONST array, whereby the 12-bit ROM address and the 4-bit index are on the EXP stack.

Category: Memory operation (double-length) / qFORTH colon definition

Library implementation:

```
DTABLE@ M+      ( ROMh ROMm ROMl ind — ROMh ROMm' ROMl'      )
IF              ( on overflow propagate CARRY                )
  ROT 1+ <ROT    ( ROMh ROMm' ROMl' — ROMh' ROMm' ROMl'      )
THEN
3>R             ( move TABLE address to RET stack          )
TABLE          ( — consthigh constlow                       )
               ( TABLE returns directly to the CALLer during microc. exec. )
[ E -2 R 0 ]   ( therefore 'EXIT' is not necessary           )
;;
```

Stack effect: EXP (ROMh ROMm ROMl index — consth constl)
RET (—)

Stack changes: EXP: 2 elements will be popped from the stack
RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 14

See also: TABLE ROMByte@ ROMCONST

Example:

```
ROMCONST DigitTable    10h , 1 , 2 , 3 , 4 , 45h , 6 , 7 , 8 ,
                        9 , Ah , Bh , Ch , Dh , Eh , 0Fh ,

( ***** )
: D_Table@
  0 0 1 ROMByte@      ( fetch byte at address 001h : 0Fh = SLEEP )
  2DROP              ( and delete it. )

                        ( sixth byte of the table: )
DigitTable 5         ( put address and index on the stack. )
DTABLE@ 2DROP       ( fetch and delete the value : 45h . )
                        ( second byte of the table: )
DigitTable 1         ( put address and index on the stack. )
DTABLE@ 2DROP       ( fetch and delete the value : 01h . )
                        ( first byte of the table and min. index : )
DigitTable 0         ( put address and index on the stack. )
DTABLE@ 2DROP       ( fetch and delete the value : 10h . )
                        ( last byte of the table and max. index : )
DigitTable Fh       ( put address and index on the stack. )
DTABLE@ 2DROP       ( fetch and delete the value : 0Fh. )
;
```


Example:

```
2VARIABLE Supra
VARIABLE 1Supra
: D_Toggle
  0 0 Supra 2!      ( reset in the RAM two nibbles to 00h.      )
  FFh Supra DTOGGLE ( 00h XOR FFh = FFh                      )
                    ( flags: no BRANCH                      )
  FFh Supra DTOGGLE ( 00h XOR FFh = 00h                      )
                    ( flags: BRANCH                        )
  AAh Supra 2!      ( set the two nibbles to AAh.                      )
  55h Supra DTOGGLE ( 1010 1010 XOR 0101 0101 = 1111 1111 )
                    ( flags: no BRANCH                      )
  5 1Supra !        ( set in the RAM one nibble to 0101.      )
  3 1Supra TOGGLE   ( truth table: 0101 XOR 0011 = 0110      )
                    ( flags: no BRANCH                      )
  Fh 1Supra !       ( set in the RAM one nibble to Fh.          )
  Fh 1Supra TOGGLE  ( 1111 XOR 1111 = 0000                      )
                    ( flags: BRANCH                        )
  Fh 1Supra TOGGLE  ( 0000 XOR 1111 = 1111                      )
;                   ( flags: no BRANCH                      )
```

DUP

“Doop”

Purpose:

Duplicate the 4-bit value on top of the stack.

Category: Stack operation (single-length) / assembler instruction

MARC4 opcode: 2D hex

Stack effect: EXP (n1 — n1 n1)
RET (—)

Stack changes: EXP: 1 element is pushed onto the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: 2DUP 3DUP DROP

Example:

```
: ONE-DUP
  9          (  — 9          )
  5          ( 9  — 9 5      )
  DUP       ( 9 5 — 9 5 5    )
  3DROP     ( 9 5 5 —       )
;
```

EI

“Enable-Interrupt” or “E-I”

Purpose:

Sets the INTERRUPT_ENABLE flag in the condition code register. RTI (return-from-interrupt) at end of \$RESET or an INTx section will set the interrupt flag too. Use EI/DI only, if different tasks use the same resources; i.e. two tasks both use the external RAM (e6111) – without semaphore handling.

Note: Under normal circumstances, the programmer will not need to disable or enable interrupts – every task will have just the right interrupt level.

Category: Interrupt handling / qFORTH macro

Library implementation: CODE EI
LIT_1 CCR! (set I_ENABLE flag)
END-CODE

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag reset
BRANCH flag reset
I_ENABLE flag set

X Y registers: not affected

Bytes used: 2

See also: DI RTI INTO .. INT7 SWI0 .. SWI7

Example:

```
: InterruptFlagRe_Set
  DI          ( disable the interrupt flag – CCR: x-x- )
  5 ROLL     ( re-order EXP stack values.           )
  EI          ( enable the interrupt flag – CCR: x-xI )
;
```


ELSE

“ELSE”

Purpose:

Part of the IF ... ELSE ... THEN control structure. ELSE, like IF and THEN, may be used only within a colon definition. Its use is optional. ELSE executes after the TRUE part following the IF construct. If the condition is true, ELSE forces execution to skip over the following FALSE part and resumes execution following the THEN construct. If the condition is false, the FALSE block after the ELSE instruction will be executed.

Category: Control structure /qFORTH macro

Library implementation:

```

CODE ELSE SET_BCF          ( set BRANCH&CARRY flag, FORCE jump      )
      (S)BRA _$THEN        ( to end of IF statement                )
      _$ELSE: [ E 0 R 0 ]
END-CODE

```

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 2 – 3

See also: IF THEN

Example:

```
: IfElseThen
  1 2 <=          ( is 1 <= 2 ?           )
  IF             ( yes, so the If block will be executed. )
    1           ( a 1 will be pushed onto the stack.     )
  ELSE          (                               )
    0           ( true => no execution of the ELSE block. )
  THEN          (                               )

  1 2 >          ( is 1 > 2 ?           )
  IF             (                               )
    DROP 0      ( false: nothing will be executed.       )
  THEN          (                               )
              (                               )

  1 2 >          ( is 1 > 2 ?           )
  IF             (                               )
    0           ( not true => no execution.              )
  ELSE          ( in this case, the                      )
    1           ( ELSE block will be executed           )
  THEN 2DROP    ( the results from the expression stack. )
;

```

END-CODE

“END-CODE”

Purpose:

Terminates an 'in-line' CODE definition.

Category: Predefined data structure

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 0

See also: CODE <-> ':' and ';'

Example 1:

```
3 ARRAY ReceiveData    ( 12-bit data item          )
(----- CODE to shift right a 12-bit data word  )
CODE ShiftRDBits
  ReceiveData Y!
  CLR_BCF          ( clear the CARRY for first shift )
  [Y]@ ROR [Y]!
  [+Y]@ ROR [Y]!  ( rotate thru CARRY              )
  [+Y]@ ROR [Y]!
END-CODE
```

: Example 2:

```
ReceiveData Y!          ( start address to Y register.    )
5 [Y]! Ah [+Y]! 0 [+Y]!
ShiftRDBits             ( shift 'ReceiveData' 1 bit right )
;
```

ENDCASE

“End-CASE”

Purpose:

Terminates a CASE ... OF ... ENDOF ... ENDCASE structure.

When it executes, ENDCASE drops the 4-bit CASE index value if it does not match any of the OF comparison values.

The 'OTHERWISE' case may be handled by a sequence placed between the last ENDOF and ENDCASE. Note however that the index value must be preserved across this sequence so that ENDCASE can drop it.

Category: Control structure /qFORTH macro

Library implementation: CODE ENDCASE
 DROP (n —)
 _\$ENDCASE: [E -1 R 0]
 END-CODE

Stack effect: EXP (n —) (if no match)
 EXP (—) (if matched, then not executed)
 RET (—)

Stack changes: EXP: 1 element will be popped from the stack
 RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: CASE OF ENDOF

Example:

```
5 CONSTANT Keyboard
1 CONSTANT Port1

: ONE 1 Port1 OUT ;      ( write 1 to the 'Port1'      )
: TWO 2 Port1 OUT ;      ( write 2 to the 'Port1'      )
: THREE 3 Port1 OUT ;    ( write 3 to the 'Port1'      )
: ERROR DUP Port1 OUT ; ( dump wrong input to Port1 )
( duplicate value for the following ENDCASE; it drops one n. )

: CASE-Example
  KeyBoard IN           ( request 1-digit keyboard input      )
  CASE                  ( depending of the input value,        )
  1 OF ONE  ENDOF       ( one of these words will be acti-    )
  2 OF TWO  ENDOF       ( vated.                               )
  3 OF THREE ENDOF      (                                     )
  ERROR              ( otherwise ...                          )
ENDCASE              ( n —                                     )
;
```

ENDOF

“End-OF”

Purpose:

Part of the OF ... ENDOF structure used within CASE ... ENDCASE.

When an OF comparison value matches the CASE index value, ENDOF transfers control to the word following ENDCASE. If there was no match, control proceeds with the word following ENDOF.

Category:

Control structure /qFORTH macro

Library implementation:

```
CODE ENDOF SET_BCF
          (S)BRA _$ENDCASE
          _$ENDOF: [ E O R 0 ]
END-CODE
```

Stack effect:

EXP (—)
RET (—)

Stack changes:

EXP: not affected
RET: not affected

Flags:

CARRY flag Set
BRANCH flag Set

X Y registers:

not affected

Bytes used:

2 – 3

See also:

ENDCASE CASE OF

Example:

```
5 CONSTANT Keyboard
1 CONSTANT Port1

: ONE 1 Port1 OUT ;      ( write 1 to the 'Port1'      )
: TWO 2 Port1 OUT ;      ( write 2 to the 'Port1'      )
: THREE 3 Port1 OUT ;    ( write 3 to the 'Port1'      )
: ERROR DUP Port1 OUT ;  ( dump wrong input to the Port1  )
( duplicate value for the following ENDCASE; it drops one n. )

: CASE-Example
  KeyBoard IN            ( request 1-digit keyboard input  )
  CASE                   ( depending of the input value,      )
    1 OF ONE ENDOF       ( one of these words will be acti-   )
    2 OF TWO ENDOF       ( vated.                          )
    3 OF THREE ENDOF     (                               )
    ERROR                ( otherwise ...                    )
  ENDCASE                ( n —                               )
;
```


ERASE

“ERASE”

Purpose:

Resets n digits in a block of memory (RAM) to zero. Where as n is less than 16; if n is zero, then 16 nibbles of RAM will be set to 0.

Category: Memory operation (multiple-length) / qFORTH colon definition

Library implementation: (ERASE – qFORTH word—————)
 (purpose: sets all digits in a block of memory to zero.)
 : ERASE <ROT Y! (addr count — count)
 0 [Y]! 1– (count — count–1)
 #DO
 0 [+Y]!
 #LOOP

;

Stack effect: EXP (addr n —)
 RET (—)

Stack changes: EXP: 3 elements are popped from the stack
 RET: not affected

Flags: CARRY flag not affected
 BRANCH flag Reset

X Y registers: The contents of the Y register will be changed.

Bytes used: 14

See also: FILL CONSTANT ARRAY

Example:

```
6  CONSTANT #Nibbles           ( #_of_valid_bits           )
0  CONSTANT #16
3  CONSTANT TwoLgth
#Nibbles ARRAY RamData        ( nibble array with 6 elements
                               and index from [0]..[5]           )
16 ARRAY ShortArray           ( index from [0] .. [15]           )
TwoLgth 2ARRAY TwoArray       ( this array includes bytes           )
20 2LARRAY TwoLongArray
: ClearArrays
  RamData #Nibbles ERASE      ( initialize the data array       )
  ShortArray #16 ERASE        ( 'delete' 16 nibbles             )
  TwoArray TwoLgth*2 ERASE    ( set whole array to 0.           )
  TwoLongArray [4] 8 ERASE    ( set byte [4] to [7] to 0.       )
  ( for setting whole arrays with more than 16 nibbles to 0,       )
  ( a special routine is required; or activate ERASE twice !       )
;
```

EXIT

“EXIT”

Purpose:

Exits from the current colon definition.

EXIT may be used in any of the following control structures:

```
BEGIN ... REPEAT,  
IF ... THEN,  
CASE ... ENDCASE
```

Note:EXIT may not be used inside of a DO loop !

For ending a colon definition, ';' will be translated by the compiler to the EXIT instruction.

Category:	Control structure / assembler instruction
MARC4 opcode:	25 hex
Stack effect:	EXP (—) RET (oldPC —)
Stack changes:	EXP: not affected RET: the return-address (3 nibbles) is popped from the return stack into the program counter.
Flags:	not affected
X Y registers:	not affected
Bytes used:	1
See also:	?LEAVE -?LEAVE ; RTI

Example 1:

```
: EXIT-Example
  6 2 3          ( — 6 2 3          )
  CMP_NE        ( 6 2 3 — 6 2      )
  IF EXIT       ( if <> then EXIT else DROP TOS )
  ELSE DROP     ( 6 2 — 6          )
  THEN
;

```

Example 2:

```
CODE Leave
DROPR EXIT      ( exits any DO .. LOOP      )
END-CODE

: Horizontal?   ( example for leave DO .. LOOP )
  DO           ( col row —                )
  DUP I Pos@ =
  IF DROP 0 Leave [ R 0 ] THEN
  LOOP
;

```

FILL

“FILL”

Purpose:

Fill a block of memory (n1 nibbles; 0 <= n1 <= Fh) with a specified digit (n2).

Category:

Memory operation (multiple-length) / qFORTH colon definition

Library implementation: : FILL

```

2SWAP ( addr count n — count n addr )
Y! DUP [Y]! ( count n addr — count n )
SWAP 1- ( count n — n count-1 )
#DO
DUP [+Y]!
#LOOP DROP

```

;

Stack effect::

```

EXP ( addr n1 n2 — )
RET ( — )

```

Stack changes:

EXP: 4 elements are popped from the stack
RET: not affected

Flags:

CARRY flag not affected
BRANCH flag reset

X Y registers:

The contents of the Y register will be changed.

Bytes used:

16 + '2SWAP'

See also:

ERASE

Example:

```
( ***** )
8 CONSTANT Size
Size ARRAY Digits
: Fill_Example
  Digits Size 3      ( — address count data          )
  FILL              ( fill array digits with 3        )
  34h Fh 5 FILL     ( RAM: 34h...42h – 15nibbles – will be 5. )
  44h 0 6 FILL      ( RAM: 44h...53h – 16nibbles – will be 6. )
;
```

I**R@**

“I” “R–Fetch”

Purpose:

Leaves (copies) the current #DO or DO loop index on the stack.

If not used within a DO ... [+]LOOP, or #DO ... #LOOP, the value returned by I or R@ is undefined.

Category: Stack operation (single-length) / assembler instruction**Library implementation:** CODE R@ I END-CODE (macro of 'R@')**MARC4 opcode:** 23 hex**Stack effect:**
EXP (— index)
RET (u|limit/u|index — u|limit/u|index)**Stack changes:**
EXP: the current loop index will be pushed onto the stack
RET: not affected**Flags:** not affected**X Y registers:** not affected**Bytes used:** 1**See also:** J DO [+]LOOP #DO ... #LOOP

Example 1:

```
1 CONSTANT Port1
( ***** )
: HASH-DO-LOOP
  14 #DO          ( loop 14 times )
    I Port1 OUT   ( write data to 'Port1': E, D, C, ..., 1. )
  #LOOP          ( repeat the loop. )
;
```

Example 2:

```
: Error          ( show program counter, where CPU fails. )
  3R@
  3 #DO          ( write address to port 1, 2 and 3 )
  I OUT [ E 0 ]
  #LOOP [ E 0 ] ( suppress compiler warnings. )
;

: RFetch
  1 3 3 3 5      ( compare all these values with 3 )
  3 >R          ( move 3 to return stack )
  R@ < IF Error THEN ( copy 3 from RET several times )
  R@ < IF Error THEN ( 'Error' should never be called. )
  R@ > IF Error THEN ( )
  R@ <> IF Error THEN ( )
  R> >= IF Error THEN ( return stack gets original state )
;
```


IF

“IF”

Purpose:

Begins a IF ... ELSE ... THEN or IF ... THEN control structure. When IF is executed, the BRANCH flag in the condition code register (CCR) determines the direction of the conditional branch. If the BRANCH flag is TRUE (set), the words between the IF and ELSE (or IF and THEN if no ELSE was compiled) are executed. If the BRANCH flag is false (= 0), and an ELSE clause exists, then the words between ELSE and THEN are executed. In either case, subsequent execution continues just after THEN.

Category: Control structure /qFORTH macro

Library implementation:

```
CODE IF TOG_BF      ( complement B-Flag, FORCE jump if false.      )
      (S)BRA _ELSE  ( to _ELSE / _THEN label                    )
END-CODE
```

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag not affected
BRANCH flag = NOT BRANCH flag

X Y registers: not affected.

Bytes used: 1 – 3

See also: THEN ELSE

Example:

```
: IfElseThen
  1 2 <=          ( is 1 <= 2 ?           )
  IF             ( yes, so the If block will be executed. )
    1           ( a 1 will be pushed onto the stack.     )
  ELSE          (                               )
    0           ( true => no execution of the ELSE block. )
  THEN          (                               )

  1 2 >          ( is 1 > 2 ?           )
  IF             (                               )
    DROP 0      ( false: nothing will be executed.       )
  THEN          (                               )
              (                               )

  1 2 >          ( is 1 > 2 ?           )
  IF             (                               )
    0           ( not true => no execution.              )
  ELSE          ( in this case, the                      )
    1           ( ELSE block will be executed           )
  THEN 2DROP    ( the results from the expression stack. )
;

```

IN

“IN”

Purpose:

Read data from one of the 16 possible 4-bit I/O ports.

NOTE: Before changing the direction of a bidirectional port from output to input, first a value of 'Fh' has to be written to that port. After power-on reset all bidirectional ports are switched to input.

Category: Stack operation / assembler instruction

MARC4 opcode: 1B hex

Stack effect: EXP (port — data)
RET (—)

Stack changes: EXP: The port address is pulled from the stack; the 'data' is pushed onto the stack.
RET: not affected

Flags: CARRY flag not affected
BRANCH flag set, if (TOS = 0)

X Y registers: not affected

Bytes used: 1

See also: OUT

Example 1:

```
1 CONSTANT Port1
: Countdown
  15 #DO          ( 15 iterations          )
    I            ( copy index from return stack. )
    Port1 OUT    ( Index is output to the 'Port1' )
  #LOOP
;
```

Example 2:

```
: ReadPort      ( port — data          )
  Fh OVER OUT   ( port — port Fh — p Fh p — p )
  IN            ( port — nibble        )
;
```

INDEX

“INDEX”

Purpose:

The qFORTH word INDEX performs RAM address computations during runtime to give the programmer the ability to access any element of an ARRAY, 2ARRAY, etc. ...

Category: Predefined data structure /qFORTH macro

Library implementation: Different routines are available for ARRAY, 2ARRAY, ...

Stack effect: EXP (n d — d) for ARRAY, 2ARRAY
EXP (d d — d) for LARRAY, 2LARRAY
RET (—)

Stack changes: EXP: 1/2 element(s) will be popped from the stack
RET: not affected

Flags: CARRY flag reset
BRANCH flag = CARRY flag

X Y registers: not affected

Bytes used: uses 'D+' or 'M+'

See also: @ ! ARRAY 2ARRAY LARRAY 2LARRAY

Example:

```
10 ARRAY 10Nibbles
( ***** )
: IndexExample      ( write 1 .. 10 into the array [0] .. [9] )
  10 #DO
    I                ( copy values for writing: A, 9, 8, ... 1 )
    I 1- 10Nibbles INDEX !
  #LOOP             ( subtract 1 from index for [9] .. [0] )
;
```

'INT0 ... INT7'

“Int-Zero” ... “Int-Seven”

Purpose:

The interrupt routines can be activated by external hardware or by internal software interrupts (SWI). These predefined HARDWARE/SOFTWARE interrupt service routines are placed by the compiler at the following fixed addresses:

Routine	Address	Length	Routine	Address
INT0\$040/64	64	INT1	\$080/128	64
INT2\$0C0/192	64	INT3	\$100/256	64
INT4\$140/320	64	INT5	\$180/384	64
INT6\$1C0/448	32	INT7	\$1E0/480	unlimited

During runtime the PC will be set by the interrupt logic to the addresses determined by the compiler. If an interrupt routine gets too long, then the compiler will not be able to place this routine in the corresponding segment. To avoid this problem, it may be necessary to divide the routine and define parts of it as new colon definitions, which will be placed at other free ROM gaps. For more informations about interrupts, please have a look in the other manuals !

- Category:** Interrupt handling
- Stack effect:** EXP (—)
RET (— [old PC]) on runtime
- Stack changes:** EXP: not affected
RET: not affected
- Flags:** Will be saved on entry; the @ (fetch) and ! (store) instructions (X@ Y@ CCR@...) will be inserted in the opcode automatically by the compiler – if necessary.
- X Y registers:** not affected
- Bytes used:** 0
- See also:** SWI0 .. SWI7 RTI \$AUTOSLEEP

Example:

```
: INT5
  CCR@           ( these instructions will be inserted by the )
  Y@ X@         ( compiler automatically ! )
  IncTime       ( activate the time-increment module every )
  X! Y! CCR!    ( 125 msec. / store the reg-data back. )
; AT 180h      ( an RTI will occur in the opcode. )
```


J

“J”

Purpose:

Leaves the current loop index of the next outer DO or #DO loop on the stack, when used within a nested loop.

If not used within two DO ... [+]LOOP, or #DO ... #LOOP, the value returned by J is undefined.

Category: Stack operation (single-length) /qFORTH macro

Library implementation:

```

CODE J 2R>      ( — limit I          )
      I         ( limit I — limit I J )
      <ROT      ( limit I J — J limit I )
      2>R       ( J limit I — J      )
END-CODE
    
```

Stack effect:

```

EXP      ( — J          )
RET      ( u|limit|J u|limit|I — u|limit|J u|limit|I )
    
```

Stack changes: EXP: 1 element will be pushed onto the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 6

See also: I DO ... [+]LOOP #DO ... #LOOP

Example:

1 CONSTANT Port1

(*****)

: HASH-DO-LOOP

```
6 #DO          ( loop 6 times )
  I Port1 OUT  ( write to 'Port1': 6, 5, 4, ..., 1. )
  2 #DO        ( loop 2 times in the loop )
    J Port1 OUT ( get index from outer loop. )
    #LOOP      ( repeat the loop 2 times. )
#LOOP          ( repeat the loop 6 times. )
; ( Port1 – result: 6, 6, 6, 5, 5, 5, 4, ... 2, 1, 1, 1. )
```

LARRAY

“Long-ARRAY”

Purpose:

Allocates RAM space for storage of a long single-length (4-bit) array, using an 8-bit array index value. Therefore the number of 4-bit array elements can be greater than 16.

The qFORTH syntax is as follows:

```
<number> LARRAY <identifier> [ AT <RAM-Addr> ]
```

At compile time, LARRAY adds <name> to the dictionary and ALLOTs memory for storage of <number> single-length values. At execution time, <name> leaves the RAM start address of the parameter field (<name> [0]) on the expression stack.

The storage ALLOTed by an LARRAY is not initialized; see ERASE.

Category:	Predefined data structure
Stack effect:	EXP (— n) a fetch (@) on runtime RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	0
See also:	2ARRAY ARRAY 2LARRAY Index ERASE

Example:

60 2CONSTANT LongLgth

12 ARRAY ShortArray (normal array example.)

LongLgth LARRAY LongArray AT 68h
(array, larger than 16 nibbles; at specific RAM address.)

: ArrayExample

ShortArray 12 ERASE (set all 12 nibbles to 0)

LongArray 15 ERASE (set nibble [0] to [15] to 0)

;

LOOP

“LOOP”

Purpose:

LOOP may be used to terminate either DO or ?DO loops.

On each iteration of a DO loop, LOOP increments the loop index. It then compares the index to the loop limit to determine whether the loop should terminate. If the new index is incremented across the boundary between limit-1 and limit, the loop is terminated and the loop control parameters are discarded. Otherwise, control jumps back to the point just after the corresponding DO.

Category: Control structure /qFORTH macro

Library implementation:

```

CODE LOOP 2R>      ( — limit index          )
              INC   (limit index — limit index' )
              OVER  (limit index' — limit index' limit )
              CMP_LT (limit index' limit — limit index' )
              2>R   (limit index' —          )
              BRA_$DO(IF Index < limit, loop again )
_$LOOP: DROPR      (forget limit, index on RET stack )
[ E 0 R -1 ]
END-CODE

```

Stack effect:

```

EXP          ( — )
IF Index+1 < Limit
THEN RET( u|Limit|Index — u|Limit|Index+1 )
ELSE RET ( u|Limit|Index — )

```

Stack changes:

```

EXP: not affected
RET: IF Index+1 = Limit THEN
1 element (3 nibbles) will be popped from the stack.

```

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 9

See also: DO #DO #LOOP +LOOP

Example:

```
: DoLoop
  6 2          ( limit and start on the stack.      )
  DO          (                                     )
    I         ( copy the index from the return stack. )
    TestPort1 OUT ( write 2, 3, 4, 5 to the 'TestPort1'. )
  LOOP       ( loop until limit = index.           )
            (                                     )
  92         (                                     )
            (                                     )
  I         (                                     )
  TestPort1 OUT ( write 2, 4, 6, 8 to the 'TestPort1'. )
  2 +LOOP    (                                     )
;           (                                     )
```

M+ M-

“M-plus” “M-minus”

Purpose:

M+ adds the digit (4 bit) on top of the data stack to the 8-bit value below that. M- subtracts a nibble on the data stack from the 8-bit value below that.

Category: Arithmetic/logical (single-double-length) /
FORTH colon definition

Library implementation: : M+
+ SWAP 0 +c SWAP
;

: M-
- SWAP 0 -c SWAP
;

Stack effect: EXP (d1 n — d2)
RET (—)

Stack changes: EXP: 1 element is popped from the stack.
RET: not affected

Flags: CARRY flag set on arithmetic over-/underflow
BRANCH flag = CARRY flag

X Y registers: not affected

Bytes used: 5

See also: D- D+ D2* D2/

Example:

```
( ***** )  
: MPlusMin  
  13h 5 M+      ( 13h + 5 = 18h  CARRY: %  BRANCH: %  )  
    5 M-        ( 18h - 5 = 13h  CARRY: %  BRANCH: %  )  
  2DROP        ( two nibbles. )  
  13h 15 M+    ( 13h +15 = 22h  CARRY: %  BRANCH: %  )  
  2DROP        ( two nibbles. )  
  FCh 9 M+     ( FCh + 9 = [105h]05h CARRY: 1  BRANCH: 1 )  
    9 M-       ( 05h - 9 = FCh  CARRY: 1  BRANCH: 1 )  
  2DROP        ( two nibbles. )  
;
```


MAX

“MAX”

Purpose:

Leaves the greater of two 4-bit values on the stack.

Category: Comparison (single-length) / qFORTH colon definition

Library implementation:

```

: MAX      OVER      ( n1 n2 — n1 n2 n1      )
      CMP_LT      ( n1 n2 n1 — n1 n2 [BRANCH flag] )
      BRA      _LESS ( jump if n1 < n2      )
      SWAP      ( n1 n2 — n2 n1      )
      _LESS:    DROP  ( leave max number on stack )
                [ E -1 R 0 ]
;
    
```

Stack effect:

```

IF n1 > n2
THEN EXP      ( n1 n2 — n1  )
ELSE EXP      ( n1 n2 — n2  )
RET           ( —          )
    
```

Stack changes: EXP: 1 element will be popped from the stack
RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y register: not affected

Bytes used: 7

See also: MIN DMAX DMIN

Example:

```
( ***** )
: Min_Max_
  Ah 2 MIN DROP      ( — A 2 — 2 — flags: CARRY — )
  2 2 MIN DROP      ( — 2 2 — 2 — flags: — — )
  2 Ah MIN DROP     ( — 2 A — 2 — flags: — BRANCH )

  Ah 2 MAX DROP     ( — A 2 — A — flags: CARRY BRANCH )
  Ah Ah MAX DROP   ( — A A — A — flags: — — )
  2 Ah MAX DROP    ( — 2 A — A — flags: — — )
;
```

MIN

“MIN”

Purpose:

Leaves the smaller of two 4-bit values on the stack.

Category: Comparison (single-length) / qFORTH colon definition

Library implementation:

```

: MIN      OVER      ( n1 n2 — n1 n2 n1 )
            CMP_GT    ( n1 n2 n1 — n1 n2 [BRANCH flag] )
            BRA      _GREAT ( jump if n1 > n2 )
            SWAP     ( n1 n2 — n2 n1 )
            _GREAT:  DROP ( leave max number on stack )
                    [ E -1 R 0 ]
;

```

Stack effect:

```

IF n1 < n2
THEN EXP      ( n1 n2 — n1 )
ELSE EXP      ( n1 n2 — n2 )
RET           ( — )

```

Stack changes: EXP: 1 element will be popped from the stack
RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 7

See also: MAX DMIN DMAX

Example:

```
( ***** )  
: Min_Max_  
  Ah 2  MIN DROP      ( — A 2 — 2 — flags: CARRY - )  
  2 2  MIN DROP      ( — 2 2 — 2 — flags: - - )  
  2 Ah MIN DROP      ( — 2 A — 2 — flags: - BRANCH )  
  
  Ah 2  MAX DROP      ( — A 2 — A — flags: CARRY BRANCH )  
  Ah Ah MAX DROP     ( — A A — A — flags: - - )  
  2 Ah MAX DROP     ( — 2 A — A — flags: - - )  
;
```

MOVE

“MOVE”

Purpose:

Copies an array of digits from one memory location to another.

Number of nibbles n : 2 <= n <= Fh (0 moves 16 nibbles)

The digit at the LOWEST memory location is copied first [unlike 'MOVE>']. This allows the transfer of data between overlapping memory arrays from a higher to a lower address (see the examples!).

Category: Memory operation (multiple-length) / qFORTH colon definition

Library implementation: : MOVE
 Y! X! (length SrcAddr DestAddr —)
 [X]@ [Y]! (Move 1st element)
 1- #DO [+X]@ [+Y]! #LOOP
 ;

Stack effect: EXP (n from to —)
 RET (—)

Stack changes: EXP: 5 elements are popped from the stack
 RET: not affected

Flags: CARRY flag not affected
 BRANCH flag reset

X Y registers: Both the X and Y index registers will be affected.

Bytes used: 14

See also: MOVE> FILL ERASE

Example:

```
16 ARRAY A16 AT 40h
: MoveArray
  A16 [15] Y! 0          ( write 0 ... Fh to RAM; start at 40h      )
  #DO I 1- [Y-]!
  #LOOP                 ( repeat 16 times: copy index to RAM.    )
  4 A16 A16 [12]        ( with start address and pre increm.:o.k.      )
  MOVE                  ( 4 nibbles are moved 12 nibb's backwards )
  ( same result, as with: '4 A16 [3] A16 [15] MOVE>' !      )

  A16 [15] Y! 0          ( write 0 ... Fh to RAM; start at 40h      )
  #DO I 1- [Y-]!
  #LOOP                 ( repeat 16 times: copy index to RAM.    )
  4 A16 [7] A16 [5]     ( with start address and pre increm.: o.k.      )
  MOVE                  ( 4 nibbles are moved 2 nibbles backwards )

  A16 [15] Y! 0          ( write 0 ... Fh to RAM; start at 40h      )
  #DO I 1- [Y-]!
  #LOOP                 ( repeat 16 times: copy index to RAM.    )
  4 A16 [5] A16 [7]     ( with start address and pre increm.:      )
  MOVE                  ( ERROR: overwriting of the source array. )
;
```

MOVE>

“MOVE-greater”

Purpose:

Copies an array of digits from one memory location to another.

Number of nibbles n : $2 \leq n \leq Fh$ (0 moves 16 nibbles)

The digit at the HIGHEST memory location is copied first [unlike 'MOVE']. This allows the transfer of data between overlapping memory arrays from a LOWER to a HIGHER address (See the examples!).

Category: Memory operation (multiple-length) / qFORTH colon definition

Library implementation: `: MOVE>
 Y! X! (length SrcAddr DestAddr —)
 #DO [X-]@ [Y-]! #LOOP
 ;`

Stack effect: EXP (n from to —)
 RET (—)

Stack changes: EXP: 5 elements are popped from the stack
 RET: not affected

Flags: CARRY flag not affected
 BRANCH flag reset

X Y register: Both the X and Y index registers will be affected.

Bytes used: 10

See also: MOVE FILL ERASE

Example:

```
16 ARRAY A16 AT 40h
: MoveArray
  ( an array is used: start at 40h; end at 4Fh, with 0...Fh. )
  A16 [15] Y! 0      ( write 0 ... Fh to RAM; start at 40h )
  #DO I 1- [Y-]!
  #LOOP             ( repeat 16 times: copy index to RAM. )
  4 A16 [6] A16 [9] ( with end address / post decrement: o.k. )
  MOVE>            ( 4 nibbles are moved 3 nibbles forward. )

  A16 [15] Y! 0      ( write 0 ... Fh to RAM; start at 40h )
  #DO I 1- [Y-]!
  #LOOP             ( repeat 16 times: copy index to RAM. )
  4 A16 [9] A16 [6] ( with end address and post decrement: )
  MOVE>            ( ERROR: overwriting of the source array. )

  A16 [15] Y! 0      ( write 0 ... Fh to RAM; start at 40h )
  #DO I 1- [Y-]!
  #LOOP             ( repeat 16 times: copy index to RAM. )
  4 A16 [3] A16 [15] ( with end address and post decrement: )
  MOVE>            ( 4 nibbles are moved 12 nibbles forward. )
  ( same result, as with: '4 A16 A16 [12] MOVE' ! )
;
```


NEGATE

“NEGATE”

Purpose:

2's complement of the TOS 4-bit value.

Category: Arithmetic/logical(single-length) /qFORTH macro

Library implementation: CODE NEGATE
NOT 1+ (n — -n)
END-CODE

Stack effect: EXP (n1 — -n1)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag not affected
BRANCH flag set, if (TOS = 0)

X Y registers: not affected

Bytes used: 2

See also: DNEGATE NOT

Example:

```
: true 1 ;           ( this is only a simple example for 'true'   )
: false 0 ;
( ***** )
: N_egate
  8 2 -             ( 8 - 2 = 6 )
  2 NEGATE 8       ( 2's complement of 2 add to 8 )
  +                ( 8 + [-2] = 6 ? )
  = IF true        ( is the result equal ? )
  ELSE false       ( 'true' = 1 = YES ! )
  THEN DROP       ( end of test: drop the result. )
;
```

NOP

“NOP”

Purpose:

No operation; one instruction cycle of time is used. Useful, if the processor has to wait a short time for an external device or interrupt.

Category: Assembler instruction

MARC4 opcode: 7C hex

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: \$AUTOSLEEP

Example 1:

Library implementation: of SWI0 :

```
CODE SWI0
  0 1 SWI NOP      ( activate the base task          )
END-CODE          ( the NOP gives time for task ..  )
                  ( switching to the interrupt control logic . )
```

Example 2:

```
: Delay          ( n — [wait 4+n*7 cycles, ..]      )
#DO             ( 1 cycle / ..without S/CALL ]      )
  NOP NOP NOP   ( wait n * 3 cycles                 )
#LOOP          ( wait n * 4 cycles / 1 cycle        )
;              ( wait 2 cycles                       )
```

NOT

“NOT”

Purpose:

1's Complement of the value on top of the stack.

Category: Arithmetic/logical(single-length) / assembler instruction

MARC4 opcode: 17 hex

Stack effect: EXP (n1 — /n1)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag not affected
BRANCH flag set, if (/n1 = 0)

X Y registers: not affected

Bytes used: 1

See also: XOR NEGATE OR AND

Example:

```
: NOT_Example
9          ( — 1001b  )
NOT       ( 9 — 0110b  )
DROP     ( 6 —        )
;
```

OF

“OF”

Purpose:

Part of the OF ... ENDOF block used within a CASE ... ENDCASE control structure.

OF compares the CASE index value with another 4-bit comparison value. If they are equal, both of them are dropped from the stack and execution continues with the sequence compiled between OF and the next ENDOF. If there was no match, only the comparison value is dropped, and control proceeds with the word following the next ENDOF.

Category: Control structure /qFORTH macro

Library implementation: CODE OF CMP_NE (n1 n2 — n1 [BRANCH flag])
 (if no match then .)
 (S)BRA _\$ENDOF (branch to next OF ... ENDOF.)
 DROP (n1 —)
 [E -1 R 0]
 END-CODE

Stack effect: EXP (n1 n2 — n1) (if no match)
 EXP (n1 n2 —) (if matched)
 RET (—)

Stack changes: EXP: 1 element is popped from the stack, if no match
 EXP: 2 elements are popped from the stack, if match
 RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 3 – 4

See also: ENDOF CASE ENDCASE

Example:

```
5 CONSTANT Keyboard
1 CONSTANT TestPort1
: ONE 1 TestPort1 OUT ;           ( write 1 to the 'TestPort1' )
: TWO 2 TestPort1 OUT ;           ( write 2 to the 'TestPort1' )
: THREE 3 TestPort1 OUT ;         ( write 3 to the 'TestPort1' )
: ERROR DUP TestPort1 OUT ;       ( dump wrong input to the port )
( duplicate value for the following ENDCASE; it drops one n. )

: CASE-Example
  KeyBoard IN                     ( request 1-digit keyboard input )
  CASE                             ( depending on the input value, )
    1 OF ONE ENDOF                 ( one of these words will be acti- )
    2 OF TWO ENDOF                 ( vated. )
    3 OF THREE ENDOF
    ERROR                          ( otherwise ... )
  ENDCASE                          ( n — )
;
```


OR

“OR”

Purpose:	Logical OR of the top two elements on the stack.
Category:	Arithmetic/logical(single-length) / assembler instruction
MARC4 opcode:	0C hex
Stack effect: :	EXP (n1 n2 — [n1 v n2]) RET (—)
Stack changes:	EXP: the TOP element is popped from the stack RET: not affected
Flags:	CARRY flag not affected BRANCH flag set, if ([n1 v n2] = 0)
X Y registers:	not affected
Bytes used:	1
See also:	XOR AND NOT

Example:

```
: ERROR          ( what shall happen in error case:      )
  3R@ 3
  #DO            ( show PC, where CPU fails              )
    I OUT [ E 0 ] ( suppress compiler warnings.                      )
  #LOOP
;
```

```
: Logical        ( part of e3400 selftest kernel program )
  1001b 1100b
  AND
  1000b <>
  IF ERROR THEN  ( IF 'result wrong' THEN call 'ERROR' ! )
  1010b 0011b
  AND
  0010b <> IF ERROR THEN
  1001b 1100b
  OR
  1101b <> IF ERROR THEN
  1010b 0011b
  OR
  1011b <> IF ERROR THEN
  1001b 1100b
  XOR
  0101b <> IF ERROR THEN
  1010b 0011b
  XOR
  1001b <> IF ERROR THEN
;
```

OUT

“OUT”

Purpose:

Write data to one of the 4-bit I/O ports.

Category: Stack operation / assembler instruction

MARC4 opcode: 1F hex

Stack effect: EXP (data port —)
RET (—)

Stack changes: EXP: data and address will be popped from the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: IN

Example:

```
1 CONSTANT Port1
5 CONSTANT Keyboard
: Counter
  15 #DO                ( 15 iterations                )
    I                  ( copy the index from the return stack )
    Port1 OUT          ( data is output to the port 1       )
  #LOOP
;

: INT0
  Keyboard IN          ( input HEX value at keyboard – port 5 )
  #DO                  ( DO-LOOP for the 'in' value       )
    Counter            ( call and execute the counter routine )
  #LOOP
;
```

OVER

“OVER”

Purpose:

Copies the second element onto the top of stack

Category: Stack operation (single-length) / assembler instruction

MARC4 opcode: 27 hex

Stack effect: EXP (n2 n1 — n2 n1 n2)
RET (—)

Stack changes: EXP: 1 element is pushed onto the top of stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: 2OVER

Example:

: OVER-Example

```
3 7 4          ( — 3 7 4 )  
OVER          ( 3 7 4 — 3 7 4 7 )  
2DROP 2DROP  ( 3 7 4 7 — )  
;
```

PICK

“PICK”

Purpose:

Copies a value from anywhere on the EXP stack to the TOS. PICK uses the value on the TOS as an index into the stack, and copies the value from that location in the stack. The value on the TOS [not including the index] is the 0th element.

$$0 \leq x \leq 14$$

NOTE: The actual EXP stack depth is not checked by this function, therefore the user should use the DEPTH instruction to ensure that the defined PICK index value is valid. (i.e.: that the depth of the stack permits the desired index)

Category: Stack operation (single-length) / qFORTH colon definition

Library implementation: `: PICK SP@ ROT 1+ M- (x SPh SPI — SPh' SPI')`
`Y! [Y-]@ (SPh' SPI' — n[x])`
`;`

Stack effect: EXP (x — n[x])
 RET (—)

Stack changes: EXP: not affected
 RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: The contents of the Y register will be changed.

Bytes used: 8 + 'M-'

See also: ROLL

Example:

```
( 0 PICK is equivalent to DUP )  
( 1 PICK is equivalent to OVER )  
: PickRoll  
 1 2 3 4 5 6 7 8           ( write data onto the stack:      )  
 9 Ah Bh Ch Dh Eh Fh 0 1 2 3 4 ( 20 values.                          )  
 0 PICK DROP              ( ..2 3 4 — ..2 3 4 4          )  
 1 PICK DROP              ( ..2 3 4 — ..2 3 4 3          )  
 2 PICK DROP              ( ..2 3 4 — ..2 3 4 2          )  
 9 PICK DROP              ( ..2 3 4 — ..2 3 4 B         )  
 14 PICK DROP             ( ..2 3 4 — ..2 3 4 6         )  
( *** ROLL ***           (                                     )  
 0 ROLL                   ( ..2 3 4 — ..2 3 4          )  
 1 ROLL                   ( ..2 3 4 — ..2 4 3          )  
 13 ROLL                  ( ..2 4 3 — ..2 4 3 7         )  
 10 #DO DROP #LOOP  
 10 #DO DROP #LOOP       ( clear the stack: 20 DROPS      )  
;
```


R>

“R-from”

Purpose:

Removes the top 4-bit value from the RET stack and puts the value on the EXP Stack.

R> pops the RET stack onto the EXP stack. To avoid corrupting the RET stack and crashing the system, each use of R> MUST be preceded by a >R within the same colon definition.

Category: Stack operation (single-length) /qFORTH macro**Library implementation:** CODE R>
R@ DROPR (copy the index and drop the RET stack)
END-CODE**Stack effect:** EXP (— n)
RET (u|u|n —)**Stack changes:** EXP: 1 element is pushed onto the stack
RET: 1 element (3 nibbles) is popped from the stack**Flags:** not affected**X Y registers:** not affected**Bytes used:** 2**See also:** >R I- 2>R 2R@ 2R> 3>R 3R@ 3R>

Example:

```
: 2SWAP          ( swap 2nd byte with top          )
  >R <ROT        ( d1 d2 — n2_h d1                )
R> <ROT         ( n2_h d1 — d2 d1                )
;

( ***** )
: M/MOD          ( d n — n_quot n_rem              )
( ***** )
  >R  Fh <ROT    ( save divider on RET              )
  BEGIN        ( preset quotient = -1              )
  ROT 1+ <ROT   ( increment quotient                )
  R@ M-        ( subtract divider                  )
  UNTIL        ( until an underflow occurs         )
  D>S R> +     ( n_quot d_rem-n — n_quot n_rem     )
;
```

RDEPTH

“R–depth”

Purpose:

Leaves the depth of the RET stack, the current number of 12-bit entries, on top of the EXP stack.

Category:

Interrupt handling / qFORTH colon definition

Library implementation: : RDEPTH RP@ (— RPh RPl)
 D2/ D2/ (compute the modulo 4 number)
 NIP 1– (of entries on the RET stack)
 ; (forget entry of RDEPTH itself)

Stack effect:

EXP (— n)
RET (—)

Stack changes:

EXP: 1 element is pushed onto the stack.
RET: not affected

Flags:

CARRY and BRANCH flags are affected

X Y registers:

not affected

Bytes used:

13

See also:

DEPTH RFREE INTO ... INT7

Example:

```
( ***** )
: Call_Again2
  RDEPTH          ( result is 2.          )
;
( ***** )
: Call_Again1
  RDEPTH          ( result is 1.          )
  Call_Again2     ( next level – new address to RET. )
;
( ***** )
B: RDepth_Example
  RDEPTH          ( result is 0.          )
  Call_Again1     ( next level – new address to RET. )
  3DROP          ( all results into wpb [waste]. )
;
: $RESET
  >RP NoRAM
  >SP S0
  RDepth_Example
;
```

REPEAT

“REPEAT”

Purpose:

Part of the BEGIN ... WHILE ... REPEAT control structure.

REPEAT forces an unconditional branch back to just after the corresponding BEGIN statement.

Category: Control structure /qFORTH macro

Library implementation: CODE REPEAT SET_BCF (set BRANCH flag, force
jump)
BRA _\$BEGIN (jump back to BEGIN)
_ \$LOOP: [E 0 R 0]
END-CODE

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY and BRANCH flags set

X Y registers: not affected

Bytes used: 2 – 3

See also: BEGIN WHILE UNTIL AGAIN

Example:

1 CONSTANT Port1
VARIABLE Count

```
: COUNTER
  Count 1+!          ( increment Count          )
  Count @ Port1 OUT  ( write new Count to Port1 )
;

: BEGIN-WHILE-REPEAT
  10 BEGIN 1-        ( decrement the TOS , from 10 to 0 )
    DUP              ( save TOS )
    0<> WHILE        ( REPEAT decrement while TOS not equal 0 )
      COUNTER        ( other instructions in this loop ... )
    REPEAT           ( after each decrement the TOS contains )
;                    ( the value of the present index. )
```

RFREE

“R-free”

Purpose:

Leaves the number of currently unused return stack entries on top of the expression stack, e.g. the available levels of nesting.

Moves the addresses of the return stackpointer and the expression stack base address [S0] onto the expression stack and subtracts them.

Final result = number of free levels for further 'calls'.

Category: Interrupt handling /qFORTH macro

Library implementation: : RFREE RDEPTH S0 (Rn — Rn S0h S0l)
 D2/ D2/ NIP (Rn S0h S0l — Rn Sn)
 SWAP — (Rn Sn — Rfree)
 ;

Stack effect: EXP (— n)
 RET (—)

Stack changes: EXP: one nibble is pushed onto the expression stack.
 RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: not affected

Bytes used: 17 + 'RDEPTH'

See also: DEPTH RDEPTH

Example:

```
VARIABLE R0 27 ALLOT ( return stack: 28 nibbles, used 21 for      )
                    ( 7 levels of task switching – additionally : )
                    ( 7 nibbles are free for 1–nibble–variables. )
VARIABLE S0 19 ALLOT ( data stack: 20 nibbles.                  )
: RFree3
RFREE DROP          ( result value is: 4                        )

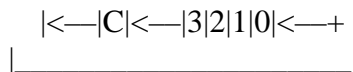
: RFree2
  RFREE DROP        ( result value is: 5                        )
  RFree3            ( “call” next level.                       )
;
: RFree1
  RFREE DROP        ( result value is: 6                        )
  RFree2            ( “call” next level.                       )
;
: INTO
  RFREE DROP        ( result value is: 7                        )
  RFree1            ( “call” next level.                       )
;
```


ROL

“Rotate-left”

Purpose:

Rotate the TOS left through CARRY



Category: Arithmetic/logical(single-length) / assembler instruction

MARC4 opcode: 11 hex

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag = Bit_3 (TOS) – before operation
BRANCH flag = CARRY flag

X Y registers: not affected

Bytes used: 1

See also: ROR SHR SHL D2*

Example:

```
(*****)  
: BitShift  
SET_BCF 3          ( 3 = 0011b          )  
ROR DROP          ( [CARRY] 3 — [CARRY] 9 = 1001b )  
CLR_BCF 3          ( 3 = 0011b          )  
ROR DROP          ( [no CARRY] 3 — [CARRY] 1 = 0001b )  
SET_BCF 3          ( 3 = 0011b          )  
ROL DROP          ( [CARRY] 3 — [no CARRY] 7 = 0111b )  
CLR_BCF 3          ( 3 = 0011b          )  
ROL DROP          ( [no CARRY] 3 — [no CARRY] 6 = 0110b )  
(-----)  
CLR_BCF Fh        (          )  
2/ DROP          ( - SHR - [no CARRY] F — [CARRY] 7 = 0111b )  
(          )  
CLR_BCF 6         ( 6 = 0110b          )  
2* DROP          ( - SHL - [no CARRY] 6 — [no C] C = 1100b )  
;
```

ROLL

“Rol-L”

Purpose:

MOVES a value from anywhere on the EXP stack to the TOS. ROLL uses the value on the TOS as an index into the stack, and moves the value at that location onto the TOS. The value on the TOS [not including the index] is the 0th element.

0 <= x <= 13

- '0 ROLL', does nothing,
- '1 ROLL', is the same as SWAP, and
- '2 ROLL', is equivalent to ROT.
- '3 ROLL', ie. corresponds to (n1 n2 n3 n4 — n2 n3 n4 n1)

Category: Stack operation (single-length) / qFORTH colon definition

Library implementation:

```

: ROLL  ?DUP
      IF
        CCR@ DI >R          ( save current I-flag setting on RET st. )
        1+ PICK >R         ( do a PICK, move PICKed value on
                             RET st. )
        Y@                  ( move ptr from Y -> X reg. )
        1 M+ X!             ( adjust X reg. pointer )
        #DO [+X]@ [+Y]! #LOOP ( shift data values one down )
        DROP R>
        R> CCR!             ( restore I-flag setting in CCR )
      ELSE DROP THEN
    ;
  
```

Stack effect: EXP (x —)
RET (—)

Stack changes: EXP: 1 element is popped from the stack
RET: not affected

Flags: CARRY and BRANCH flags are affected

X Y registers: Both, the X and Y index registers will be affected

Bytes used: 39 + 'PICK' + 'M+'

See also: PICK

xxxxv

Example:

```
: PickRoll
  1 2 3 4 5 6 7 8          ( write data onto the stack:      )
  9 Ah Bh Ch Dh Eh Fh 0 1 2 3 4 ( 20 values.                          )
  0 PICK DROP             ( ..2 3 4 — ..2 3 4 4 =DUP          )
  1 PICK DROP             ( ..2 3 4 — ..2 3 4 3 =OVER        )
  2 PICK DROP             ( ..2 3 4 — ..2 3 4 2          )
  9 PICK DROP             ( ..2 3 4 — ..2 3 4 B          )
  14 PICK DROP            ( ..2 3 4 — ..2 3 4 6          )
  0 ROLL                  ( ..2 3 4 — ..2 3 4          )
  1 ROLL                  ( ..2 3 4 — ..2 4 3          )
  13 ROLL                 ( ..2 4 3 — ..2 4 3 7          )
  10 #DO DROP #LOOP 10 #DO DROP #LOOP ( clear the stack      )
;                          ( 20 DROPS                )
```

ROMByte@TABLE;;

“ROM–byte–fetch”

Purpose:

Fetches an 8-bit constant from ROM from any place, whereby the 12-bit ROM address is on the expression stack.

Category: Memory operation (double-length) / qFORTH colon definition

MARC4 opcode: 20 hex (TABLE)

Library implementation: : ROMByte@ (ROMh ROMm ROMl — d)
 3>R (move TABLE address to RET stack)
 TABLE (— consthigh constlow)
 (TABLE returns directly to the CALLer during microc.exec.)
 [E -1 R 0] ;; (therefore 'EXIT' is not required)

Stack effect: EXP (ROMh ROMm ROMl — consth constl)
 RET (—)

Stack changes: EXP: 1 element will be popped from the stack
 RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 2

See also: DTABLE@ ROMCONST

Example:

ROMCONST DigitTable 10h, 1, 2, 3, 4 45h, 6, 7, 8, 9, Ah, Bh, Ch, Dh, Eh, 0Fh,

```
( take attention to the blanks before and after the ', ' and )
( to the last ', ' ! )
( ***** )
: D_Table@
  0 0 1 ROMByte@      ( fetch byte at address 001h : 0Fh = SLEEP )
  2DROP              ( delete it. )
  DigitTable 3>R      ( save address of L/U table on RET stack )
                    ( sixth byte of the table: )
  3R@      5          ( put address and index on the stack. )
  DTABLE@ 2DROP      ( fetch and delete the value : 45h . )
                    ( second byte of the table: )
  3R@      1          ( put address and index on the stack. )
  DTABLE@ 2DROP      ( fetch and delete the value : 01h . )
                    ( first byte of the table and min. index : )
  3R@              ( put address on the stack. )
  ROMByte@ 2DROP     ( fetch and delete the value : 10h . )
                    ( last byte of the table and max. index )
  DigitTable 15      ( put address and index on the stack. )
  DTABLE@ 2DROP     ( fetch and delete the value : 0Fh )
;

```

ROMCONST

“ROM-CONSTANT”

Purpose:

Defines 8-bit constants in the ROM for look-up tables or as ASCII string constants.

Syntax: attention to the blanks before and after the ',' and to the last ', ' !

Category:	Predefined data structure
Stack effect:	EXP (—) RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	Depends on number of defined items (bytes)
See also:	DTABLE@ ROMByte@

Example 1:

```

13 CONSTANT TextLength    03h CONSTANT LCD_Data
ROMCONST LCD_Text TextLength , ' Text for LCD. ' ,
: ExampleTextLCD
  TextLength #DO           ( loop 'TextLength' times.      )
  LCD_Text I DTABLE@      ( gets: 2Eh '.', 44h 'D',...54h'T' )
  DecodeAscii             ( convert 8-bit -> 16-bit - segm. )
  4 #DO LCD_Data OUT      ( write 4 * 4-bit to LCD.      )
  [ E 0 ] #LOOP           ( suppress warnings of compiler. )
[ E 0 ] #LOOP
;

```

Example 2:

```

0 CONSTANT Aa           0 CONSTANT P1
1 CONSTANT Ab           1 CONSTANT P2
2 CONSTANT Ac           2 CONSTANT P3
3 CONSTANT Ad           3 CONSTANT P4
ROMCONST Matrix 11 , 12 , 13 , 14 ,
  21 , 22 , 23 , 24 ,
  31 , 32 , 33 , 34 ,
  41 , 42 , 43 , 44 ,

: L/U-Table             ( Ss Pn — 8-bit-value      )
  2>R                   ( save indices on return stack )
  Matrix                ( push matrix base address      )
  R@ 2* S>D D2*         ( compute parameter set offset )
  D+                    ( ROMaddr := matrix + Pn * 4    )
  IF ROT 1+ <ROT THEN   ( handle MSD of matrix address )
  2R> DROP              ( load Aa ... Ad from RET stack )
  DTABLE@ ;            ( fetch indexed value from ROM )

: Example2 Ab P4 L/U-table
;

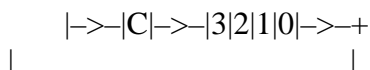
```


ROR

“Rotate-right”

Purpose:

Rotate right through CARRY



Category: Arithmetic/logical (single-length) / assembler instruction

MARC4 opcode: 13 hex

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag = bit_0 (TOS) – before operation
BRANCH flag = CARRY flag

X Y registers: not affected

Bytes used: 1

See also: ROL SHR SHL

Example:

```
( ***** )
: BitShift
SET_BCF 3      ( 3 = 0011b )
ROR DROP      ( [CARRY] 3 — [CARRY] 9 = 1001b )
CLR_BCF 3      ( 3 = 0011b )
ROR DROP      ( [no CARRY] 3 — [CARRY] 1 = 0001b )
SET_BCF 3      ( 3 = 0011b )
ROL DROP      ( [CARRY] 3 — [no CARRY] 7 = 0111b )
CLR_BCF 3      ( 3 = 0011b )
ROL DROP      ( [no CARRY] 3 — [no CARRY] 6 = 0110b )
(-----)
CLR_BCF Fh    ( )
2/ DROP      ( - SHR - [no CARRY] F — [CARRY] 7 = 0111b )
              ( )
CLR_BCF 6     ( 6 = 0110b )
2* DROP      ( - SHL - [no CARRY] 6 — [no C] C = 1100b )
;
```

ROT

“Rote”

Purpose:

Moves the third value on the stack to the top of the stack.

Category: Stack operation (single-length) / assembler instruction

MARC4 opcode: 2C hex

Stack effect: EXP (n1 n2 n3 — n2 n3 n1)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: 2ROT <ROT 2<ROT

Example:

```
: ROTATE_Example
  4 6 9          (    — 4 6 9 )
  ROT           ( 4 6 9 — 6 9 4 )
  3DROP        ( 6 9 4 —    )
;
```

'Return stack pointer operations' RP@ RP!'>RP \$xx'RO

"R-P-fetch" "R-P-store" "To-RP address" "R-zero"

Purpose:

Fetch, restore and initialize the return stack pointer.

Category: assembler instruction

MARC4 opcodes: RP@ = 71 hex
RP! = 75 hex
>RP \$xx = 79xx hex

Stack effect: RP@: EXP (— RPh RPl)
RET (—)

RP!: EXP (RPh RPl —)
RET (—)

>RP \$xx: EXP (—)
RET (—) RP := \$xx

Stack changes: RP@ EXP: 2 elements are pushed onto the stack
RET: not affected

RP! EXP: 2 elements are popped from the stack
RET: return stack pointer modified

>RP \$xx EXP: not affected
RET: new base address

Flags: not affected

X Y registers: not affected

Bytes used: 1 each ; >RP \$xx : 2

See also: SP! SP@ >SP \$xx

Example 1:

```
VARIABLE R0 27 ALLOT ( return stack: 28 nibbles, used 21 for 7 )  
( levels of task switching; 7 nibbles are free for variables. )  
VARIABLE S0 19 ALLOT ( data stack: 20 nibbles. )  
( the qFORTH word RDEPTH uses 'RP@' to push the return stack )  
( pointer onto the expression stack )  
:RDEPTH ( implementation code )  
  RP@ ( EXP stack : — RPh RPl )  
  D2/ D2/ ( compute number of entries on the RET stack )  
  NIP ( EXP stack : 0 n — n )  
  1- ( forget the entry of RDEPTH itself. )  
;
```

Example 2:

```
:$RESET  
> SP S0 ( initialize the stack pointers. )  
> RP NoRAM ( set RP to autosleep address. )  
RDEPTH DROP ( result is 0 )  
;
```

S>D

“Single-to-double”

Purpose:

Transform a 4-bit value to an unsigned 8-bit value. Pushes 0 onto the 2nd stack position.

Category: Stack operation / qFORTH macro

Library implementation: CODE S>D LIT_0 (n — n 0)
SWAP (n 0 — d)
END-CODE

Stack effect: EXP (n — d)
RET (—)

Stack changes: EXP: 1 element (0) is pushed onto the 2nd stack position
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 2

See also: D>S

Example:

```
( ***** )  
4          (          — 4      )  
S>D       (          4 — 0  4  )  
25h D+    (  0  4  2  5 — 2    )  
D>S       (    2    9 — 9      )  
DROP      (          9 —        )  
;
```


SET_BCF

“Set-BRANCH-and-CARRY-flag”

Purpose:

Set BRANCH and CARRY flags in the MARC4 condition code register

Category: Assembler instruction

MARC4 opcode: 19 hex

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY and BRANCH flags set

X Y registers: not affected

Bytes used: 1

See also: TOG_BF CLR_BCF

Example:

```
: Error          ( show program counter, where CPU fails. )
  3R@
  3 #DO          ( write address to port 1, 2 and 3      )
    I OUT [ E 0 ]
    #LOOP [ E 0 ] ( suppress compiler warnings.          )
;

: CCRf
  CCR@ 1010b AND ( fetch – mask the used flags out          )
;

: BC_Flags
  SET_BCF        ( flags: C%Bx – CARRY % BRANCH x      )
  CCRf 1010b     ( BRANCH and CARRY flag is set.      )
  <> IF Error THEN ( no error occured.                                )

  CLR_BCF        ( delete BRANCH and CARRY flag       )
  CCRf 0000b     ( all flags reset or masked off.     )
  <> IF Error THEN ( no error occured.                                )

  CLR_BCF        ( clear table from the compare before. )
  TOG_BF         ( 0000 -> 00B0                        )
  CCRf 0010b     ( BRANCH flag is set.                )
  <> IF Error THEN ( no error occured.                                )

  SET_BCF        ( define new machine state.           )
  TOG_BF         ( C%B% -> C%0%                       )
  CCRf 1000b     ( CARRY flag is still set.           )
  <> IF Error THEN ( no error occured.                                )
;
```

'Stack pointer operations' SP@SP!'>SP \$xx'SO

"S-P-fetch" "S-P-store" "To-SP address" "S-zero"

Purpose:

Fetch, restore or initialize the expression stack pointer.

Category: Assembler instruction

MARC4 opcodes: SP@ = 70 hex >SP \$xx = 78xx
SP! = 74 hex

Stack effect: SP@: EXP (— SPh SPI) >SP \$xx: EXP (—)
RET (—) RET (—)

SP!: EXP (SPh SPI —)
RET (—)

Stack changes: SP@ EXP: 'stack pointer + 1' is pushed onto the stack
RET: not affected

SP! EXP: 2 elements are popped into the stack pointer
RET: not affected

>SP \$xx EXP: the expression stack pointer is initialized
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1 each >SP \$xx : 2

See also: RP! RP@ >RP \$xx

Example 1:

```
( The qFORTH word DEPTH uses 'SP@' to push the EXP stack )  
( pointer onto the expression stack )  
  
: DEPTH SP@ ( — SPh SPI )  
  S0 ( SPh SPI — SPh SPI S0h S0l )  
  D- ( SPh SPI S0h S0l — diffh diffl )  
  NIP ( 0 n — n ) ( result only on nibble, max Fh )  
  1- ( the value of SP on stack is incremented. )  
;
```

Example 2:

```
$RESET  
> SP S0 ( initialize the Stack pointers. )  
> RP NoRAM ( Set RP to autosleep address. )  
DEPTH DROP ( result is 0 )  
;
```

SWAP

“SWAP”

Purpose:

Exchange the top two elements on the expression stack

Category: Stack operation (single-length)

MARC4 opcode: 26 hex

Stack effect: EXP (n2 n1 — n1 n2)
RET (—)

Stack changes: EXP: exchanges the top two elements with one another
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 1

See also: 2SWAP

Example:

: SWAP-Example

3 7 4

(— 3 7 4)

SWAP

(3 7 4 — 3 4 7)

DROP

(3 4 7 — 3 4)

+ 0=

(3 4 — 7 —)

;

'SWI0 . . . SWI7'

“Software-interrupt-zero” ... “Software-interrupt-seven”

Purpose:

These qFORTH words generate a software interrupt. They allow the programmer to postpone less important jobs until all the important work is completed. Under special circumstances it may also be used to spawn higher priority jobs from a lower priority task, to make them less interruptable.

Category:

Interrupt handling

Library implementation:

CODE SWI3 0 8 SWI NOP (the NOP gives time for task-)
END-CODE (switching)

Stack effect:

EXP (—)
RET (—)

Stack changes:

EXP: not affected
RET: not affected

Flags:

not affected

X Y registers:

not affected

Bytes used:

4

See also:

RTI INT0 ... INT7

Example:

```
: INT7          ( external hardware interrupt – level 7      )
  ScanKeys      ( do most important action here and now     )
  SWI3          ( more action later, after this job is      )
Count 1+!      ( finished; activated from the inter-       )
  Count @ 0=    ( rupt logic after the higher priorit      )
  IF Overflow THEN ( ised job is completed.                )
;              ( compiler translates ';' to 'RTI'.         )

: INT3          ( software triggered interrupt              )
  UpdateLCD     ( update LCD display                       )
;              (                                           )
```


T+!

“T-plus-store”

Purpose:

ADD the TOS 12-bit value to a 12-bit variable in RAM and store the result in that variable. On function entry the start/base address of the array is the TOS value.

Category: Memory operation (triple-length) / qFORTH colon definition

Library implementation:

```

: T+!  2  M+      ( increment addr      )
      Y!         ( nh nm nl addr — nh nm nl )
      [Y]@ +     ( nh nm nl — nh nm nl' )
      [Y-]!      ( nh nm nl' — nh nm )
      [Y]@ +c    ( nh nm — nh nm' )
      [Y-]!      ( nh nm' — nh )
      [Y]@ +c    ( nh — nh' )
      [Y]!       ( nh' — )
;
    
```

Stack effect: EXP (nh nm nl addr —)
RET (—)

Stack changes: EXP: 5 elements are pushed from the stack
RET: not affected

Flags: CARRY flag set, if arithmetic overflow
BRANCH flag = CARRY flag

X Y registers: The contents of the Y register will be changed.

Bytes used: 14 + 'M+'

See also: 3@ 3! T-!

Example:

```
3 ARRAY 3Nibbles AT 40h
( ***** )
: Triples
  1 2 3 3Nibbles      ( store 123 in the 3 nibbles array.      )
  3!                  (                                       )
  3 2 1 3Nibbles     ( 123 + 321 = 444                               )
  T+!                (                                       )
  3Nibbles 3@ 3>R    ( save the result onto return stack.      )
  1 2 3 3Nibbles     ( 444 - 123 = 321                               )
  T-!                (                                       )
  DROPR              ( forget the saved result on return stack. )
;
```

T-!

“T-minus-store”

Purpose:

Subtracts the TOS 12-bit value from a 12-bit variable in RAM and store the result in that variable. On function entry the start/base address of the array is the TOS value.

Category: Memory operation (triple-length) / qFORTH colon definition

Library implementation:

```

: T-!  2  M+      ( increment addr      )
      Y!         ( nh nm nl addr — nh nm nl )
      [Y]@ SWAP  ( nh nm nl — nh nm rl nl  )
      - [Y-]!    ( nh nm rl nl — nh nm      )
      [Y]@ SWAP  ( nh nm — nh rm nm      )
      -c [Y-]!   ( nh rm nm — nh          )
      [Y]@ SWAP  ( nh — rh nh           )
      -c [Y]!    ( rh nh —             )
;

```

Stack effect: EXP (nh nm nl addr —)
RET (—)

Stack changes: EXP: 5 elements are pushed from the stack
RET: not affected

Flags: CARRY flag set, if arithmetic underflow
BRANCH flag = CARRY flag

X Y registers: The contents of the Y register will be changed.

Bytes used: 17 + 'M+'

See also: 3@ 3! T+!

Example:

```
3 ARRAY 3Nibbles AT 40h
( ***** )
: Triples
  1 2 3 3Nibbles      ( store 123 in the 3 nibbles array.      )
  3!                  (                                       )
  3 2 1 3Nibbles     ( 123 + 321 = 444                               )
  T+!                 (                                       )
  3Nibbles 3@ 3>R     ( save the result onto return stack.         )
  1 2 3 3Nibbles     ( 444 - 123 = 321                               )
  T-!                 (                                       )
  DROPR              ( forget the saved result on return stack )
;
```

TD+!

“T-D-plus-store”

Purpose:

ADD the 8-bit number on the stack to an 12-bit variable in RAM and store the result in that variable. On function entry the address of the variable is the TOS.

Category: Arithmetic/logical (triple-length) / qFORTH colon definition

Library implementation:

```

: TD+!  2  M+      ( increment addr  )
        Y!        ( nh nl addr — nh nl  )
        [Y]@ +    ( nh nl — nh rl'   )
        [Y-]!    ( nh rl' — nh      )
        [Y]@ +c  ( nh — nh rm'     )
        [Y-]! 0  ( nh rm' — 0      )
        [Y]@ +c  ( 0 — rh'        )
        [Y]!    ( rh' —          )
;
    
```

Stack effect: EXP (d addr —)
RET (—)

Stack changes: EXP: 4 elements are popped from the stack.
RET: not affected

Flags: CARRY flag set, if arithmetic overflow
BRANCH flag = CARRY flag

X Y registers: The contents of the Y register will be changed.

Bytes used: 15 + 'M+'

Example:

```
( ***** )
3 ARRAY 3Nibbles
( ***** )
: TripDigi
  Fh 8 7 3Nibbles 3!      ( store F87 in the RAM.      )
  44h 3Nibbles TD+!      ( F87 + 44 = FCB  CARRY: %  BRANCH: %  )
  44h 3Nibbles TD+!      ( FCB + 44 = [1]00F CARRY: 1  BRANCH: 1  )

  44h 3Nibbles TD-!      ( 00F - 44 = FCB  CARRY: 1  BRANCH: 1  )
  44h 3Nibbles TD-!      ( FCB - 44 = F87  CARRY: %  BRANCH: %  )
;
```

TD-!

“T-D-minus-store”

Purpose:

SUBTRACT the 8-bit number on the stack from an 12-bit variable in RAM and store the result in that variable. On function entry the address of the variable is the TOS.

Category: Arithmetic/logical (triple-length) / qFORTH colon definition

Library implementation:

```

: TD-! 2 M+      ( increment addr      )
      Y!         ( nh nl addr — nh nl  )
      [Y]@ SWAP  ( nh nl — nh rl nl  )
      - [Y-]!    ( nh rl nl — nh      )
      [Y]@ SWAP  ( nh — rm nh      )
      -c [Y-]!   ( rm nh — [borrow] )
      [Y]@ 0     ( [borrow] — rh 0   )
      -c [Y]!    ( rh 0 —          )
    
```

Stack effect: EXP (d addr —)
RET (—)

Stack changes: EXP: 4 elements are popped from the stack.
RET: not affected

Flags: CARRY flag set, if arithmetic underflow
BRANCH flag = CARRY flag

X Y registers: The contents of the Y register will be changed.

Bytes used: 17 + 'M+'

See also: 3@ 3! T-!

Example:

```
(*****)  
3 ARRAY 3Nibbles  
(*****)  
: TripDigi  
  Fh 8 7 3Nibbles 3!      ( store F87 in the RAM.      )  
  44h 3Nibbles TD+!      ( F87 + 44 = FCB  CARRY: %  BRANCH: %  )  
  44h 3Nibbles TD+!      ( FCB + 44 = [1]00F CARRY: 1  BRANCH: 1  )  
  
  44h 3Nibbles TD-!      ( 00F - 44 = FCB  CARRY: 1  BRANCH: 1  )  
  44h 3Nibbles TD-!      ( FCB - 44 = F87  CARRY: %  BRANCH: %  )  
;
```


THEN

“THEN”

Purpose:

Part of the IF ... ELSE ... THEN control structure, which closes the corresponding IF statement.

The IF block will be executed ONLY if the condition is TRUE, otherwise – if available – the ELSE block will be executed. If the condition is FALSE, the processing goes on with the ELSE part; if there is no ELSE part, the processing goes on after the THEN label.

Category:	Control structure
Library implementation:	CODE THEN _\$THEN: [E O R O] END-CODE
Stack effect:	EXP (—) RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	CARRY and BRANCH flags are affected
X Y registers:	not affected
Bytes used:	0
See also:	IF ELSE

Example:

```
: IfElseThen
  1 2 <=          ( is 1 <= 2 ?           )
  IF              ( yes, so the If block will be executed. )
    1             ( a 1 will be pushed onto the stack.     )
  ELSE           (                               )
    0             ( true => no execution of the ELSE block. )
  THEN           (                               )
  1 2 >          ( is 1 > 2 ?           )
  IF              (                               )
    DROP 0        ( false: nothing will be executed.       )
  THEN           (                               )
               (                               )
  1 2 >          ( is 1 > 2 ?           )
  IF              (                               )
    0             ( not true => no execution.              )
  ELSE           ( in this case, the                       )
    1             ( ELSE block will be executed           )
  THEN 2DROP     ( the results from the expression stack.  )
;
```

TOGGLE

“TOGGLE”

Purpose:

TOGGLEs [exclusive-or] a 4-bit variable at a specified memory location with a given bit pattern. On entry to this function, the 8-bit address of the variable is on the top of Stack.

Category: Memory operation (single-length) /qFORTH macro

Library implementation: CODE TOGGLE
 Y! (n1 addr — n1)
 [Y]@ (n1 — n1 n2)
 XOR (n1 n2 — n3)
 [Y]! (n3 —)
 END-CODE

Stack effect: EXP (n1 addr —)
 RET (—)

Stack changes: EXP: 3 elements are popped from the stack
 RET: not affected

Flags: CARRY flag not affected
 BRANCH flag Set, if (TOS = 0)

X Y registers: The contents of the Y or X register may be changed.

Bytes used: 4

See also: DTOGGLE XOR

Example:

```
( ***** )  
VARIABLE Hoss  
: D_Toggle  
 5 Hoss !      ( set in the RAM one nibble to 0101  )  
 3 Hoss TOGGLE ( truth table: 0101 XOR 0011 = 0110  )  
                ( flags: no BRANCH                )  
 Fh Hoss !     ( set in the RAM one nibble to Fh.  )  
 Fh Hoss TOGGLE ( 1111 XOR 1111 = 0000            )  
                ( flags: BRANCH                    )  
 Fh Hoss TOGGLE ( 0000 XOR 1111 = 1111            )  
                ( flags: no BRANCH                )  
;
```

TOG_BF

“Toggle-BRANCH-flag”

Purpose:

Toggle the state of the BRANCH flag in the MARC4 condition code register.

If the BRANCH flag = 1 THEN set the BRANCH flag to 0 ELSE set the BRANCH flag to 1.

Category: Assembler instruction

MARC4 opcode: 18 hex

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag not affected
BRANCH flag = NOT BRANCH flag

X Y registers: not affected

Bytes used: 1

See also: SET_BCF CLR_BCF

Example:

```
: Error                ( what shall happen in error case:      )
  3R@ 3
  #D                   ( show PC, where CPU fails          )
  I OUT [ E 0 ]        ( suppress compiler warnings.        )
  #LOOP
;
: CCRf CCR@ 1010b AND ; ( fetch – mask the used flags out      )
: BC_Flags
  SET_BCF              ( flags: C%BI – CARRY % BRANCH x    )
  CCRf 1010b           ( BRANCH and CARRY flag is set.      )
  <> IF Error THEN      ( no error occurred.          )
  CLR_BCF              ( delete BRANCH and CARRY flag    )
  CCRf 0000b           ( all flags reset or masked off.      )
  <> IF Error THEN      ( no error occurred.          )
  CLR_BCF              ( clear table from the compare before. )
  TOG_BF              ( 0000 -> 00B0                    )
  CCRf 0010b           ( BRANCH flag is set.          )
  <> IF Error THEN      ( no error occurred          )
  SET_BCF              ( define new machine state.        )
  TOG_BF              ( C%B% -> C%0%                    )
  CCRf 1000b           ( CARRY flag is still set.        )
  <> IF Error THEN      ( no error occurred.        )
;
```

TUCK

“TUCK”

Purpose:

Duplicates the top of the stack and copies the top of stack under the second 4-bit value.

Category: Stack operation (single-length) /qFORTH macro

Library implementation: CODE TUCK SWAP (n1 n2 — n2 n1)
OVER (n2 n1 — n2 n1 n2)
END-CODE

Stack effect: EXP (n1 n2 — n2 n1 n2)
RET (—)

Stack changes: EXP: 1 element is pushed onto the stack
RET: not affected

Flags: not affected

X Y registers: not affected

Bytes used: 2

See also: 2TUCK ROT OVER

Example:

```
: TUCK-Example
  1 2          ( — 1 2      )
  TUCK        ( 1 2 — 2 1 2 )
  3DROP      ( 2 1 2 —    )
;
```


UNTIL

“UNTIL”

Purpose:

Part of the BEGIN ... UNTIL control structure.

When UNTIL is executed, the BRANCH flag determines the result of a conditional branch.

If the BRANCH flag is set (TRUE), execution will continue with the word following the UNTIL statement. If the BRANCH flag is FALSE, control branches back to the word following the BEGIN.

Category: Control structure

Library implementation: CODE UNTIL TOG_BF (complement BRANCH
flag)
BRA _\$BEGIN (BRANCH to BEGIN, if
TRUE)
_\$LOOP: [E 0 R 0]
END-CODE

Stack effect: EXP (—)
RET (—)

Stack changes: EXP: not affected
RET: not affected

Flags: CARRY flag not affected
BRANCH flag = NOT BRANCH flag

X Y registers: not affected

Bytes used: 2 – 3

See also: BEGIN

Example:

```
: Example
3 BEGIN          ( increment from 3 until 7          )
  1+             ( TOS := TOS + 1                  )
  DUP           ( DUP current TOS because the compiler will )
  7 = UNTIL     ( DROP the current value            )
;              ( the BRANCH flag is reset after the loop. )
```

VARIABLE

“VARIABLE”

Purpose:

Allocates RAM memory for storage of a 4-bit value.

The qFORTH syntax is as follows

```
VARIABLE <name> [ AT <addr.> ] [ <number> ALLOT ]
```

At compile time, VARIABLE adds <name> to the dictionary and ALLOTs memory for storage of one normal-length value. If AT <addr.> is appended, the variable/s will be placed at a specific address (i.e.: 'AT 40h').

If <number> ALLOT is appended, a set of <number> 4-bit memory locations will be allocated. At execution time, <name> leaves the RAM start address on the expression stack.

The storage ALLOTed by VARIABLE is not initialized.

Category:	Predefined data structure
Stack effect:	EXP (— d) at execution time. RET (—)
Stack changes:	EXP: not affected RET: not affected
Flags:	not affected
X Y registers:	not affected
Bytes used:	0
See also:	2VARIABLE ALLOT ARRAY CONSTANT

Example:

(Constants:)

3 CONSTANT TimeLen

(Variables:)

VARIABLE R0 27 ALLOT (return stack: 28 nibbles, used 21 for)
(7 levels of task switching.)

VARIABLE S0 19 ALLOT (7 nibbles are free for 1nibble variat.)
(data stack: 20 nibbles.)

VARIABLE MainMode (one 4-bit variable)

2VARIABLE LargeCounter (one 8-bit variable)

VARIABLE Nibble50 AT 50h (4-bit at a specific address in RAM)

VARIABLE MoreNibbles AT 40h 6 ALLOT (6 nibbles in the RAM.)

(Arrays:)

TimeLen ARRAY ActualTime (3 nibble array.)

WHILE

“WHILE”

Purpose:

Part of the BEGIN ... WHILE ... REPEAT control structure. When WHILE is executed, the BRANCH flag determines the destination of a conditional branch.

If the BRANCH flag is set(TRUE), the words between WHILE and REPEAT are executed. If the BRANCH flag is FALSE, control branches to the word just past REPEAT.

Category: Control structure

Library implementation: CODE WHILE TOG_BF(complement BRANCH flag)
 BRA _\$LOOP (condjump just behind REPEAT loop)
 [E 0 R 0]
 END-CODE

Stack effect: EXP (—)
 RET (—)

Stack changes: EXP: not affected
 RET: not affected

Flags: CARRY flag not affected
 BRANCH flag = NOT BRANCH flag

X Y registers: not affected

Bytes used: 1 – 3

See also: BEGIN REPEAT UNTIL

Example:

```
: Example
  10 BEGIN          ( decrement the TOS , from 10 to 0.          )
    1- DUP          ( save TOS                                  )
    0<> WHILE      ( REPEAT decrement while TOS not equal 0    )
    REPEAT          ( after each decrement the TOS contains    )
                   ( the value of the present index            )
;
```

X-Register-Operations

X@ [X]@ [+X]@ [X-]@ [>X]@ \$xx'[X]!

[+X]! [X-]! [>X]! \$xx' X! '>X \$xx'

“X-fetch” “indirect-X-fetch” “pre-increment-indirect-X-fetch” “post-dec..” ...

Purpose:

The X register can be used as pointer to access variables or arrays in the RAM. For the compilation of a qFORTH word which directly uses these MARC4 instructions, the qFORTH compiler switch XYTRACE should be turned off (no optimizing).

Category: Assembler instruction

MARC4 opcodes:

X@	= 72 hex	[+X]@	= 31 hex	[>X]	@\$xx	= 33xx hex
[X]@	= 30 hex	[X-]@	= 32 hex			
X!	= 76 hex	[+X]!	= 39 hex	[>X]!	\$xx	= 3Bxx hex
[X]!	= 38 hex	[X-]!	= 3A hex	>X	\$xx	= 7Axx hex

Stack effect:

EXP	(— n)	@-operations;	X@ :	(— x_h x_l)
EXP	(n —)	!-operations;	X! :	(x_h x_l —)
EXP	>X \$xx :	(—)		
RET	(—)	(all)		

Stack changes:

EXP: 1 element is pushed onto the stack (@)
 EXP: 1 element is popped from the stack (!)
 EXP: not affected (>X \$xx)
 RET: not affected (all)

Flags:

not affected

X Y registers:

The X register will be changed by most instructions.

Bytes used:

1 each; except '>X / [>X]! / [>X]@ \$xx': 2 bytes

See also:

Y! [Y]! [+Y]! [Y-]! '[>Y]! \$xx'
 Y@ [Y]@ [+Y]@ [Y-]@ '[>Y]@ \$xx' '>Y \$xx'

Example:

```
( ***** )  
: M+! ( n addr — )  
( ***** )  
( Purpose: Add the 4-bit number on TOS to a 8-bit byte in RAM )  
( ***** )  
  X! [+X]@ + [X-]!  
  [X]@ 0 +c [X]!  
;
```


XOR

“XOR”

Purpose:

XOR leaves the bit-wise logical exclusive-or of the top two values on the stack.

Category: Arithmetic/logical (single-length) / assembler instruction

MARC4 opcode: 04 hex

Stack effect: EXP (n1 n2 — [n1 XOR n2])
RET (—)

Stack changes: EXP: 1 element is popped from the stack.
RET: not affected

Flags: CARRY flag not affected
BRANCH flag set, if ([n1 XOR n2] = 0)

X Y registers: not affected

Bytes used: 1

See also: TOGGLE OR AND

Example:

```

: Error          ( what shall happen in error case:      )
  3R@ 3          ( show PC, where CPU fails              )
  #DO           ( suppress compiler warnings.           )
  I OUT [ E 0 ]
  #LOOP
;

: XOR_Example    ( — )
  ( truth table:  a  b  a XOR b                          )
  (               0  0  0                                )
  (               0  1  1                                )
  (               1  0  1                                )
  (               1  1  0                                )
  (=====)
  (= hexadec.:  3  5  6                                  )
  (                                                     )
  3 5 XOR        ( BRANCH flag is reset [result <> 0]   )
  6 <> IF         ( XOR top two values; error will not be )
  Error         ( activated                               )
  THEN          (                                       )

  1100b 1100b XOR ( BRANCH flag is set [result = 0]     )
  0000b <> IF     ( XOR top two values; error will not be )
  Error         ( activated.                               )
  THEN          (                                       )
;

```

Y-Register-operations

Y@ [Y]@ [+Y]@ [Y-]@' [>Y]@ \$xx'[Y]!

[+Y]! [Y-]!' [>Y]! \$xx'Y! '>Y \$xx'

“Y-fetch” “indirect-Y-fetch” “pre-increment-indirect-Y-fetch” “post-dec..” ...

Purpose:

The Y register can be used as pointer to access variables or arrays in the RAM. For the compilation of a qFORTH word which directly uses these MARC4 instructions, the qFORTH compiler switch XYTRACE should be turned off (no optimizing).

Category: Assembler instruction

MARC4 opcodes:

Y@	= 73 hex	[+Y]@	= 35 hex	[>Y]	@ \$xx	= 37xx hex
[Y]@	= 34 hex	[Y-]@	= 36 hex			
Y!	= 77 hex	[+Y]!	= 3D hex	[>Y]!	\$xx	= 3Fxx hex
[Y]!	= 3C hex	[Y-]!	= 3E hex	>Y	\$xx	= 7Bxx hex

Stack effect: EXP (— n) @-operations; Y@ : (— n_h n_l)
 EXP (n —) !-operations; Y! : (n_h n_l —)
 EXP >Y \$xx : (—)
 RET (—) (all)

Stack changes: EXP: 1 element is pushed onto the stack (@)
 EXP: 1 element is popped from the stack (!)
 EXP: not affected (>Y \$xx)
 RET: not affected (all)

Flags: not affected

X Y registers: The Y register will be changed by most instructions.

Bytes used: 1 each; except '>Y / [>Y]! / [>Y]@ \$xx': 2 bytes

See also: X! [X]! [+X]! [X-]! '[>X]! \$xx'
 X@ [X]@ [+X]@ [X-]@ '[>X]@ \$xx' '>X \$xx'

Example 1:

```
4 ARRAY Ramaddr AT 30h
: Y-STORE
  $OPTIMIZE -XY          ( switch the XY-optimiser off.          )
  >Y Ramaddr             ( assign the variable Ramaddr to the Y reg )
  5 [Y]!                 ( store 5 in the RAM location 30         )
  6 [Y-]!                ( store 6 in the RAM location 30         )
  2 [Y-]!                ( store 2 in the RAM location 29         )
  7 [+Y]!                ( store 7 in the RAM location 29         )
  9 [>Y]! 8              ( store 9 in the RAM location 08         )
  0 4 Y!                 ( assign the Y register to RAM address $04 )
  $OPTIMIZE +XY          ( switch the XY-optimiser on again.      )
;
```

Example 2:

```
: M-!                    ( n addr — )
( Purpose: Subtract a 4-bit number on TOS from 8-bits in RAM )
Y! [+Y]@ - [Y-]! [Y]@ 0 -c [Y]!
;
```