# MARC4 4-bit Microcontrollers

# Programmer's Guide

# Table of Contents

# Section 1

# Hardware Description

## 1.1 Features

- **4-bit HARVARD Architecture**
- **High-level Language Oriented CPU**
- **256 × 4 bits of RAM**
- **Up to 9 KBytes of ROM**
- **8 Vectored Prioritized Interrupt Levels**
- **Low Voltage Operating Range**
- **Low Power Consumption**
- **Power-down Mode**
- **Various On-chip Peripheral Combination Available**
- **qFORTH High-level Programming Language**
- **Programming and Testing is Supported by an Integrated Software Development System**

## 1.2 Introduction

Atmel's MARC4 microcontroller family is based on a low-power 4-bit CPU core. The modular MARC4 architecture is HARVARD like, high-level language oriented and well suited to realize high integrated microcontrollers with a variety of applications 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 a real-time emulator. The stack-oriented microcontroller concept enables the qFORTH compiler to generate a compact and efficient MARC4 program code.

## 1.3 General Description

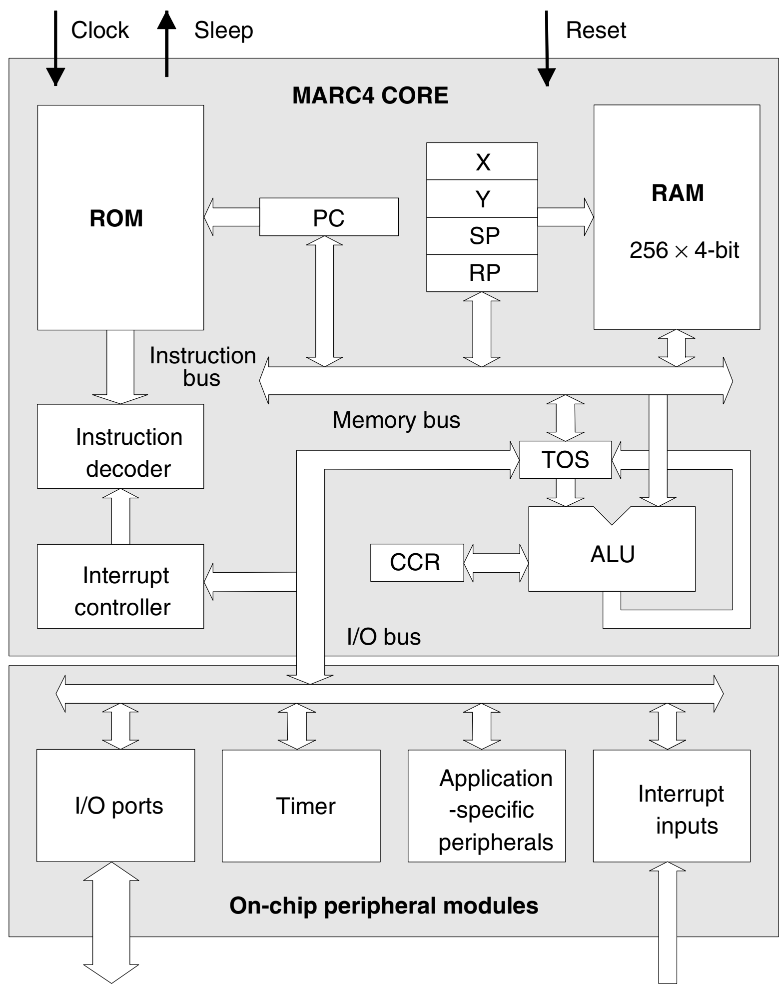
The MARC4 microcontroller consists of an advanced stack-based 4-bit CPU core and application-specific, on-chip peripherals such as I/O ports, timers, counters, ADC, etc.

The CPU is based on the HARVARD architecture with a 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 powerful integrated interrupt controller, with eight prioritized interrupt levels, supports fast processing of hardware events.

The MARC4 is designed for the qFORTH high-level programming language. A lot of qFORTH instructions and two stacks, the Return Stack and the Expression Stack, are already implemented. The architecture allows high-level language programming without any loss of efficiency and code density.

***Figure 1-1.*** MARC4 Core



## 1.4 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.

### 1.4.1 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 of 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 ROM bank register (RBR). Each ROM bank has a size of 2 Kbytes and is placed above the base bank in the upper 2 K (800h-FFFh) of the address space. This therefore enables program memory sizes of up to 10 Kbytes. 1 Kbyte of bank 3 is 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 1-2.

Look-up tables of constants are also stored in ROM and are accessed via the MARC4 built in TABLE instruction.

### 1.4.2 Data Memory (RAM)

The MARC4 contains a 256 × 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.

#### 1.4.2.1 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 in the same way as an accumulator.

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

***Figure 1-2.*** ROM Map

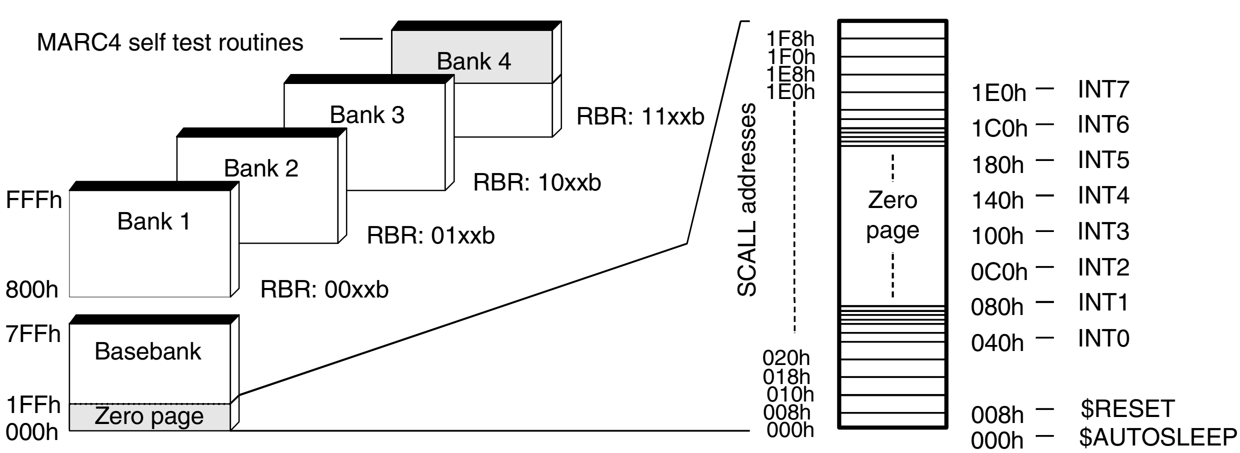
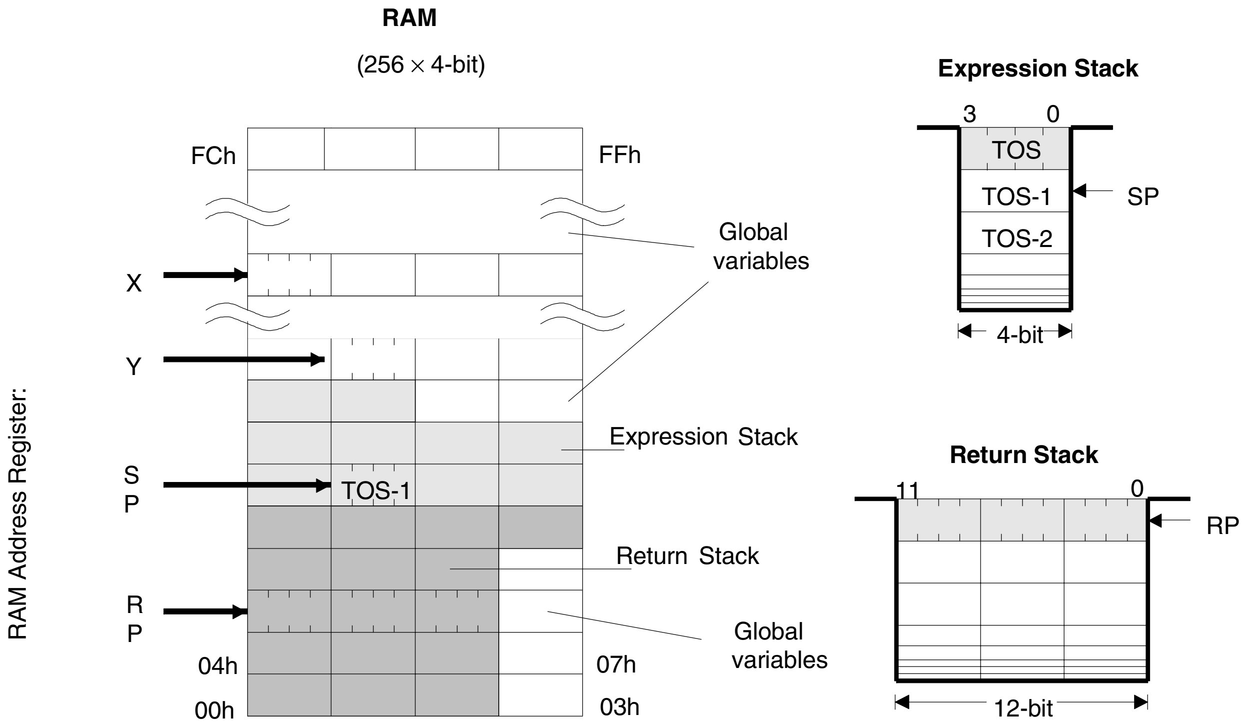


***Figure 1-3.*** RAM Map




**1.4.2.2 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.

**1.4.3 Registers**

The MARC4 controller has six programmable registers and one condition code register. They are shown in the programming model in Figure 1-4.

**1.4.3.1 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.

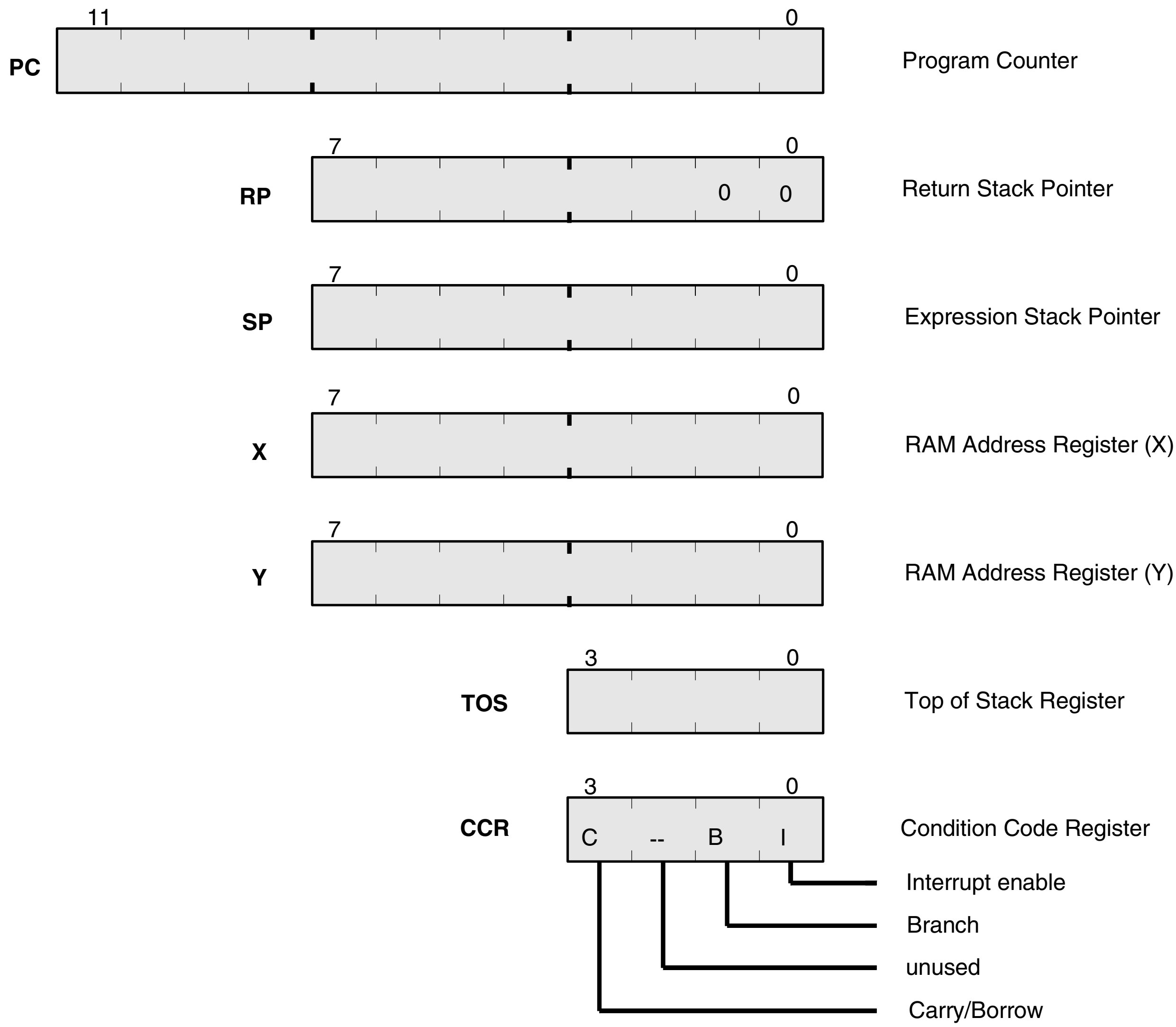
For linear code (no calls or branches), the program counter is incremented with every instruction cycle. If a branch, call, return instruction or an interrupt is executed, the program counter is loaded with a new address.

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

**1.4.3.2 RAM Address Register**

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

***Figure 1-4.*** Programming Model




### 1.4.3.3 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 if 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.

### 1.4.3.4 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. This location is 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 in a special way. Read operations will return the AUTOSLEEP address 000h, whereby write operations have no effect. After a reset, the Return Stack Pointer has to be initialized with " >RP FCh ".

#### 1.4.3.5 RAM Address Register (X and Y)

The X and Y registers 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.

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

#### 1.4.3.6 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.

#### 1.4.3.7 Condition Code Register (CCR)

The 4-bit wide Condition Code Register contains the branch, the carry and the interrupt enable flag. These bits indicate 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.

#### 1.4.3.8 Carry/Borrow (C)

The Carry/Borrow flag indicates that the borrowing or carrying out of the Arithmetic Logic Unit (ALU) occurred during the last arithmetic operation. During shift and rotate operations, this bit is used as a fifth bit. Boolean operations have no effect on the Carry flag.

#### 1.4.3.9 Branch (B)

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

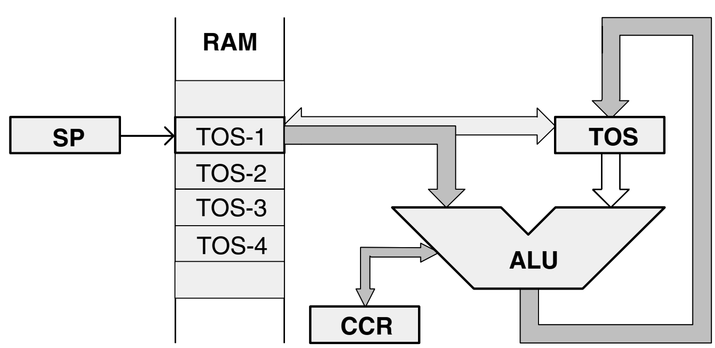
#### 1.4.3.10 Interrupt Enable (I)

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

### 1.4.4 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 1-5.*** ALU Zero Address Operations




### 1.4.5 Instruction Set

The MARC4 instruction set is optimized for the qFORTH high-level programming language. A lot of MARC4 instructions are qFORTH words. This enables the compiler to generate a 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 but no source or destination address information. The operations are performed with the data placed on the stack.

An instruction pipeline enables the controller to fetch the next instruction from ROM at the same time as the present instruction is being executed. One- and two-byte instructions 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.

#### 1.4.5.1 MARC4 Instruction Timing

The internal instruction timing and pipelining during the MARC4's instruction execution are shown in Figure 1-6.

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 machine-cycle and the third instruction needs two machine-cycles.

### 1.4.6 I/O Bus

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 1-1 on page 8).

During IN/OUT operations, the address and data, and during an interrupt cycle the low and the high priority interrupts are multiplexed by using the N_Cycle signal. When N_Cycle is low the address respectively or the low interrupts “0, 1, 2, 3” are sent, when N_Cycle is high the data respectively or the higher priority interrupts “4, 5, 6, 7” are transfered (see Figure 1-7).

An IN operation transfers the port address from TOS (Top Of Stack) onto the I/O bus and reads 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.

***Figure 1-6.*** Instruction Cycle (Pipelining)

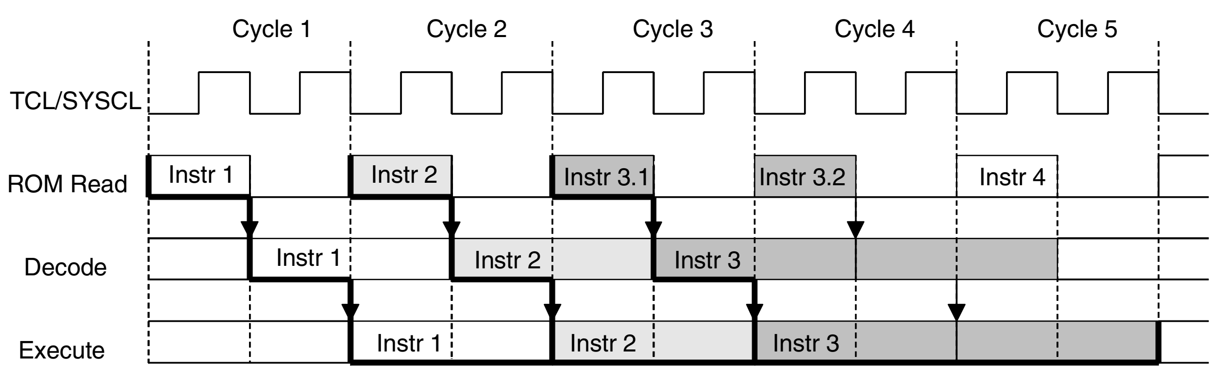
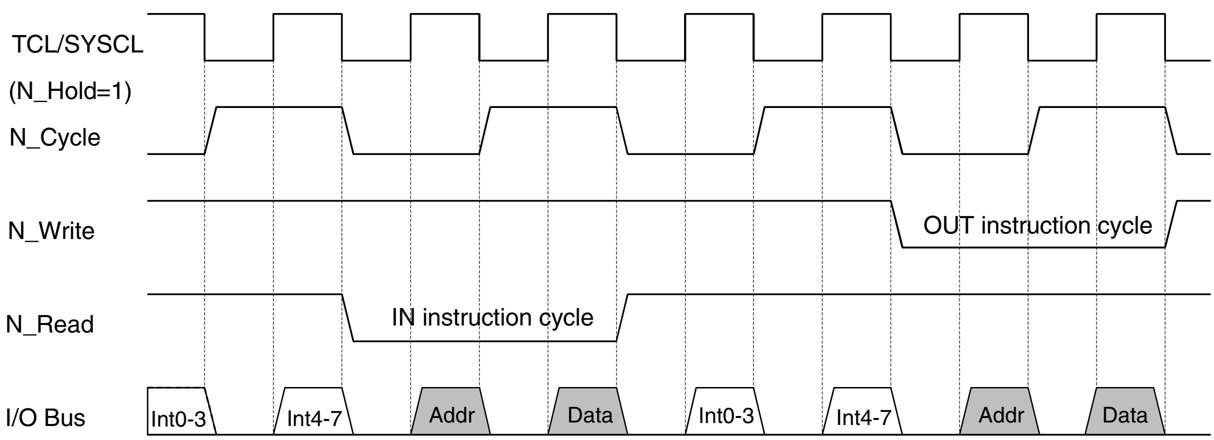


***Table 1-1.*** I/O Bus Modes

| 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 1-7.*** Timing for IN/OUT Operations and Interrupt Requests




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

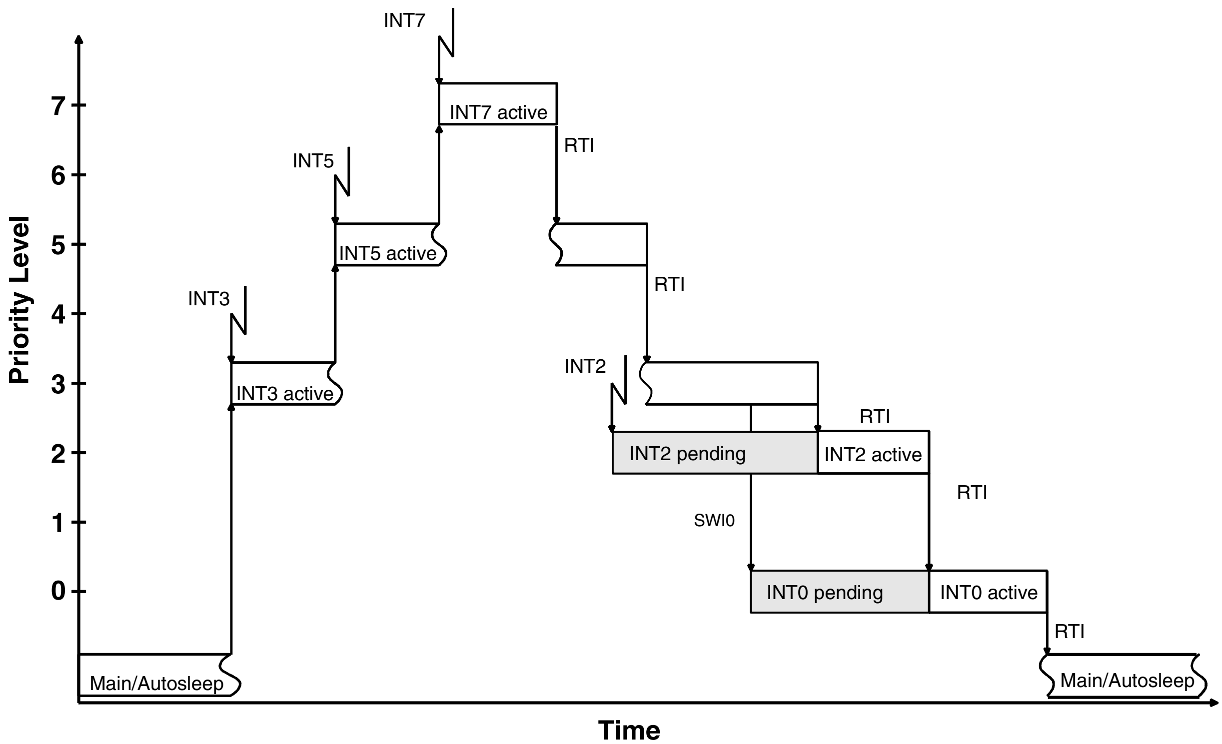
### 1.4.7 Interrupt Structure

The MARC4 can handle interrupts with eight 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 1-2). The programmer can enable or disable all interrupts at once by setting or resetting the Interrupt-enable flag (I) in the CCR.

#### 1.4.7.1 Interrupt Processing

To process the eight 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.

***Figure 1-8.*** Interrupt Handling




An interrupt service routine is finished with the RTI instruction. This instruction 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 has been 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 an 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 Registers are reset.

#### 1.4.7.2 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 three to five machine cycles depending on the state of the core.

#### 1.4.7.3 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 in the same way as 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. Therefore, by using the SWI instruction, interrupts can be re-prioritized or lower priority processes scheduled for later execution.

#### 1.4.7.4 Hardware Interrupts

Hardware interrupt sources such as external interrupt inputs, timers etc. are used for fast automatically event-controlled program flow. The different vectored interrupts permit program dividing into different interrupt-controlled tasks.

**Figure 1-9.** Interrupt Request Cycle

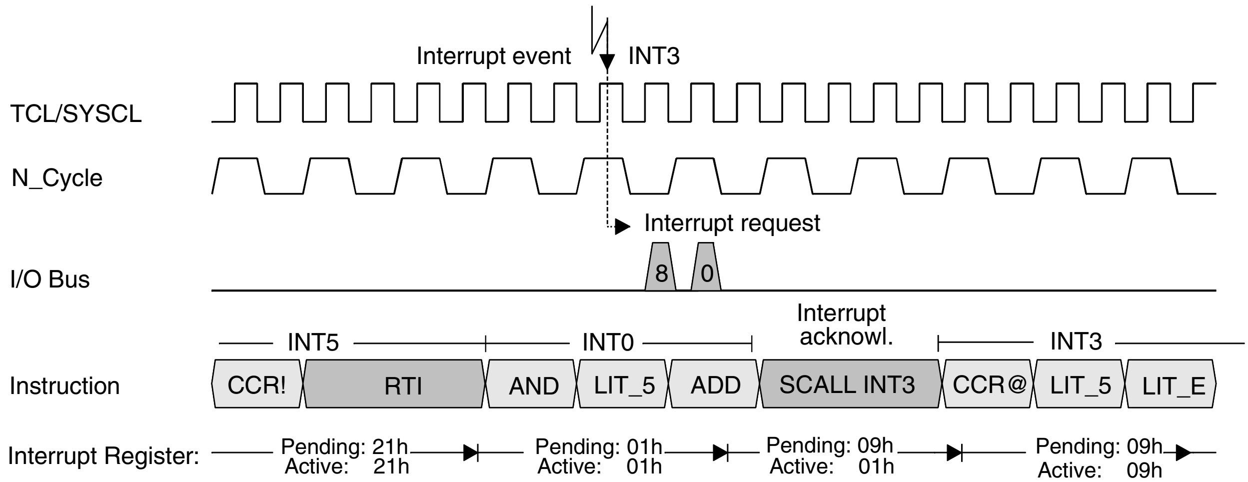
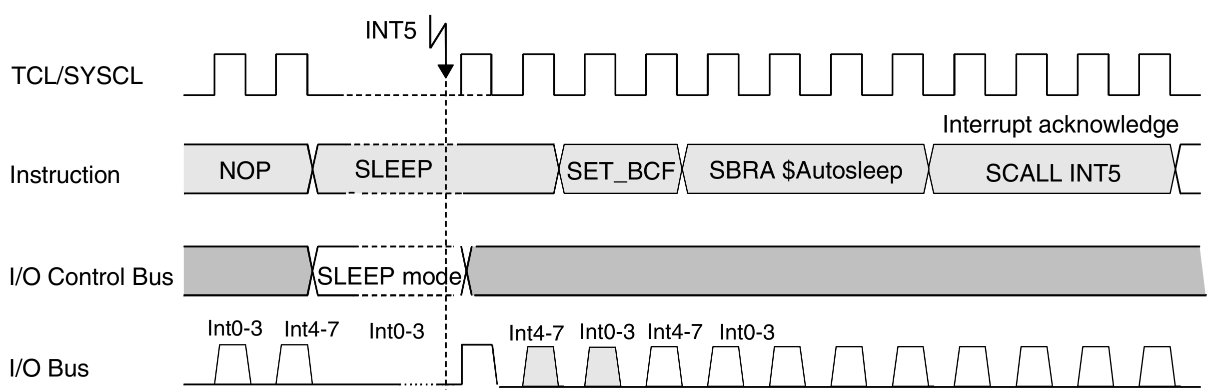



**Table 1-2.** Interrupt Priority Table

| 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 |

**Figure 1-10.** Timing Sleep Mode



## 1.5 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. In this routine the stack pointers, variables in the RAM and the peripheral must be initialized.

## 1.6 Sleep Mode

The sleep mode is a shutdown condition which is used to reduce the average system power consumption in applications where the microcontroller is not fully utilized. 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 microcontroller exits the SLEEP mode with any interrupt or a reset.

The sleep mode can only be kept when none of the Interrupt Pending or Active Register bits are set. The application of the $AUTOSLEEP routine ensures the correct function of the sleep mode.

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

$$\mathbf{I_{total} = I_{Sleep} + (I_{DD} \times T_{active}/T_{total})}$$

$I_{DD}$ depends on $V_{DD}$ and $f_{SYSCL}$.

## 1.7 Peripheral Communication

All communication to and from on-chip peripheral modules takes place via the peripheral I/O bus. In this way the I/O does not interfere with core internal operations. Data transfer is always mastered by the core CPU. A peripheral device, if necessary, can however draw attention to itself by means of an interrupt.

### 1.7.1 Port Communication

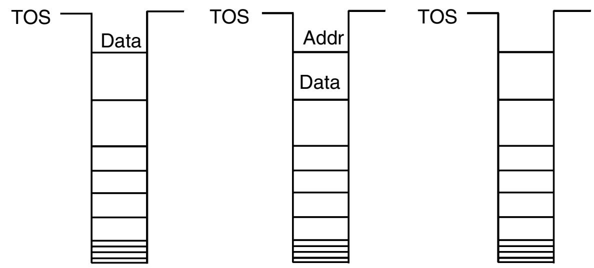
The MARC4 peripheral modules are I/O-mapped by using an IN or OUT instruction which in turn either inputs or outputs a 4-bit data or from one of 16 direct accessible port addresses.

Before an OUT instruction is executed the port destination address and the data to be transmitted must be pushed onto the Expression Stack.

**Example:**

```
: TurnLED_Off
   8 Port4 OUT
;
```

***Figure 1-11.*** OUT Instruction - Stack Effects

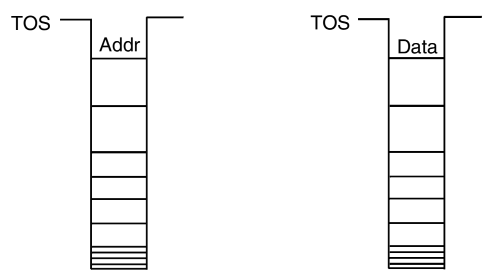



In the case of an IN instruction only the port address needs to be pushed onto the Expression Stack.

**Example:**

```
: KeyPressed?
   KeyIn IN
;
```

***Figure 1-12.*** IN Instruction - Stack Effects




For more complex peripherals please refer to the corresponding data sheets and the supplied hardware programming routines.

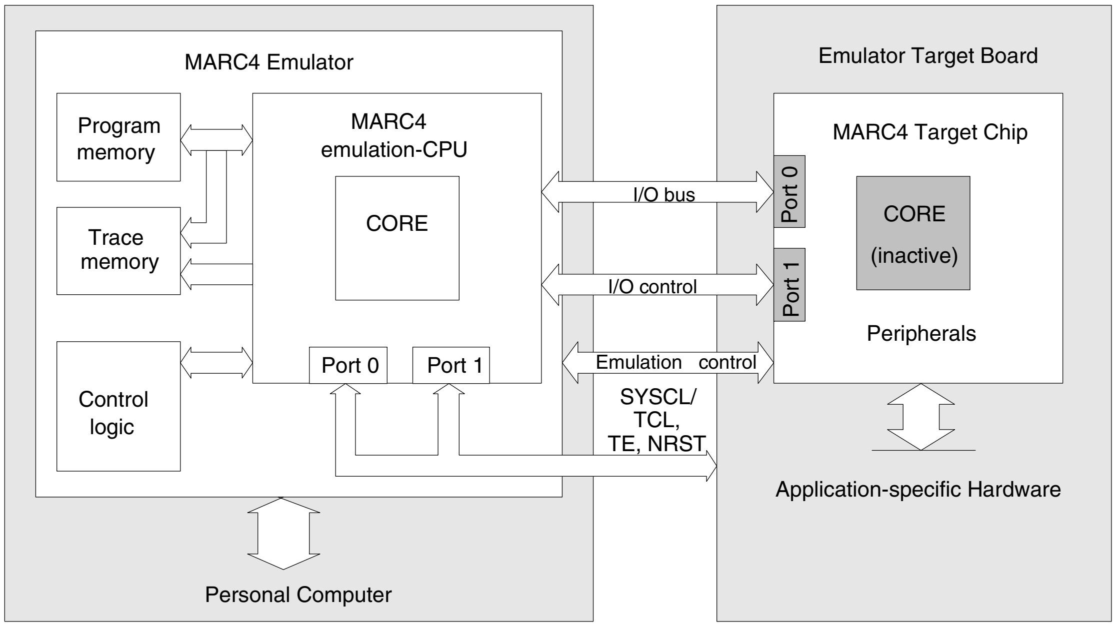
## 1.8 Emulation

The basic function of emulation is to test and evaluate the customer's program and hardware in real time. This therefore enables 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.

***Figure 1-13.*** MARC4 Emulation



# Section 2

# Instruction Set

## 2.1 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.

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 directly accessible RAM address space of up to 256 × 4 bits.

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 4 K × 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.

***Figure 2-1.*** MARC4 Opcode Formats

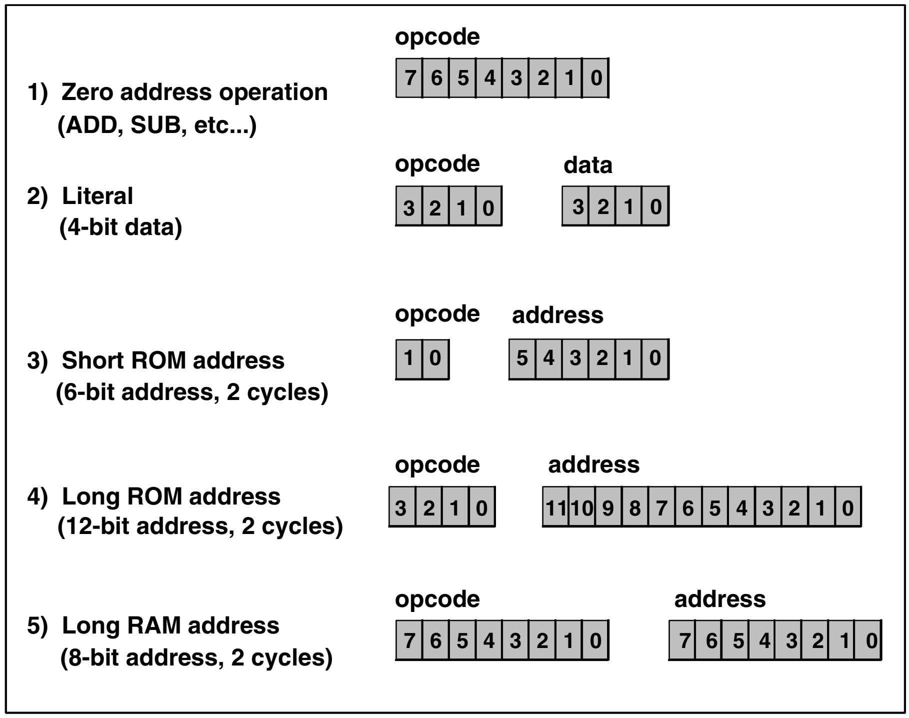


***Table 2-1.*** Instruction Set Overview

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **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” | | | | | |

### 2.1.1 Descripion of Identifiers and Abbreviations Used

| | |
|---|---|
| **n1 n2 n3** | Three nibbles on the Expression Stack |
| **n3n2n1** | Three nibbles on the Return Stack which combine to form a 12-bit word |
| **un2n1** | Two nibbles on the Return Stack (i.e. **DO** loop index and limit), “u” is an unused (undefined) nibble on the Return Stack |
| **/n** | 1's complement of the 4-bit word n |
| **3210** | Numbered bits within a 4-bit word |
| **$xx** | 8-bit hexadecimal RAM address |
| **$xxx** | 2-bit hexadecimal ROM address |
| **PC** | Program Counter (12 bits) |
| **SP** | Expression Stack Pointer (8 bits), the RAM Address Register which points to the RAM location containing the second nibble (TOS-1) 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 |
| **X** | RAM Address Register (8 bits) |
| **Y** | RAM Address Register Y (8 bits), these registers can be used in 3 different addressing modes (direct, pre-incremented or post-decremented addressing) |
| **TOS** | **T**op **o**f (Expression) **S**tack (4 bits) |
| **CCR** | Condition Code Register (4 bits) which contains: |
| **I [bit 0]** | Interrupt-enable flag |
| **B [bit 1]** | Branch flag |
| **% [bit 2]** | Reserved (currently unused) |
| **C [bit 3]** | Carry flag |
| **/C** | NOT Carry (Borrow) flag |

### 2.1.2 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 (nibble) |
| **d** | 8-bit data value (byte) |
| **addr** | 8-bit RAM address |
| **ROMAddr** | 12-bit ROM address |

***Table 2-2.*** Instruction Set

| 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 -- n1+n2 )<br>If overflow<br>then B:=C:=1<br>else B:=C:=0 | **1** | **x x x -** |
| **01** | ADDC | Add with carry the top 2 stack digits | E ( n1 n2 -- n1+n2+C )<br>If overflow<br>then B:=C:=1<br>else B:=C:=0 | **1** | **x x x -** |
| **02** | SUB | 2's complement subtraction of the top 2 digits | E ( n1 n2 -- n1+/n2+1 )<br>If overflow<br>then B:=C:=1<br>else B:=C:=0 | **1** | **x x x -** |
| **03** | SUBB | 1's complement subtraction of the top 2 digits | E ( n1 n2 -- n1+/n2+/C )<br>If overflow<br>then B:=C:=1<br>else B:= C:=0 | **1** | **x x x -** |
| **04** | XOR | Exclusive-OR top 2 stack digits | E ( n1 n2 -- n1 XOR n2 )<br>If result=0 then B:=1<br>else B:=0 | **1** | **- x x -** |
| **05** | AND | Bitwise-AND top 2 stack digits | E ( n1 n2 -- n1 AND n2 )<br>If result=0 then B:=1<br>else B:=0 | **1** | **- x x -** |
| **06** | CMP_EQ | Equality test for top 2 stack digits | E ( n1 n2 -- n1 )<br>If n1=n2 then B:=1<br>else B:=0 | **1** | **x x x -** |
| **07** | CMP_NE | Inequality test for top 2 stack digits | E ( n1 n2 -- n1 )<br>If n1<>n2 then B:=1<br>else B:=0 | **1** | **x x x -** |
| **08** | CMP_LT | Less-than test for top 2 stack digits | E ( n1 n2 -- n1 )<br>If n1<n2 then B:=1<br>else B:=0 | **1** | **x x x -** |
| **09** | CMP_LE | Less-or-equal for top 2 stack digits | E ( n1 n2 -- n1 )<br>If n1<<=n2 then B:=1<br>else B:=0 | **1** | **x x x -** |
| **0A** | CMP_GT | Greater-than for top 2 stack digits | E ( n1 n2 -- n1 )<br>If n1>n2 then B:=1<br>else B:=0 | **1** | **x x x -** |
| **0B** | CMP_GE | Greater-or-equal for top 2 stack digits | E ( n1 n2 -- n1 )<br>If n1>=n2 then B:=1<br>else B:=0 | **1** | **x x x -** |
| **0C** | OR | Bitwise-OR top 2 stack digits | E ( n1 n2 -- n1 OR n2 )<br>If result=0 then B:=1<br>else B:=0 | **1** | **- x x -** |
| **0D** | CCR@ | Copy condition code onto TOS | E ( -- n ) R ( -- ) | **1** | **- - - -** |
| **0E** | CCR! | Restore condition codes | E ( n -- ) R ( -- ) | **1** | **x x x x** |
| **0F** | SLEEP | CPU in "sleep mode", interrupts enabled | E ( -- ) R ( -- )<br>I:=1 | **1** | **- x - 1** |

*Table 2-2.* Instruction Set (Continued)

| Code [hex] | Mnemonic | Operation | Symbolic Description [Stack Effects] | Instr. Cycles | Flags C % B I |
|---|---|---|---|---|---|
| **10** | SHL | Shift TOS left into carry | C<--197>3210<--0<br>B:=C:=MSB | **1** | **x x x -** |
| **11** | ROL | Rotate TOS left through carry | ..<--C<--3210<--C<--..<br>B:=C:=MSB | **1** | **x x x -** |
| **12** | SHR | Shift TOS right into carry | 0-->3210-->C<br>B:=C:=LSB | **1** | **x x x -** |
| **13** | ROR | Rotate TOS right through carry | ..-->C-->3210-->C-->..<br>B:=C:=LSB | **1** | **x x x -** |
| **14** | INC | Increment TOS | E ( n -- n+1 )<br>If result=0 then B:=1<br>else B:=0 | **1** | **- x x -** |
| **15** | DEC | Decrement TOS | E ( n -- n-1 )<br>If result=0 then B:=1<br>else B:=0 | **1** | **- x x -** |
| **16** | DAA | Decimal adjust for addition (in BCD arithmetic) | If TOS>9 OR C=1<br>then E ( n -- n+6 )<br>B:=C:=1<br>else E ( n -- n) R (--)<br>B:=C:=0 | **1** | **1 x 1 -**<br>**0 x 0 -** |
| **17** | NOT | 1’s complement of TOS | E ( n -- /n )<br>If result=0 then B:=1<br>else B:=0 | **1** | **- x x -** |
| **18** | TOG_BF | Toggle Branch flag | If B = 1 then B:=0<br>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 ( -- )<br>I:=0 | **1** | **- x - 0** |
| **1B** | IN | Read data from 4-bit I/O port | E ( port -- n )<br>If port=0 then B:=1<br>else B:=0 | **1** | **- x x -** |
| **1C** | DECR | Decrement index on Return Stack | R ( uun -- uun-1 )<br>If n-1=0 then B:=0<br>else B:=1 | **2** | **- 1 0 -**<br>**- 0 1 -** |
| **1D** | RTI | Return from interrupt routine; enable all interrupts | E ( -- ) R ( $xxx -- )<br>PC := $xxx | **2** | **- - - -** |
| **1E** | SWI | Software interrupt | E ( n1 n2 -- ) R ( -- )<br>[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**<br>**21** | TABLE | Fetch an 8-bit ROM constant and performs an EXIT to Ret_PC | E ( -- nh nl )<br>R ( Ret_PC ROM_addr -- )<br>PC:= Ret_PC | **3** | **- - - -** |
| **22** | >R | Move (loop) index onto Return Stack | E ( n -- )<br>R ( -- uun) | **1** | **- - - -** |
| **23** | I<br>R@ | Copy (loop) index from the Return Stack onto TOS | E ( -- n )<br>R ( uun -- uun) | **1** | **- - - -** |
| **24**<br>**25** | EXIT | Return from subroutine (“ ; ”) | E ( -- ) R ( $xxx --)<br>PC:=$xxx | **2** | **- - - -** |

***Table 2-2.*** Instruction Set (Continued)

| Code [hex] | Mnemonic | Operation | Symbolic Description [Stack Effects] | Instr. Cycles | Flags C % B I |
|---|---|---|---|---|---|
| **26** | SWAP | Exchange the top 2 digits | E ( n1 n2 -- n2 n1 )<br>R ( -- ) | **1** | **- - - -** |
| **27** | OVER | Push a copy of TOS-1 onto TOS | E ( n1 n2 -- n1 n2 n1 )<br>R ( -- ) | **1** | **- - - -** |
| **28** | 2>R | Move top 2 digits onto Return Stack | E ( n1 n2 -- )<br>R ( -- un1n2 ) | **3** | **- - - -** |
| **29** | 3>R | Move top 3 digits onto Return Stack | E ( n1 n2 n3 -- )<br>R ( -- n1n2n3 ) | **4** | **- - - -** |
| **2A** | 2R@ | Copy 2 digits from Return to Expression Stack | E ( -- n1 n2 )<br>R ( un1n2 -- un1n2 ) | **2** | **- - - -** |
| **2B** | 3R@ | Copy 3 digits from Return to Expression Stack | E ( -- n1 n2 n3 )<br>R ( n1n2n3 -- n1n2n3 ) | **4** | **- - - -** |
| **2C** | ROT | Move third digit onto TOS | E ( n1 n2 n3 -- n2 n3 n1 )<br>R ( -- ) | **3** | **- - - -** |
| **2D** | DUP | Duplicate the TOS digit | E ( n -- n n )<br>R ( -- ) | **1** | **- - - -** |
| **2E** | DROP | Remove TOS digit from the Expression Stack | E ( n -- )<br>R ( -- )<br>SP:=SP-1 | **1** | **- - - -** |
| **2F** | DROPR | Remove one entry from the Return Stack | E ( -- )<br>R( uuu -- )<br>RP:=RP-4 | **1** | **- - - -** |
| **30** | [X]@ | Indirect fetch from RAM addressed by the X register | E ( -- n )<br>R ( -- )<br>X:=X Y:=Y | **1** | **- - - -** |
| **31** | [+X]@ | Indirect fetch from RAM addressed by the pre-incremented X register | E ( -- n )<br>R ( -- )<br>X:=X+1 Y:=Y | **1** | **- - - -** |
| **32** | [X-]@ | Indirect fetch from RAM addressed by the post-decremented X register | E ( -- n )<br>R ( -- )<br>X:=X-1 Y:=Y | **1** | **- - - -** |
| **33 xx** | [>X]@ $xx | Direct fetch from RAM addressed by the X register | E ( -- n )<br>R ( -- )<br>X:=$xx Y:=Y | **2** | **- - - -** |
| **34** | [Y]@ | Indirect fetch from RAM addressed by the Y register | E ( -- n )<br>R ( -- )<br>X:=X Y:=Y | **1** | **- - - -** |
| **35** | [+Y]@ | Indirect fetch from RAM addressed by the pre-incremented Y register | E ( -- n )<br>R ( -- )<br>X:=X Y:=Y+1 | **1** | **- - - -** |
| **36** | [Y-]@ | Indirect fetch from RAM addressed by the post-decremented Y register | E ( -- n )<br>R ( -- )<br>X:=X Y:=Y-1 | **1** | **- - - -** |
| **37 xx** | [>Y]@ $xx | Direct fetch from RAM addressed by the Y register | E ( -- n )<br>R ( -- )<br>X:=X Y:=$xx | **2** | **- - - -** |
| **38** | [X]! | Indirect store into RAM addressed by the X register | E ( n -- ) R ( — ) X:=X Y:=Y | **1** | **- - - -** |

***Table 2-2.*** Instruction Set (Continued)

| Code [hex] | Mnemonic | Operation | Symbolic Description [Stack Effects] | Instr. Cycles | Flags C % B I |
|---|---|---|---|---|---|
| **39** | [+X]! | Indirect store into RAM addressed by the pre-incremented X register | E ( n -- )<br>R ( -- )<br>X:=X+1 Y:=Y | **1** | - - - - |
| **3A** | [X-]! | Indirect store into RAM addressed by the post-decremented X register | E ( n -- )<br>R ( -- )<br>X:=X-1 Y:=Y | **1** | - - - - |
| **3B xx** | [>X]! $xx | Direct store into RAM addressed by the X register | E ( n -- )<br>R ( -- )<br>X:=$xx Y:=Y | **2** | - - - - |
| **3C** | [Y]! | Indirect store into RAM addressed by the Y register | E ( n -- )<br>R ( -- )<br>X:=X Y:=Y | **1** | - - - - |
| **3D** | [+Y]! | Indirect store into RAM addressed by the pre-incremented Y register | E ( n -- )<br>R ( -- )<br>X:=X Y:=Y+1 | **1** | - - - - |
| **3E** | [Y-]! | Indirect store into RAM addressed by the post-decremented Y register | E ( n -- )<br>R ( -- )<br>X:=X Y:=Y-1 | **1** | - - - - |
| **3F xx** | [>Y]! $xx | Direct store into RAM addressed by the Y register | E ( n -- )<br>R ( -- )<br>X:=X Y:=$xx | **2** | - - - - |
| **70** | SP@ | Fetch the current Expression Stack Pointer | E ( -- SPh SPl+1 )<br>R ( -- )<br>SP:=SP+2 | **2** | - - - - |
| **71** | RP@ | Fetch the current Return Stack Pointer | E ( -- RPh RPl )<br>R ( -- ) | **2** | - - - - |
| **72** | X@ | Fetch the current X register contents | E ( -- Xh Xl)<br>R ( -- ) | **2** | - - - - |
| **73** | Y@ | Fetch the current Y register contents | E ( -- Yh Yl )<br>R ( -- ) | **2** | - - - - |
| **74** | SP! | Move the address into the Expression Stack Pointer | E ( dh dl -- ? )<br>R ( -- )<br>SP:=dh_dl | **2** | - - - - |
| **75** | RP! | Move the address into the Return Stack Pointer | E ( dh dl -- )<br>R ( -- ? )<br>RP:=dh_dl | **2** | - - - - |
| **76** | X! | Move the address into the X register | E ( dh dl -- ) R ( -- )<br>X:=dh_dl | **2** | - - - - |
| **77** | Y! | Move the address into the Y register | E ( dh dl -- ) R ( -- )<br>Y:=dh_dl | **2** | - - - - |
| **78 xx** | >SP $xx | Set the Expression Stack Pointer | E ( -- ) R ( -- )<br>SP:=$xx | **2** | - - - - |
| **79 xx** | >RP $xx | Set the return Stack Pointer direct | E ( -- ) R ( -- )<br>RP:=$xx | **2** | - - - - |
| **7A xx** | >X $xx | Set the RAM address register X direct | E ( -- ) R ( -- )<br>X:=$xx | **2** | - - - - |
| **7B xx** | >Y $xx | Set the RAM address register Y direct | E ( -- ) R ( -- )<br>Y:=$xx | **2** | - - - - |

***Table 2-2.*** Instruction Set (Continued)

| Code [hex] | Mnemonic | Operation | Symbolic Description [Stack Effects] | Instr. Cycles | Flags C % B I |
|---|---|---|---|---|---|
| **7C** | NOP | No operation | PC:=PC+1 | **1** | **- - - -** |
| **7D..7F** | NOP | Illegal instruction | PC:=PC+1 | **1** | **- - - -** |
| **4x xx** | CALL $xxx | Unconditional long CALL | E ( -- ) R ( -- PC+2 )<br>PC:=$xxx | **3** | **- - - -** |
| **5x xx** | BRA $xxx | Conditional long branch | If B=1 then PC:=$xxx<br>else PC:=PC+1 | **2** | **- - 1 -**<br>**- - 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<br>else PC:=PC+1 | **2** | **- - - -**<br>**- - - -** |
| **C0..FF** | SCALL $xxx | Unconditional short CALL | E ( -- ) R ( -- PC+1 )<br>PC:= $xxx | **2** | **- - - -** |

*Table 2-3.* 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 |
| **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 |

***Table 2-3.*** MARC4 Instruction Set Overview (Continued)

| Mnemonic | Description | Cycles/Bytes |
|---|---|---|
| **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 |
| **Stack Operations** | | |
| SWAP | Exchange the top 2 nibbles | 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 nibbles onto the Return Stack | 3/1 |
| 3>R | Move the top 3 nibbles 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 the Return Stack (12-Bit) | 1/1 |
| LIT_n | Push immediate value (1 nibble) onto TOS | 1/1 |
| **ROM Data Operations** | | |
| TABLE | Fetch 8-bit constant from ROM | 3 |
| **Memory Operations** | | |
| [X]@<br>[Y]@ | Fetch 1 nibble from RAM indirectly addressed by X- or Y-register | 1/1 |
| [+X]@<br>[+Y]@ | Fetch 1 nibble from RAM indirectly addressed by pre-incremented X- or Y-register | 1/1 |
| [X–]@<br>[Y–]@ | Fetch 1 nibble from RAM indirectly addressed by post-decremented X- or Y- register | 1/1 |
| [>X]@ $xx<br>[>Y]@ $xx | Fetch 1 nibble from RAM directly addressed by X- or Y-register | 2/2 |
| [X]!<br>[Y]! | Store 1 nibble into RAM indirectly addressed by [X] | 1/1 |
| [+X]!<br>[+Y]! | Store 1 nibble into RAM indirectly addressed by pre-incremented [X] | 1/1 |

*Table 2-3.* MARC4 Instruction Set Overview (Continued)

| Mnemonic | Description | Cycles/Bytes |
|---|---|---|
| [X–]!<br>[Y–]! | Store 1 nibble into RAM indirectly addressed by post-decremented X- or Y- register | 1/1 |
| [>X]! $xx<br>[>Y]! $xx | Store 1 nibble into RAM directly 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 |

# Section 3

# Programming in qFORTH

## 3.1 Language 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**
- **Re-entrant: Different Service Routines can Share the Same Code, as Long as Global Variables are Not Modified within this 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**

## 3.2 Why Program in qFORTH

**Programming in qFORTH reduces the software development time!**

Atmel'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. The MARC4 software development system enables the MARC4 programmer to edit, compile, simulate and/or evaluate a program code using an integrated package with predefined key codes and pull-down menus. The compiler-generated MARC4 code is optimized for demanding application requirements, such as the 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 the development cost. Programs that are developed using the Brute Force approach (where the program is realized in software using a sequential code) tend to be considerably larger in memory consumption, and are extremely difficult to maintain.

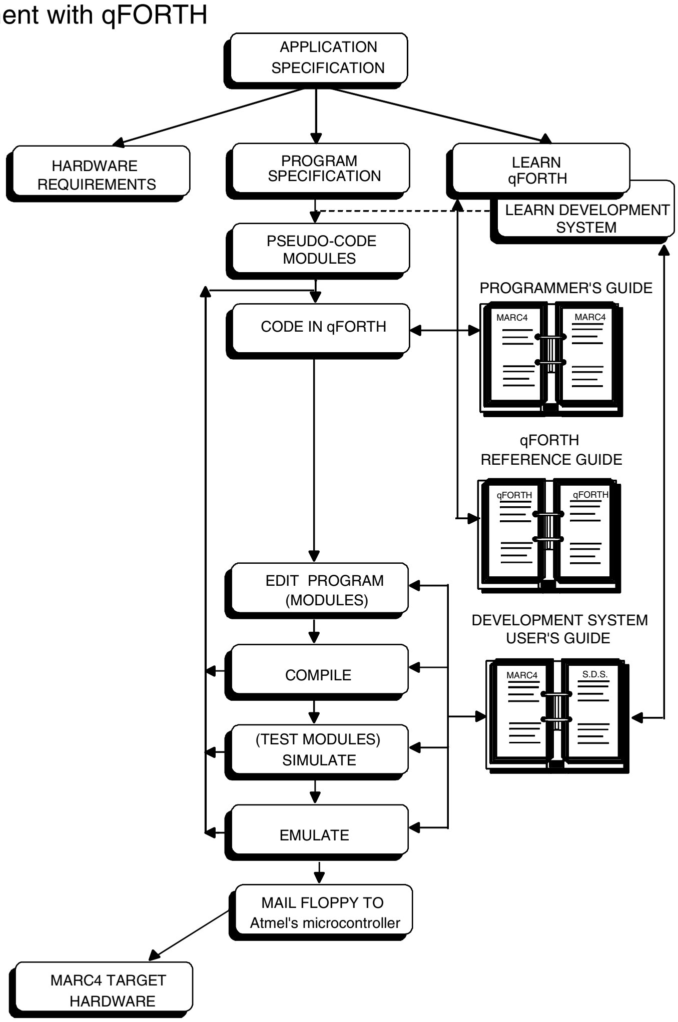
A qFORTH program, engineered using the building block modular approach is compact in size, easy to understand and thus, easier to maintain. The added benefit for the user is a library of software routines which can be interchanged with other MARC4 applications as long as the input and output conditions 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 required. Programming in qFORTH results in a re-usable code. Re-usable for other applications which will be programmed at a later date. This is an important factor in ensuring that future software development costs are kept to a minimum. Routines written by one qFORTH programmer can be easily incorporated by a different qFORTH user.

## 3.3 Language Overview

qFORTH is based on the FORTH-83 language standard, the qFORTH compiler generates a native code for a 4-bit FORTH-architecture single-chip microcomputer - Atmel's MARC4.

MARC4 applications are all programmed in qFORTH which is designed specifically for efficient real-time control. Since the qFORTH compiler generates highly optimized codes, 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 peephole optimizations.

***Figure 3-1.*** 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 3-1.*** 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 3-2.*** Differences between qFORTH and FORTH-83

| **Arithmetic/Logical** | **Stack Operations** |
|---|---|
| 4-bit Expression Stack<br>12-bit Return Stack<br>The prefix "2" on a keyword (e.g. 2DUP refers to an 8-bit data type)<br>Branch and Carry flag in the Condition Code Register<br>Only predefined data types for handling untyped memory blocks, arrays or tables of constants | 16-bit Expression Stack<br>16-bit Return Stack<br>The prefix "2" on a keyword (e.g. 2DUP refers to a 32-bit data type)<br>Flag value on top of the Expression Stack<br>CREATE, >BUILD .. DOES |

## 3.4 The qFORTH Vocabulary

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.

### 3.4.1 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:

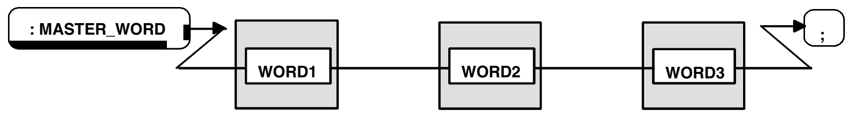
```
: MASTER-WORD WORD1 WORD2 WORD3 ;
```

The colon **":"** and the semicolon **";"** are the start and stop declarations for the definition. A qFORTH programmer refers to a colon definition to specify a word name which follows the colon. The following diagram depicts the execution sequence of these three words:

The sequential order shows the way the compiler (and the MARC4) will understand what the 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).
**4th step** Specify the end of the colon definition with a semicolon.

***Figure 3-2.*** Threaded qFORTH Word Definition




## 3.5 Stacks, RPN and Comments

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

### 3.5.1 Reverse Polish Notation

qFORTH is a **R**everse **P**olish **N**otation language (**RPN**), which 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 on 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.

### 3.5.2 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 bits wide and is used for temporary storage during calculations or to pass parameters to words.

The **Return Stack** is 12 bits 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.

### 3.5.3 Stack Notation

The qFORTH stack notation shows the stack contents before and after the execution of a qFORTH word. The before and after operations are separated 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:

*Table 3-3.* Stack Notation

| Before Side | After Side | Example | Stack Notation |
|---|---|---|---|
| ( n3 n2 n1 --<br>↑<br>TOS | n3 n2 n1)<br>↑<br>TOS | 4 2 1<br>+<br>SWAP | ( -- 4 2 1 )<br>( 4 2 1 -- 4 3 )<br>( 4 3 -- 3 4 ) |

### 3.5.4 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 closing the 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 a blank space to follow the comment declaration.

*Table 3-4.* Comments

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

## 3.6 Constants and Variables

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

### 3.6.1 Constants

A constant is an unalterable qFORTH definition. 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.

*Table 3-5.* Constant Definitions

| 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)
```

#### 3.6.1.1 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 **4.0 K**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 at 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        Freq EXTERNAL
: Check_Freq  Freq [4]  2@   80h  D>
              IF 0 0 Frequency [5] 2!
              THEN
;
```

### 3.6.2 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 each delimited by a space and a comma.

The content of a table is not limited to literals such as 5 or 67h, but may also include user- or pre-defined 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.

***Table 3-6.*** Table Definitions

| qFORTH Table Definitions | |
|---|---|
| ROMCONST <table-name> | Const , Const , Const ,<br>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 " ,
```

Notes: 1. A comma must follow the last table item.
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).

## 3.7 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 also permits a double-length 8-bit value to be assigned as a **2VARIABLE**.

***Table 3-7.*** Variable Definitions

| qFORTH Variable Definitions | |
|---|---|
| VARIABLE <variable-name> | 4-bit variable |
| 2VARIABLE <variable-name> | 8-bit variable |

**Example:**

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

### 3.7.1 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:

```
VARIABLE DATA 7 ALLOT
```

or using the qFORTH array implementation:

```
8 ARRAY DATA
```

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

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

***Table 3-8.*** Array Definitions

| 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 |

## 3.8 Stack Allocation

Both the Expression and Return Stacks are located in RAM. The size of the stacks is variable and must be defined by the programmer by using the predefined variables **R0** and **S0**.
Figure 3-3 shows the location of the stacks in RAM. 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 is 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:

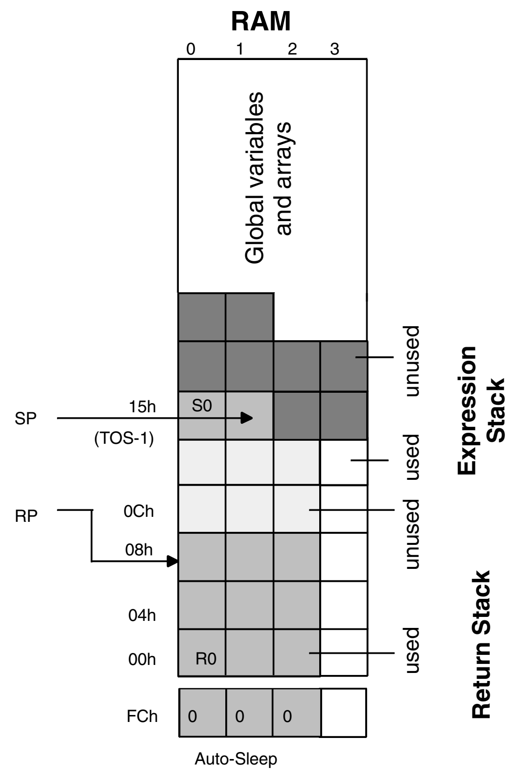
**RET_Value := RET_Depth × 4**

**Example:**

```
VARIABLE R0 20 ALLOT \ RET Depth of 5
VARIABLE S0 17 ALLOT \ EXP Depth of 17
```

### 3.8.1 Stack Pointer Initialization

***Figure 3-3.*** Stacks Inside RAM




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

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

**Example:**

```
VARIABLE R0 32 ALLOT \ RET stack depth = 8
VARIABLE S0 12 ALLOT \ EXP stack depth = 12 nibbles
: $RESET
    >RP   FCh           \ Initialize the two stack pointers
    >SP   S0
    RAM_Test
      ...
;
```

## 3.9 Stack Operations, Reading and Writing

### 3.9.1 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"**. Stack operators used most often and which manipulate the order of the elements on the Data Stack like **DUP, DROP, SWAP, OVER** and **ROT** are explained later on.

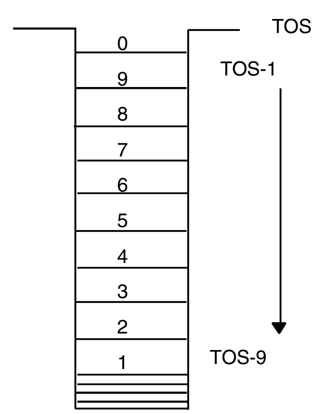
#### 3.9.1.1 Data Stack

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, so 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 3-4.*** Push-down Data Stack



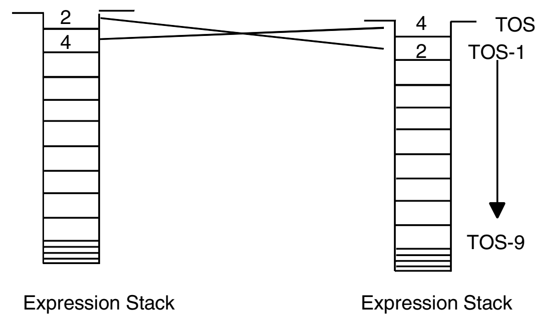

When executed, the value **0** at the top and the value **1** at the bottom of the Expression Stack will be the result.

### 3.9.1.2 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 reserve order.

**4 2 SWAP ( 4 2--2 4)**

***Figure 3-5.*** The SWAP Operation




### 3.9.1.3 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 )
```

### 3.9.1.4 ROT and <ROT

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

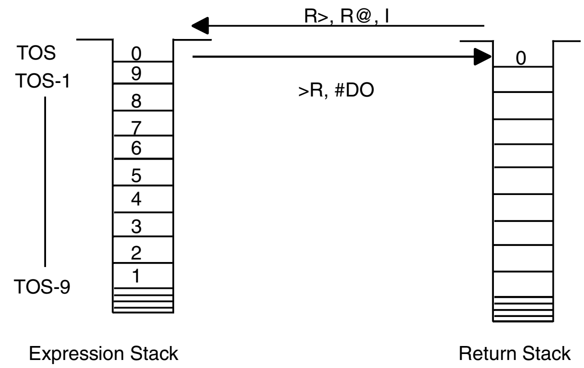
The **ROT** operation moves the third value (TOS-2) to the TOS. The operation **<ROT** (which is the same as **ROT ROT**) does the opposite of **ROT**, moving the value from the TOS to the TOS-2 location on the Expression Stack.

### 3.9.1.5 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 3-6.*** Return Stack Data Transfers




### 3.9.1.6 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 section 4.6 “The qFORTH Language - 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 | ?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 -- )<br>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 position |
| 2R> | EXP ( -- n1 n2 )<br>RET ( u\| n2\| n1 -- ) | Move top 8 bits from Return to Expression Stack |
| 2R@ | EXP ( -- n1 n2 )<br>RET(u\| n2\| n1--u\| n2\| n1) | Copy top 8 bits from Return to Expression Stack |
| 3>R | EXP ( n1 n2 n3 -- )<br>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 )<br>RET ( n3\| n2\| n1 -- ) | Move top 3 nibbles from Return to the Expression Stack |
| 3R@ | EXP ( -- n1 n2 n3 )<br>RET<br>( n3\| n2\| n1 -- n3\| n2\| n1 ) | Copy 3 nibbles (1 ROM address entry) from the Return Stack to the Expression Stack |

### 3.9.2 Reading and Writing (@, !)

In the previous section it was mentioned that data can be placed onto, and taken off 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 the TOS to contain the variable's 8-bit RAM address and that the data to be stored in the RAM be contained at the TOS-2 location.

The read operator 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.

To 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 its 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 loads them onto the Expression Stack. The lower-order nibble will always end up on TOS.

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

### 3.9.3 Low-level Memory Operations

#### 3.9.3.1 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 organize arrays within the RAM. They can be pre-incremented or post-decremented by using CPU control.

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. If a default occurs, the compiler uses the Y register.

***Table 3-9.*** Memory Operators which Use the X/Y Register

| Memory Operators which Use the X/Y 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+!
```

***Table 3-10.*** Low Level Memory Operation

| 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) register | Y! |
| >X xx | Set register address of X (or Y) directly | >Y yy |
| [>X]@ xx | Directly RAM fetch, X (or Y) addressed | [>Y]@ yy |
| [>X]! xx | Directly RAL store, X (or Y) addressed | [>Y]! yy |
| [X]@ | Indirectly X (or Y) fetch of RAM contents | [Y]@ |
| [X]! | Indirectly X (or Y) store of RAM contents | [Y]! |
| [+X]@ | Pre-increment X (or Y) indirect RAM fetch | [+Y]@ |
| [X-]@ | Post-decrement X (or Y) indirect RAM fetch | [Y-]@ |
| [+X]! | Pre-increment X (or Y) indirect RAM store | [+Y]! |
| [X-]! | Post-decrement X (or Y) indirect RAM store | [Y-]! |

#### 3.9.3.2 Bit Manipulations in RAM

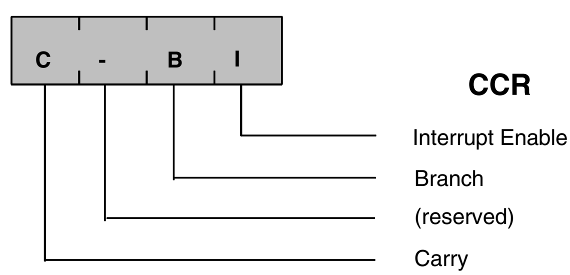
By using the X or Y registers, it is possible to manipulate the content of the RAM on a bit-wise basis. The following examples all have the same stack notation.

```
: 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
```

## 3.10 MARC4 Condition Codes

The MARC4 processor has within its **A**rithmetic **L**ogic **U**nit (ALU) a 4-bit wide **C**ondition **C**ode **R**egister (CCR) which contains 4 flag bits. These are the Branch (**B**) flag, the Interrupt-Enable (**I**) flag and the Carry (**C**) flag.

***Figure 3-7.*** MARC4 Condition Code Register Flags




Most arithmetic/logical operations, for example, will have an effect on the CCR. If you try to add **12** and **5**, the Carry and Branch flags will be set, since an arithmetic overflow has occurred.

### 3.10.1 CCR and 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 is also used during a 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 has been 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, it will be reset following the execution.

The **SET_BCF** instruction will set the Branch and Carry on execution, while the **CLR_BCF** operation will reset both flags.

## 3.11 Arithmetic Operations

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

### 3.11.1 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 )
```

#### 3.11.1.1 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, which are defined with a **3** prefix (e.g: **3DROP**). Examples for all qFORTH dictionary words are included in section 4 “qFORTH Language Dictionary”**.**

### 3.11.2 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, this expression as **4 2 +**. The 4 is deposited onto the Data Stack, followed by the 2. The operator gives a command 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 in a similar way 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 --)
;
```

### 3.11.3 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 becomes zero.

### 3.11.4 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!
```

### 3.11.5 BCD Arithmetic

#### 3.11.5.1 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.

**3.11.5.1.1 DAA**

Decimal adjust for BCD arithmetic, adds 6 to values between 10 and 15. It will also add 6 to the TOS, if the carry flag is 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 )**

**3.11.5.1.2 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
;
```

### 3.11.6 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 )** | Subtract 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 )** | Divide n from d giving 4-bit results |
| **D/MOD** | **( d n -- d_quot n_rem )** | Divide 8-bit value & 4-bit remainder |
| **TD+!** | **( d addr -- )** | Add 8-bit TOS to 12-bit RAM var. |
| **TD-!** | **( d addr -- )** | Subtract 8-bit from 12-bit RAM var. |
| **TD+** | **( d addr -- t )** | Add 8-bit to 12-bit RAM var. |
| **TD-** | **( d addr -- t )** | Subtract 8-bit from 12-bit RAM var. |
| **D->BCD** | **( d -- n_100 n_10 n_1 )** | Convert 8-bit binary to BCD |

## 3.12 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.

### 3.12.1 Logical Operators

The truth table shown below is the standard table used to represent the effects of the logical operators on two data values (**n1** and **n2**).

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 section 4 “qFORTH Language Dictionary”.

The stack notation for all logical qFORTH words is:

**EXP ( n1 n2 -- n3 )**

***Table 3-11.*** Logical Operations

| 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 |

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 --     )
;
```

#### 3.12.1.1 TOGGLE

The TOGGLE operation is classified in the section 4 “qFORTH Language Dictionary” as belonging to the set of memory operations. Although this is true, the **TOGGLE** and its relative, the **DTOGGLE**, are both 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:**

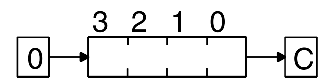
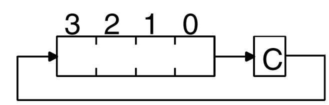
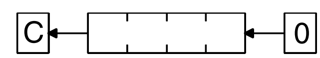
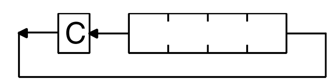
```
VARIABLE LED_Status
: Toggle-LED
     0001b LED_Status TOGGLE ( toggles bit 0 only )
;
```

#### 3.12.1.2 SHIFT and ROTATE Operations

The MARC4 instruction set contains two shift and two rotate instructions which are shown in Table 3-12. 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, and cause the Carry and Branch flags to be altered. When using these instructions, it is advisable to set or reset the flags within your initialization routine, using either the **SET_BCF** or the **CLR_BCF** instructions.

***Table 3-12.*** Shift and Rotate Instructions

| Mnemonic | Description | Function |
|---|---|---|
| SHR<br>2/ | Shift TOS right into Carry |  |
| ROR | Rotate TOS right through Carry |  |
| SHL<br>2* | Shift TOS left into Carry |  |
| ROL | Rotate TOS left through Carry |  |

**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 ;
```

## 3.13 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 - )`

### 3.13.1 < , >

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 where 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.

### 3.13.2 <= , >=

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 are 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 and subsequently 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-] )
```

### 3.13.3 <> , =

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, are dropped off the stack after the operation has been executed, except if 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 so far, the contents of the TOS and TOS-1 previous to the operations are dropped from the stack.

### 3.13.4 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 then 16 are assumed to be 4-bit values.

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

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

## 3.14 Control Structures

The control structures presented here can be divided into two categories: Selection and looping. The Table 3-13 and Table 3-14 compare qFORTH's control structures will those found in PASCAL.

As the comparison of the two languages shows, qFORTH offers a rich variety of structures which enable your program to branch to different code segments within the program.

***Table 3-13.*** qFORTH Selection Control Structures

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

***Table 3-14.*** qFORTH Loop Control Structures

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

### 3.14.1 Selection Control Structures

The code to be executed is dependent on a specific condition. This condition can be indicated by setting 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.

#### 3.14.1.1 IF .. THEN

The **IF .. THEN** construct is a conditional phrase permitting the sequence of program statements to be executed dependent on 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 if the values given on TOS as a parameter to the word are greater than 9.

First the current TOS value is duplicated. 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. The qFORTH words following the **IF** will therefore be executed. In the first example the TOS value will be dropped and replaced by the value 1.

#### 3.14.1.2 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
            THEN THEN THEN THEN
     DROP ;
```

In the word “2BIT-TEST”, the TOS is checked to see if it contains one of three possible values. If either one 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 a qFORTH code which is more 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** selectors are not limited to constants (e.g. high-score @).

```
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 ;
```

### 3.14.2 Loops, Branches and Labels

#### 3.14.2.1 Definite Loops

The **DO .. LOOP** control structure is an example of a definite loop. The number of times the loop is 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** starts 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 compares 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 loop constructs.

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

#### 3.14.2.2 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 Port 1, then finish the loop.

**Example:**

```
: UNTIL-Example
     3  BEGIN
          DUP Port1 OUT      ( Write the current value to Port 1 )
          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 is then executed until the Branch flag is set (TRUE). The Branch flag is set when  the desired condition (TOS > 9) is met.

The second conditional loop control structure **BEGIN .. WHILE .. REPEAT** repeats a sequence of qFORTH words as long as a condition (computed between **BEGIN** and **WHILE**) is still being met.

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

**Example:**

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

***Table 3-15.*** Indefinite Loops

| qFORTH - Indefinite Loops | |
|---|---|
| **BEGIN** <Condition><br>**WHILE ... REPEAT** | Condition tested at start of loop |
| **BEGIN** ...<Condition> **UNTIL** | Condition tested at end of loop |
| **BEGIN ... AGAIN** | Unconditional loop |

### 3.14.3 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_Labl1:
```

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_Labl1
```

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
;;
```

### 3.14.4 Arrays and Look-up Tables

#### 3.14.4.1 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.)

#### 3.14.4.2 Initializing and Erasing an Array

By using the qFORTH word **ERASE,** it is possible to erase an array's content to be filled with zeros.

#### 3.14.4.3 Array Filling

A third way to initialize an array is using the word **FILL**. **FILL** requires that the beginning address of the array and the size of the array are 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
;
```

#### 3.14.4.4 Looping in an Array

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

#### 3.14.4.5 Moving Arrays

The words **MOVE** and **MOVE>** copy a specified number of digits from one address to another within the RAM. The difference between the two instructions is that **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
;
```

#### 3.14.4.6 Comparing Arrays

The word **"?Arrays="** compares two array fields, starting at the last field element in desending 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 section 4 “qFORTH Language Dictionary”.

### 3.14.5 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 section 4 “qFORTH Language 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 array's 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.

### 3.14.6 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 a vectored execution where a word definition is executed indirectly, this can be performed by placing the address of a definition into a variable. The content of the variable is then EXECUTEd as desired. This gives the user increased flexibility as complicated pointer manipulations can now be performed.

**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                         \ x9:59 -> x+1 0:00
      Time [Hrs_10] 1+!         \ 0 -> 1 or 1 -> 2
     THEN
     Time [Hrs_10] 2@  2 4 D=   \ 24:00:00      ?
     IF                         \ 23:59 -> 00.00
      0 0 Time [Hrs_10] 2!      \ It's midnight
     THEN
;

:    Inc_Hour
     LAP_Timer [Hours] 1+!      \ Inc Hours binary by 1
;                               \ Wrap around at 16:00.00

:    Inc_Min                    \ <Y> = ^Digit[Min_1]
     BCD_+1!                    \ 18:29 -> 18:30
     IF                         \ On overflow ..
      [Y]@  1+   6 CMP_EQ [Y]!  \ 18:59 -> 19:00
      IF
       0 [Y-]!                  \ Reset Min_10
       Hours_Inc 3@ EXECUTE     \ Computed Hrs_Inc '
       [ E 0 R 0 ]
      THEN
     THEN
;

:    Inc_Secs                   \ <Y> = ^Digit[Sec_1]
     BCD_+1!                    \ Increment seconds
     IF                         \ 8:25:19 -> 8:25:20
       [Y]@ 1+  6 CMP_EQ [Y]!
       IF                       \ 8:30:59 -> 8:31:00
          0 [Y-]!               \ Reset Sec_10
          Inc_Min               \ Incr. Minutes
      THEN
     THEN
;
```

```
:    Inc_1/100s
     BCD_+1!                      \ Increment 10_ms
     IF                           \ 25.19.94 -> 25.19.95
        BCD_+1!                   \ Incr. 100_ms
        IF                        \ 30.49.99 -> 30.50.00
           Inc_Secs               \ Incr. seconds ..
       THEN
     THEN
;

:    IncTime                      \ Incr. T.O.D.
     ' Inc_Hrs Hours_Inc [2] 3!   \ Note use of Tick
     Time [Sec_1] Y! Inc_Secs     \ Increment seconds
;

:    Inc_10ms                     \ Incr. LAP timer
     ' Inc_Hour Hours_Inc [2] 3!  \ Note use of TICK
     LAP_Timer [10_ms]   Y!
     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                         \ Real-Time Clock Interrupt
                                    every 1/2s
     1 TimeCount TOGGLE
     IF   DI   IncTime EI  THEN    \ Be on the save side
;

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

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

## 3.15 Making the Best Use of Compiler Directives

Compiler directives allow the programmer to have direct manual control of the generation and placement of program code and RAM variables. The qFORTH compiler will automatically generate an efficient code, so it is not necessary or recommended to manually optimize the application program at the beginning of the project. However, when 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 disk.

### 3.15.1 Controlling ROM Placement

By forcing a zero page placement of the most 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
```

### 3.15.2 Macro Definitions, EXIT and ;;

If fast execution is required, 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 which are replaced by **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. If this is the 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
```

### 3.15.3 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
;
```

### 3.15.4 $INCLUDE Directive

It is common programming practice to split a large program into a number of smaller modules, i.e, 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 of four levels.

**Example:**

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

### 3.15.5 Conditional Compilation

Conditional compilation enables the programmer to control which parts of the program are to be compiled. A typical program under development for example has an extra code to aid debugging. This code is removed on the final version. By using a conditional compilation, the programmer can keep all the debugging information in the source, but generate the 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 Emulation       \ Use EVA prescaler
$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
```

### 3.15.6 Controlling XY Register Optimizations

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!** is optimized to: **>X $pq**

- XY@!

the sequence **>X $pq [X]!** is 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 “On_Time” and “SwitchNr” are stored in consecutive RAM locations.

***Table 3-16.***

| qFORTH Source | Intermediate Code | XYLOAD, XY@! Optimized | Final Code after XYTRACE |
|---|---|---|---|
| On_Time @ | LIT_3 LIT_4 | [>X]@ $On_Time | [>X]@ $On_Time |
| SwitchNr +! | X! [X]@ | [>Y]@ $SwitchNr | [+X]@ |
| | LIT_3 LIT_5 | ADD | ADD |
| | Y! [Y]@ | [Y]! | [X]! |
| | ADD | | |
| | [Y]! | | |
| | 10 Bytes | 6 Bytes | 5 Bytes |

## 3.16 Recommended Naming Conventions

### 3.16.1 How to Pronounce the Symbols

! store
@ fetch
# sharp or “number”
$ dollar
% percent
^ caret
& ampersand
* star
( ) left paren and right paren ; paren
- dash; not
+ plus
= equals
{ } faces or “curly brackets”

[ ] square brackets
" quote
' as prefix: Tick; as suffix: prime
~ tilde
| bar
\ backslash
/ slash
< less-than; left dart
> greater-than; right dart
? question or “query”
, comma
. dot

| Form | Example | Meaning |
|---|---|---|
| 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”.

## 3.17 Literature List

### 3.17.1 Recommended Literature

**"Starting Forth"** is highly recommended as a good general introduction to FORTH, especially chapters 1 to 6.

**"Starting FORTH"** is now also 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

### 3.17.2 Literature of General Interest

The following list shows the spectrum of FORTH literature. This literature is of background interest ONLY and may contain information which is not completely relevant for 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: | I PC 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"** |
| Author: | Kelly & Spier |
| Publisher: | Prentice Hall, 1986 |
| ISBN: | 0-13-326331-2 |

# Section 4

# qFORTH Language Dictionary

## 4.1 Preface

This dictionary is written as a reference guide for 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 flags, X and Y register changes are also described in detail. The length of each instruction is specified by the number of bytes generated at the time of compilation. A short demonstration program for each instruction is also included.

The qFORTH language is described in section 3 “Programming in qFORTH” which includes a language tutorial and learner’s guide. First-time programmers of qFORTH are urged to read this chapter before consulting this guide.

The associated effects and changes of the listed qFORTH word 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 topics in the following sections.

### 4.2.1 Purpose

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

### 4.2.2 Category

A classification of the qFORTH vocabulary entries is given.

All entries in this dictionary are classified in the following categories (the same categories are used in section 4.6 “MARC4 qFORTH Quick Reference Guide”):

A: Usage-specific categories:

#### 4.2.2.1 Arithmetic/Logical

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

#### 4.2.2.2 Comparisons

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

**4.2.2.3 Control Structures**

Control structures are used for conditional branches such as

IF ... ELSE ... THEN and loops (DO ... LOOP).

**4.2.2.4 Interrupt Handling**

The MARC4 instruction set allows the programmer to handle up to 8 hardware/software interrupts and to enable/disable all interrupts. Other qFORTH words permit the programmer to determine the actually used depth or available free space on the Expression and Return Stack.

**4.2.2.5 Memory Operations**

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

**4.2.2.6 Stack Operations**

The sequence of the items and 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:

**4.2.2.7 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.

**4.2.2.8 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 “;” is translated to an EXIT instruction (return from subroutine). At execution time, the program counter is 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.

**4.2.2.9 qFORTH Macro Definitions**

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

**4.2.2.10 Predefined Data Structures**

Predefined data structures do not 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.

**4.2.2.11 Compiler Directives**

Compiler directives are used to include other source files at compilation time, to 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 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.

**4.2.3 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 “:” / “;” for improved understanding of this representation.

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

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

The assembler mnemonic “(S)BRA” means that the compiler tries to optimize all BRA mnemonics to SBRA (short branches * only one byte) if the option is switched on and optimization is possible (page boundaries can not be crossed by the SBRA, but only by the BRA).

### 4.2.4 Stack Effect

This category describes the effects on the Expression and Return Stack when executing the described instruction. See section 4.3 “Stack-related Conventions” to better understand the herein used syntax and semantics.

### 4.2.5 Stack Changes

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

### 4.2.6 Flags

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

### 4.2.7 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 can 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).

### 4.2.8 Bytes Used

This part gives the 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.

### 4.2.9 See Also

This section includes similar qFORTH words or words of the same category. The “%” symbol in this field signifies that there are no similar words for this entry.

### 4.2.10 Example

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

## 4.3 Stack-related Conventions

### 4.3.1 Expression Stack

The 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.

### 4.3.2 Return Stack

The Return Stack contains the subroutine return addresses as well as the 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”.

### 4.3.3 X/Y-registers

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

### 4.3.4 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 flags of the Condition Code Register |

The stack effects shown in the dictionary represent the stack content, 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.
                  ↓          ↓
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 elements | ? | | top elements | 2 | |
| | . | | | . | | | . | |
| | . | | | . | | | . | |

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 possible nibbles 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 Atmel, they are accessible via the 4-bit Condition Code Register - “CCR”.

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

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

| | | | |
|---|---|---|---|
| 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

### 4.5.1 Memory Operations

| | | | | | | |
|---|---|---|---|---|---|---|
| 4-bit variable | address points to | → | n | ←RAM | | |
| 8-bit variable | address points to | → | nh | nl | | |
| 12-bit variable | address points to | → | nh | nm | nl | |

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, 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!         ( initialize 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 The qFORTH Language - Quick Reference Guide

### 4.6.1 Arithmetic/Logical

| | | |
|---|---|---|
| **-** | **EXP ( n1 n2 -- n1–n2 )** | Subtract the top two nibbles |
| **+** | **EXP ( n1 n2 -- n1+n2 )** | Add up the two top 4-bit values |
| **-C** | **EXP ( n1 n2 -- n1+/n+/C** | 1’s complement 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 )** | Bit-wise AND of top two values |
| **OR** | **EXP ( n1 n2 -- n1 v n2 )** | Bit-wise 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 )** | Bit-wise Ex-OR the top 2 values |

### 4.6.2 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 |

### 4.6.3 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 -- )**<br>**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 |

<table>
<tr><td>+LOOP</td><td>EXP ( n -- )<br>RET ( u| limit| I -- u| limit| I+n )</td><td>Repeat LOOP, if I+n < limit</td></tr>
<tr><td>#DO</td><td>EXP ( n -- ) RET ( -- u| u| n )</td><td>Execute the #DO .. #LOOP block n times</td></tr>
<tr><td>#LOOP</td><td>EXP ( -- )<br>RET ( u| u| I--u| u| I-1 )</td><td>Decrement loop index by 1 down to zero</td></tr>
<tr><td>?DO</td><td>EXP ( Limit Start -- )</td><td>if start=limit, skip LOOP block</td></tr>
<tr><td>?LEAVE</td><td>EXP ( -- )</td><td>Exit any loop, if condition is true</td></tr>
<tr><td>-?LEAVE</td><td>EXP ( -- )</td><td>Exit any loop, if condition is false</td></tr>
</table>

### 4.6.4 Stack Operations

<table>
<tr><td>0 .. Fh,</td><td>EXP ( -- n )</td><td></td></tr>
<tr><td>0 .. 15</td><td>EXP ( -- n )</td><td>Push 4-bit literal on Exp. Stack</td></tr>
<tr><td>’ <name></td><td>EXP ( -- ROMAddr )</td><td>Places ROM address of colon definition <name> on Exp. Stack</td></tr>
<tr><td><ROT</td><td>EXP ( n1 n2 n -- n n1 n2)</td><td>Move top value to 3rd stack position</td></tr>
<tr><td>>R</td><td>EXP ( n -- ) RET ( -- u| u| n )</td><td>Move top value onto the Return Stack</td></tr>
<tr><td>?DUP</td><td>EXP ( n -- n n )</td><td>Duplicate top value, if n <>0</td></tr>
<tr><td>DEPTH</td><td>EXP ( -- n )</td><td>Get current Expression Stack depth</td></tr>
<tr><td>DROP</td><td>EXP ( n -- )</td><td>Remove the top 4-bit value</td></tr>
<tr><td>DUP</td><td>EXP ( n -- n n )</td><td>Duplicate the top 4-bit value</td></tr>
<tr><td>I</td><td>EXP ( -- I ) RET ( u| u| I — u| u| I )</td><td>Copy loop index I from Return to Expression Stack</td></tr>
<tr><td>J</td><td>EXP ( -- J )<br>RET ( u| u| J u| u| I -- u| u| J u| u| I )</td><td>Fetch index value of outer loop [2nd Return Stack level entry]</td></tr>
<tr><td>NIP</td><td>EXP ( n1 n2 -- n2 )</td><td>Drop second to top 4-bit value</td></tr>
<tr><td>OVER</td><td>EXP ( n1 n2 -- n1 n2 n1 )</td><td>Copy 2nd over top 4-bit value</td></tr>
<tr><td>PICK</td><td>EXP ( x -- n[x] )</td><td>Copy the $x^{th}$ value from the Expression Stack onto TOS</td></tr>
<tr><td>RFREE</td><td>EXP ( -- n )</td><td>Get # of unused Return Stack entries</td></tr>
<tr><td>R></td><td>EXP ( -- n ) RET ( u| u| n -- )</td><td>Move top 4-bits from return to Expression Stack</td></tr>
<tr><td>R@</td><td>EXP ( -- n )<br>RET ( u| u| n -- u| u| n )</td><td>Copy top 4-bits from return to Expression Stack</td></tr>
</table>

<table>
<tr><td>ROLL</td><td>EXP ( n -- )</td><td>Move n<sup>th</sup> value within stack to top</td></tr>
<tr><td>ROT</td><td>EXP ( n1 n2 n -- n2 n n1)</td><td>Move 3rd stack value to top pos.</td></tr>
<tr><td>SWAP</td><td>EXP ( n1 n2 -- n2 n1 )</td><td>Exchange top two values on stack</td></tr>
<tr><td>TUCK</td><td>EXP ( n1 n2 -- n2 n1 n2 )</td><td>Duplicate top value, move under second item</td></tr>
<tr><td>2>R</td><td>EXP ( n1 n2 -- )<br>RET ( -- u| n2| n1 )</td><td>Move top two values from Expression to Return Stack</td></tr>
<tr><td>2DROP</td><td>EXP ( n1 n2 -- )</td><td>Drop top 2 values from the stack</td></tr>
<tr><td>2DUP</td><td>EXP ( d -- d d )</td><td>Duplicate top 8-bit value</td></tr>
<tr><td>2NIP</td><td>EXP ( d1 d2 -- d2 )</td><td>Drop 2nd 8-bit value from stack</td></tr>
<tr><td>2OVER</td><td>EXP ( d1 d2 -- d1 d2 d1 )</td><td>Copy 2nd 8-bit value over top value</td></tr>
<tr><td>2<ROT</td><td>EXP ( d1 d2 d -- d d1 d2)</td><td>Move top 8-bit value to 3rd pos.</td></tr>
<tr><td>2R></td><td>EXP ( -- n1 n2 )<br>RET ( u| n2| n1 -- )</td><td>Move top 8-bits from Return to Expression Stack</td></tr>
<tr><td>2R@</td><td>EXP ( -- n1 n2 )<br>RET ( u| n2| n1 -- u| n2| n1)</td><td>Copy top 8-bits from return to Expression Stack</td></tr>
<tr><td>2ROT</td><td>EXP ( d1 d2 d -- d2 d d1)</td><td>Move 3rd 8-bit value to top value</td></tr>
<tr><td>2SWAP</td><td>EXP ( d1 d2 -- d2 d1 )</td><td>Exchange top two 8-bit values</td></tr>
<tr><td>2TUCK</td><td>EXP ( d1 d2 -- d2 d1 d2 )</td><td>Tuck top 8-bits under 2nd byte</td></tr>
<tr><td>3>R</td><td>EXP ( n1 n2 n3 -- )<br>RET ( -- n3| n2| n1 )</td><td>Move top 3 nibbles from the Expression onto the Return Stack</td></tr>
<tr><td>3DROP</td><td>EXP ( n1 n2 n3 -- )</td><td>Remove top 3 nibbles from stack</td></tr>
<tr><td>3DUP</td><td>EXP ( t -- t t )</td><td>Duplicate top 12-bit value</td></tr>
<tr><td>3R></td><td>EXP ( -- n1 n2 n3 )<br>RET ( n3|n2|n1 -- )</td><td>Move top 3 nibbles from Return to the Expression Stack</td></tr>
<tr><td>3R@</td><td>EXP ( -- n1 n2 n3 )<br>RET ( n3| n2| n1 -- n3| n2| n1 )</td><td>Copy 3 nibbles (1 entry) from the Return to the Expression Stack</td></tr>
</table>

### 4.6.5 Memory Operations

<table>
<tr><td>!</td><td>EXP ( n addr -- )</td><td>Store a 4-bit value in RAM</td></tr>
<tr><td>@</td><td>EXP ( addr -- n )</td><td>Fetch a 4-bit value from RAM</td></tr>
<tr><td>+!</td><td>EXP ( n addr -- )</td><td>Add 4-bit value to RAM contents</td></tr>
<tr><td>1+!</td><td>EXP ( addr -- )</td><td>Increment a 4-bit value in RAM</td></tr>
<tr><td>1-!</td><td>EXP ( addr -- )</td><td>Decrement a 4-bit value in RAM</td></tr>
</table>

| | | |
|---|---|---|
| **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 an 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 |

### 4.6.6 Predefined Structures

| | | |
|---|---|---|
| **( ccccccc)** | | In-line comment definition |
| **\ ccccccc** | | Comment until end of the line |
| **: <name>** | **RET ( -- )** | Beginning 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 |

<table>
<tr><td>2LARRAY</td><td>EXP ( d -- )</td><td>Allocates space for a long byte array</td></tr>
<tr><td>Index</td><td>EXP (n|d addr--addr’)</td><td>Run-time array access using a variable array index</td></tr>
<tr><td>ROMCONST</td><td>EXP ( -- )</td><td>Define ROM look-up table with 8-bit values</td></tr>
<tr><td>VARIABLE</td><td>EXP ( -- )</td><td>Allocates memory for 4-bit value</td></tr>
<tr><td>2VARIABLE</td><td>EXP ( -- )</td><td>Creates an 8-bit variable</td></tr>
<tr><td><n> ALLOT</td><td></td><td>Allocate space for <n+1> nibbles of un-initialized RAM</td></tr>
<tr><td>AT <address></td><td></td><td>Fixed <address> placement</td></tr>
<tr><td>: INTx</td><td>RET ( -- ROMAddr )</td><td>Interrupt service routine entry</td></tr>
<tr><td>: $AutoSleep</td><td></td><td>Entry point address on Return Stack underflow</td></tr>
<tr><td>: $RESET</td><td>EXP ( -- )</td><td>Entry point on power-on reset</td></tr>
</table>

### 4.6.7 Assembler Mnemonics

<table>
<tr><td>ADD</td><td>EXP ( n1 n2 -- n1+n2 )</td><td>Add the top two 4-bit values</td></tr>
<tr><td>ADDC</td><td>EXP ( n1 n2 -- n1+n2+C )</td><td>Add with Carry top two values</td></tr>
<tr><td>CCR!</td><td>EXP ( n -- )</td><td>Write top value into the CCR</td></tr>
<tr><td>CCR@</td><td>EXP ( -- n )</td><td>Fetch the CCR onto top of stack</td></tr>
<tr><td>CMP_EQ</td><td>EXP ( n1 n2 -- n1 )</td><td>If n1=n2, then Branch flag set</td></tr>
<tr><td>CMP_GE</td><td>EXP ( n1 n2 -- n1 )</td><td>If n1>=n2, then Branch flag set</td></tr>
<tr><td>CMP_GT</td><td>EXP ( n1 n2 -- n1 )</td><td>If n1>n2, then Branch flag set</td></tr>
<tr><td>CMP_LE</td><td>EXP ( n1 n2 -- n1 )</td><td>If n1<=n2, then Branch flag set</td></tr>
<tr><td>CMP_LT</td><td>EXP ( n1 n2 -- n1 )</td><td>If n1<n2, then Branch flag set</td></tr>
<tr><td>CMP_NE</td><td>EXP ( n1 n2 -- n1 )</td><td>If n1<>n2, then Branch flag set</td></tr>
<tr><td>CLR_BCF</td><td>EXP ( -- )</td><td>Clear Branch and Carry flag</td></tr>
<tr><td>SET_BCF</td><td>EXP ( -- )</td><td>Set Branch and Carry flag</td></tr>
<tr><td>TOG_BF</td><td>EXP ( -- )</td><td>Toggle the Branch flag</td></tr>
<tr><td>DAA</td><td>EXP ( n>9 or C set -- n+6)</td><td>BCD arithmetic adjust [addition]</td></tr>
<tr><td>DAS</td><td>EXP ( n -- 10+/n+C )</td><td>9’s complement for BCD subtract</td></tr>
<tr><td>DEC</td><td>EXP ( n -- n-1 )</td><td>Decrement top value by 1</td></tr>
<tr><td>DECR</td><td>RET ( u| u| l — u| u| l-1 )</td><td>Decrement value on the Return Stack</td></tr>
<tr><td>DI</td><td>EXP ( -- )</td><td>Disable interrupts</td></tr>
<tr><td>DROPR</td><td>RET ( u| u| u -- )</td><td>Drop element from Return Stack</td></tr>
<tr><td>EXIT</td><td>RET ( ROMAddr -- )</td><td>Exit from current “:”-definition</td></tr>
<tr><td>EI</td><td>EXP ( -- )</td><td>Enable interrupts</td></tr>
<tr><td>IN</td><td>EXP ( port -- data )</td><td>Read data from an I/O port</td></tr>
</table>

| | | |
|---|---|---|
| **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!** | **XP ( d -- )** | Store as Return Stack Pointer |
| **RP@** | **EXP ( -- d )** | Fetch current Return 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 complement subtract with Borrow |
| **TABLE** | **EXP ( -- d )**<br>**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-increment × indirect RAM fetch |
| **[X-]@** | **EXP ( -- n )** | Post-decrement × 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-increment × indirect RAM store |
| **[X-]!** | **EXP ( n -- )** | Post-decrement × 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-increment Y indirect RAM fetch |
| **[Y-]@** | **EXP ( -- n )** | Post-decrement 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-increment Y indirect RAM store |
| **[Y-]!** | **EXP ( n -- )** | Post-decrement 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 immediately |
| **>Y  $xx** | **EXP ( -- )** | Set Y register immediately |

## 4.7 Short Form Dictionary

***Table 4-1.*** MARC4 - Control Commands

| Command | Bytes | 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--<br>u u index-1 | | | | B | |
| +LOOP | 10 | n -- | u limit index--<br>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 | |
| EXECUTE | 3 | ROMaddr -- | --(2+x level)-- | | | | | |
| IF | 3 | | | | | | B | |
| OF | 4 | n1 -- n1n2 --(n1) | | | | CY | B | |
| THEN | 0 | | | | | CY | B | |
| CCR@ | 1 | -- n | | | | | | |
| CCR! | 1 | n -- | | | | CY | B | I |
| CLR_BCF | 2 | --(1 level)-- | | | | CY | B | |
| EI | 2 | | | | | CY | B | I |
| EXIT | 1 | | oldPC -- | | | | | |
| DI | 1 | | | | | | | I |
| SET_BCF | 1 | | | | | CY | B | |
| SWI0..SWI7 | 4 | -- (2 level) -- | | | | | | I |
| TOG_BF | 1 | | | | | | B | |

***Table 4-2.*** MARC4 - Mathematic Commands

| Command | Bytes | 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 | d n -- d+n | -- (1 level) -- | | | CY | B | |
| T+! | 19 | nh nm nl addr<br>-- (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 addr<br>-- (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 | |
| D0= | 2 | d -- | | | | | B | |
| CMP_GE | 1 | n1 n2 -- n1 | | | | CY | B | |
| D>= | 19 | d1 d2 -- | -- (1 level u d2h d2l) -- | | | CY | B | |
| CMP_GT | 1 | n1 n2 -- n1 | | | | CY | B | |

**Table 4-2.** MARC4 - Mathematic Commands (Continued)

| Command | Bytes | Expression Stack | Return Stack | X | Y | CY | B | I |
|---|---|---|---|---|---|---|---|---|
| > | 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 | |
| DMAX | 30 | d1 d2 --d1 d1 d2-- dmax | -- (3 level) -- | | | CY | B | |
| MIN | 7 | n1 n2 -- nmin | -- (1 level) -- | | | CY | B | |
| DMIN | 30 | d1 d2 --d1 d1 d2-- dmin | -- (3 level) -- | | | CY | B | |
| NEGATE | 2 | n1 -- -n1 | | | | | B | |
| DNEGATE | 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 and 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 | | | | | | |

***Table 4-3.*** MARC4 - Memory Commands

| Command | Bytes | 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 -- nh nl | -- (2 level) -- | | | CY | B | |
| TABLE | 1 | -- nh nl | (2 level) -- | | | | | |
| ROMBYTE@ | 2 | ROMaddr -- nh nl | -- (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 -- | | | | | | |
| ’ | 3 | - - ROMaddr | | | | | | |

***Table 4-4.*** MARC4 - Commands

| Command | Bytes | Expression Stack | Return Stack | X | Y | CY | B | I |
|---|---|---|---|---|---|---|---|---|
| ! | 2 | n addr-- | | X | Y | | | |
| #DO | 1 | index-- | -- u u index | | | | | |
| #LOOP | 4 | | u u index--<br>u u index-1 | | | | B | |
| +LOOP | 10 | n -- | u limit index--<br>u limit index+n | | | CY | B | |
| -?LEAVE | 3 | | | | | | B | |
| <ROT | 2 | n1 n2 n3 -- n3 n1 n2 | | | | | | |
| >RP xxh | 2 | | | | | | | |
| >SP xxh | 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 | -- n | u u n -- u u n | | | | | |
| 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 -- | | | | | | |

***Table 4-4.*** MARC4 - Commands (Continued)

| Command | Bytes | Expression Stack | Return Stack | X | Y | CY | B | I |
|---|---|---|---|---|---|---|---|---|
| 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 and n2 | | | | | B | |
| BEGIN | 0 | | | | | | | |
| CASE | 0 | n -- n | | | | | | |
| CCR! | 1 | n -- | | | | CY | B | I |
| CCR@ | 1 | -- n | | | | | | |
| CLR_BCF | 2 | - - ( 1 level ) - - | | | | 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 | |

*Table 4-4.* MARC4 - Commands (Continued)

| Command | Bytes | Expression Stack | Return Stack | X | Y | CY | B | I |
|---|---|---|---|---|---|---|---|---|
| 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 SPl 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 | n -- | | | | | | |
| DROPR | 1 | | u u u -- | | | | | |
| 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 | 3 | ROMaddr - - | -- (2 + x 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 | |

***Table 4-4.*** MARC4 - Commands (Continued)

| Command | Bytes | Expression Stack | Return Stack | X | Y | CY | B | I |
|---|---|---|---|---|---|---|---|---|
| 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 | | | | | |
| 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 |
| ROMBYTE@ | 2 | ROMaddr -- conh conl | -- (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 SPl -- | | | | | | |
| SP@ | 1 | -- SPh SPl+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 | |

***Table 4-4.*** MARC4 - Commands (Continued)

| Command | Bytes | Expression Stack | Return Stack | X | Y | CY | B | I |
|---|---|---|---|---|---|---|---|---|
| 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 | | | | | | |

***Table 4-5.*** MARC4 - STACK Commands

| Command | Bytes | Expression Stack | Return Stack | X | Y | CY | B | I |
|---|---|---|---|---|---|---|---|---|
| DECR | 1 | | u u n -- u u n-1 | | | | B | |
| DEPTH | 9 | -(SPh SPl 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 | d/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 | | | | | |

***Table 4-5.*** MARC4 - STACK Commands (Continued)

| Command | Bytes | 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 | |
| RFREE | 30 | -(3 level)- n | -- (2 level) -- | | | CY | B | |
| ROT | 1 | n1 n2 n3 -- n2 n3 n1 | | | | | | |
| 2ROT | 5 - 7 | 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 SPl+1 | | | | | | |
| SP! | 1 | SPh SPl -- | | | | | | |
| 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) -- | | | | | |

## 4.8 Detailed Description of the qFORTH Language

### 4.8.1 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 actually generated code sequence can result due to 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

**See Also:** +! 2! 3! @

| | |
|---|---|
| | **!** |

**Example:**

```
              VARIABLE ControlState
          VARIABLE Semaphore

: InitVariables       ( initialize VARs      )
  0 ControlState !    ( Set up control flag  )
  2 Semaphore !       ( Set up a preset value )
;
```

**’**

**4.8.2 tick**

| | |
|---|---|
| **Purpose:** | Leaves a compiled ROM code address on the EXP stack. Used in the form ’ <name>.<br>’ searches for a name in the qFORTH dictionary and returns that name's compilation address (code address). If the name is not found in the dictionary, an error message results. |
| **Category:** | Control structure |
| **Stack Effect:** | EXP ( - -ROM_addr ) |
| | RET ( -- ) |
| **Stack Changes:** | EXP: 3 nibbles are pushed on the stack |
| | RET: not affected |
| **Flags:** | Not affected |
| **Bytes Used:** | 3 |
| **See Also:** | EXECUTE |

| ’ |
|---|

**Example:**

’ is typically used to initialize the content of a variable with the code address of the qFORTH word for vectored execution.

For example, a program might contain a variable named $ERROR which would specify the action to be taken if a certain type of error occurred. At compilation time, $ERROR is “vectored” with a sequence such as

```
3 ARRAY $ERROR
’ ERROR-ROUTINE $ERROR 3!
```

and the main program, if it detects an error, can execute the sequence

```
$ERROR 3@ EXECUTE
```

to invoke the error handler.

This program's error-handling routine can be changed “on the fly” by changing the address stored in $ERROR without modifying the main program in any way.

# #DO

## 4.8.3 Hash-DO

**Purpose:** #DO indicates the start of an iterative “decrement-if-nonzero” loop structure. It is used only within a macrocolon 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 behavior 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

# #DO

**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

### 4.8.4 Hash-LOOP

**Purpose:** #LOOP indicates the end of an iterative “decrement-if-non-zero” 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, i.e., the execution of the #LOOP word, until the index reaches zero. If the new index is decre mented 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 behavior 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 as long as index-1 <> 0

**X Y Registers:** Not affected

**Bytes Used:** 3 - 4

**See Also:** #DO ?LEAVE -?LEAVE

# #LOOP

**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

**4.8.5 Autosleep**

**Purpose:** The $AUTOSLEEP function will automatically be placed at ROM address $000 by the compiler and may be redefined slightly 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 is automatically loaded to the address $000 ($AUTOSLEEP). This forces the MARC4 into sleep mode through processing the $AUTOS LEEP routine. This sleep mode is a shutdown condition which is used to reduce the average system power consumption, whereby the CPU is halted.

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

Attention: It is not recommended to use the SLEEP instruction other than in the $RESET or $AUTOSLEEP because it might result in unwanted side effects within other interrupt routines. If any interrupt is active or pending, the SLEEP instruction will be executed in the same way as an NOP!

**Category:** Interrupt handling/predefined qFORTH colon definition

**Library Implementation:**

```
: $AUTOSLEEP
$TIRED:       NOP
              SLEEP
              SET_BCF
              BRA_$TIRED
                 [ E 0 R 0 ]
;;
```

**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, DI, EI, RTI, SLEEP

# $AUTOSLEEP

**Example:**

```
\ Improved $AUTOSLEEP routine for noisy environment

: Clr_INT_controller
;; RTI [N]

:$AUTOSLEEP
$_Tired:    NOP
            SLEEP
            NOP SET_BCF
            CPr_INT_controller        \ Clear unwanted spurious
            BRA $_Tired [E0 R0]       \ INT pending/active bits
;;
```

# $RESET

## 4.8.6 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 also used.

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 a customer specified input value.

**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- |

**X Y Registers:** Not affected

**Bytes Used:** Modified by the customer

**See Also:** $AUTOSLEEP

# $RESET

**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
      Selftest
      THEN
      Init_Peripherals
;
```

# ( comment) \

## 4.8.7 Paren, Backslash

**Purpose:** Begins a comment, used either in the form
( ccccc) or \ comment is rest of line

The characters “ccccc” 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:** %

# ( comment) \

**Example:**

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

# + ADD

## 4.8.8 Plus

| | |
|---|---|
| **Purpose:** | Adds the top two 4-bit values and replaces 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 |

# + ADD

**Example:**

```
VARIABLE Result
: Single-addition
        5 3 +               ( RPN addition: 5 + 3 := 8     )
        Result !            ( Save result in memory        )
        3 Result +!         ( Add 3 to memory location  )
;
```

# +!

### 4.8.9 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 such as

“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

### 4.8.10 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 | |



4747A–4BMCU–01/04

## +C ADDC

**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
  SWAP
;

: ADDC-Example
  50 190 D+                        ( -- 240 = F0h ; no CARRY or BRANCH       )
  PLUS-C
  Overflow
  2DROP 2DROP                      ( the results.                            )
;
```

# +LOOP

## 4.8.11 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 (>=), 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

# +LOOP

**Example:**

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

## - SUB

### 4.8.12 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

## - SUB

**Example:**

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

# -?LEAVE

**4.8.13 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

# -?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

### 4.8.14 Minus-C

**Purpose:** Subtract with BORROW [ = NO CARRY or /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

| -C | SUBB |
|---|---|

**Example:**

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

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

### 4.8.15 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 |

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

**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 --                           )
;
```

# 0<>

## 4.8.16 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). This differs from 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<>

# 0<>

**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=

### 4.8.17 Zero-equal

**Purpose:** Compares the 4-bit value on top of the stack with zero.

If the value of the stack is equal to zero, then the BRANCH flag is set in the condition code register. This differs from 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<>

## 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

### 4.8.18 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+! |

## 1+ INC

**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+!

## 4.8.19 One-plus-store

**Purpose:** Increments 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 or X 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-!

## 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

### 4.8.20 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-! |

## 1- DEC

**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-!

### 4.8.21 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 (or X) register

**Bytes Used:** 4

**See Also:** +! 1+! !

# 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!

### 4.8.22 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-!

## 2!

**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

### 4.8.23 Two-multiply

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

| | | TOS | | | | | |
|---|---|---|---|---|---|---|---|
| C | ← | 3 | 2 | 1 | 0 | ← | 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: MSB of TOS is shifted into CARRY

BRANCH flag = CARRY flag

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** ROL ROR SHR 2/ D2* D2/

## 2* SHL

**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
  2/     DROP                (-SHR-[no CARRY] F -- [CARRY] 7 = 0111b )

  CLR_BCF 6                  ( 6 = 0110b                                 )
  2*     DROP                (- SHL - [no CARRY] 6 -- [no C] C = 1100b)
;
```

## 2/ SHR

### 4.8.24 Two-divide

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

| | | TOS | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | → | 3 | 2 | 1 | 0 | → | C |

**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: LSB of TOS is shifted into CARRY

BRANCH flag = CARRY flag

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** 2* SHL D2* D2/ ROR ROL

# 2/ SHR

**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

### 4.8.25 Two-left-rote

**Purpose:** Moves the top 8-bit value to the third byte position on the EXP stack, i.e., 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

# 2<ROT

**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

### 4.8.26 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>

## 2>R

**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@

### 4.8.27 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<>

## 2@

**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

**4.8.28 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 the time of compilation, 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

# 2ARRAY

**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           )
;

: Setup_RAM
  Init_RawData
  RawData [3] 2@
  0h   D+
;
```

## 2CONSTANT

### 4.8.29 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

# 2CONSTANT

**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

## 4.8.30 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

# 2DROP

**Example:**

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

## 2DUP

### 4.8.31 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

# 2DUP

**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

### 4.8.32 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 the time of compilation, 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

# 2LARRAY

**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

### 4.8.33 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

## 2NIP

**Example:**

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

# 2OVER

**4.8.34 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

# 2OVER

**Example:**

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

# 2R>

## 4.8.35 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>

# 2R>

**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@

### 4.8.36 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>

## 2R@

**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

### 4.8.37 Two-rote

**Purpose:** Moves the third double-length (8-bit) value onto the top of the Expression Stack.

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

**Library Implementation:**

```
CODE 2ROT
    2>R 2SWAP        ( d1 d2 d3 -- d2 d1 )
    2R> 2SWAP        ( d2 d1 -- d2 d3 d1 )
END-CODE
```

**Stack Effect:** EXP ( d1 d2 d3 --- d2 d3 d1 )

RET ( -- )

**Stack Changes:** EXP: affected ( changes order of elements )

RET: affected (3 levels are used in between)

**Flags:** Not affected

**X Y Registers:** Not affected

**Bytes Used:** 5 - 7

**See Also:** 2<ROT <ROT ROT

# 2ROT

**Example:**

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

# 2SWAP

## 4.8.38 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

# 2SWAP

**Example:**

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

# 2TUCK

**4.8.39 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

# 2TUCK

**Example:**

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

# 2VARIABLE

**4.8.40 Two-VARIABLE**

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

The qFORTH syntax is as follows

2VARIABLE <name> [ AT <address.> ]
[ <number> ALLOT ]

At the time of compilation, 2VARIABLE adds <name> to the dictionary and ALLOTs memory for storage of one double-length value. If AT <address> 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 * (<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 area allocated 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

# 2VARIABLE

**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!

### 4.8.41 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 elements 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-!

## 3!

**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

### 4.8.42 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>

## 3>R

**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@

### 4.8.43 Three-fetch

**Purpose:** Fetch a 12-bit variable in RAM and push it on the Expression Stack. The most significant digit address of the variable must be on the 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-!

## 3@

**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

## 4.8.44 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

# 3DROP

**Example:**

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

# 3DUP

**4.8.45 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 |

# 3DUP

**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>

### 4.8.46 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

## 3R>

**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@

### 4.8.47 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> |

## 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', implicite EXIT       )
```

| : |
|---|

### 4.8.48 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

### 4.8.49 Semicolon

**Purpose:**

Terminates a qFORTH colon definition, i.e. exits 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 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, such as:

BEGIN ... WHILE, REPEAT, ... UNTIL, ... AGAIN, and

IF ... [ ELSE ... ] THEN

**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: | not affected |
| RET: | The address on top of the stack is moved into the PC |

**Flags:** CARRY and BRANCH flags are not affected

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** **:** ?LEAVE -?LEAVE ;;

## ;      EXIT      RTI

**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'                       )
;
```

## ;;

### 4.8.50 Double-Semicolon

| | | |
|---|---|---|
| **Purpose:** | Suppresses the code generation of an EXIT or RTI at the end of a colon definition. This function is typically used after any TABLE instruction (see ROMByte@, DTABLE@), in C computed goto jump tables (see EXECUTE) or in the $AUTOSLEEP 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:** | : ; $AUTOSLEEP, ROMByte@, DTABLE@ | |

;;

**Example:**

```
: Do_Incr
              DROPR                  \ Skip Return address
  Time_count   1*!
[N];

: Do_Decr
              DROPR
              Time_count   1-!
[N];

: Do_Reset
              DROPR
              0 Time_Count !
[N];

:
  Jump_Table
              Do_Nothing
              Do_Inrc
              Do_Decr
              Do_Reset
;; AT FF0h                            \ Do not generate an EXIT

: Exec_Example ( n  - - )
              >R ' Jump_Table   R>
              2* M+                  \ calculate vector address
              EXECUTE
;
```

# < CMP_LT

## 4.8.51 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<=

## <   CMP_LT

**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

## 4.8.52 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<= | | |

## <= CMP_LE

**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

### 4.8.53 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<> | | |

## <> CMP_NE

**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

### 4.8.54 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 |

# <ROT

**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

## 4.8.55 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. This is 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 ) |
| both | 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=

## = CMP_EQ

**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

### 4.8.56 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. This is 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<

## > CMP_GT

**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

## 4.8.57 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<=

## >= CMP_GE

**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

## 4.8.58 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>

## >R

**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

**4.8.59 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 ?DO
            OVER   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 and 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

# ?DO

**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

## 4.8.60 Query-doop

**Purpose:** Duplicates the top value on the stack only if it is not 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 ?DUP
        DUP   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<>

# ?DUP

**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

### 4.8.61 Query-leave

**Purpose:** Conditional exit from within a control structure if the previous tested condition was TRUE (i.e., 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

# ?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                            )
;
```

## @

### 4.8.62 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, until 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

## 4.8.63 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

# AGAIN

**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

## 4.8.64 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 |

# ALLOT

**Example:**

```
VARIABLE Limits 7 ALLOT       ( Allocates 8 nibbles for the         )
                              ( variable LIMITS                     )
VARIABLE R0     31 ALLOT      ( allocate space for RETURN stack     )
VARIABLE S0     19 ALLOT      ( allot 20 nibbles for EXP stack      )
```

# AND

### 4.8.65 AND

| | |
|---|---|
| **Purpose:** | Bit-wise 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 |

# AND

**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

### 4.8.66 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 the time of compilation, 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

# ARRAY

**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

### 4.8.67 AT

| | |
|---|---|
| **Purpose:** | Specifies the ABSOLUTE memory location AT where either a variable is 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 |

## AT

**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                 )

: INT0_Service
        Fh State !
        BEGIN
            CheckState
            State 1-!
    UNTIL
; AT 400h                       ( force placement at ROM address 400h          )
```

# BEGIN

### 4.8.68 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

# BEGIN

**Example:**

```
: BEGIN-UNTIL
  3 BEGIN                 ( increment value from 3 til 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

## 4.8.69 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

# CASE

**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          )
      2 OF TWO   ENDOF               ( activated.                          )
      3 OF THREE ENDOF
      ERROR                          ( otherwise ...                       )
  ENDCASE                            ( n --                                )
;
```

# CCR!

### 4.8.70 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

# CCR!

**Example 1:**

```
: INT5              ( timer interrupt service routine      )
    CCR@            ( save the current condition codes     )
    Inc_Time        ( call procedure.                      )
    CCR!            ( restore CCR status                   )
;                   ( RTI                                  )
```

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@

## 4.8.71 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 |

## CCR@

**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 stack       )
;

: 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

### 4.8.72 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 |

# CLR_BCF

**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 is not 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

## 4.8.73 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 not used too often by the program.

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 longer 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 (;)

# CODE

**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

**4.8.74 Dollar-Include**

**Purpose:** Compiles qFORTH source code from another text file. Used in form

$INCLUDE <filename>

$INCLUDE loads a qFORTH program from an ASCII text file. Such a source text file may be created using any standard text editor.

$INCLUDE is “state-smart” and may be used (together with a filename) inside of a colon definition.

The file name extension ’INC’ is the default and may be omitted.

**Category:** Compiler

**Stack Changes:** EXP ( - - )

RET ( - - )

**Flags:** Not affected

# $INCLUDE

**Example:**

The sequence $INCLUDE MYPROG.SCR causes the qFORTH source code in file MYPROG.SCR to be compiled.

# $RAMSIZE $ROMSIZE

**4.8.75 Dollar-RAMSize Dollar-ROMSize**

**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. For more details refer to the "MARC4 User's Guide"

**Category:** Compiler directives

**$RAMSIZE:** Specifies the RAM size of the target processor. Default size is 255 nibbles (from $00 .. $FF). Some processors contain 253 nibbles only, whereby the RAM cells at $FC, $FD and $FE are not available.

**$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 8-bit constant in your source program.

# $RAMSIZE $ROMSIZE

**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

## 4.8.76 CONSTANT

**Purpose:** Creates a 4-bit constant; implemented in a qFORTH program as:

n CONSTANT <name>

with 0 <= n <= 15 or 0 <= n <= Fh or 0000b <= n <= 1111b

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

# CONSTANT

**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+

### 4.8.77 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*

# D+

**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+!

## 4.8.78 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<>

# D+!

**Example:**

```
2VARIABLE count AT 43h
: Double_Arithm
   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-

## 4.8.79 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>

## D-

**Example:**

```
: Double_Minus
   15h 13h
   D- 2DROP          ( result: -- 02 ; no flags      )
   13h 15h
   D- 2DROP          ( result: -- FE ; C & B flag    )
   18h 18h D-        ( result: -- 00 ; no flags      )
   D0=               ( result: -- ; B flag set       )
;
```

# D-!

## 4.8.80 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 are changed

**Bytes Used:** 10

**See Also:** D- D+ 2! 2@ D+!

## 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 set  )
;
```

# D0<>

## 4.8.81 D-zero-not-equal

**Purpose:** Compares the 8-bit value on top of the stack with 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= | |

# D0<>

**Example:**

```
1 CONSTANT true
0 CONSTANT false
: DCompare
        12h D0<>
        IF true ELSE false THEN DROP( result is 'true' )
        12h D- D0<>
        IF true ELSE false THEN DROP( result is 'false' )
;
```

# D0=

### 4.8.82 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 in the CCR.

**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 are 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<>

# D0=

**Example:**

```
1 CONSTANT true
0 CONSTANT false
: DCompare
  12h D0=
  IF true ELSE false THEN DROP          ( result is ’false’ )
  12h D0- D0=
  IF true ELSE false THEN DROP          ( result is ’true’  )
;
```

# D2*

### 4.8.83 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<>

# D2*

**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/

### 4.8.84 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 has been set
BRANCH flag = CARRY flag

**X Y Registers:** Not affected

**Bytes Used:** 4

**See Also:** D- D+ D+! D-! D2* D<> D=

# D2/

**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<

### 4.8.85 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<>

# D<

**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' )
;
```

# D<=

## 4.8.86 D-less-equal

**Purpose:** ’Less-than-or-equal’ comparison for the top two unsigned 8-bit values. Instead of pushing a Boolean TRUE flag on the stack if the 2nd number on the stack is ’less-or-equal-than’ the TOS number, ’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:** ’D>’ ± 1

**See Also:** D< D> D<> D= D>= D0= D0<>

## D<=

**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<>

## 4.8.87 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<>

## D<>

**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=

## 4.8.88 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:** ’D<>’ ± 1

**See Also:** D< D> D<> D<= D>= D0= D0<>

# D=

**Example:**

```
1 CONSTANT true
0 CONSTANT false
: DCompare
  12h 15h D=
  F 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>

## 4.8.89 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<>

# D>

**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>=

### 4.8.90 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:** 'D<' ± 1

**See Also:** D< D> D<> D= D<= D0= D0<>

## D>=

**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

## 4.8.91 Double-to-single NIP

**Purpose:** Transform an 8-bit value into a 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

## D>S NIP

**Example 1:**

```
: NIP_Example
  10H                 (       -- 1 0          )
  3                   ( 1 0   -- 1 0 3        )
  NIP                 ( 1 0 3 -- 1 3          )
  2DROP               ( 1 3   --              )
;
```

**Example 2:**

```
: StoD_DtoS
  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 SPl S0h S0l -- diff  )
          NIP 1-         ( diff -- n                    )
;
```

# DAA

### 4.8.92 Decimal Adjust

| | |
|---|---|
| **Purpose:** | Decimal-arithmetic-adjustment 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 |

# DAA

**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

### 4.8.93 D-A-S or Decimal-Adjust for Subtraction

**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

# DAS

**Example:**

```
8 CONSTANT BCD#                  ( number of BCD digits              )
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 registers )
      #DO
         [Y]@ [X-]@ DAS          ( 9's complement generation         )
         + DAA [Y-]!
      #LOOP
;

: Calculate
      BCD# Input2 [7] Input1 [7] BCD-         ( Inp1:=Inp1-Input2    )
      3 BCD# Input1 [7] DIG-                  ( Input1 := Input1 - 3   )
;
```

# DECR

### 4.8.94 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) |

# 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     )
;
```

**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 determines 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 start 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

### 4.8.95 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 SPl 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

# DEPTH

**Example:**

```
: DEPTH-Ex
  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.                )
;
```

# DI

### 4.8.96 Disable-Interrupt or D-I

**Purpose:** Disable execution of higher prioritized interrupts until the next EI or RTI instruction is performed. The access to semaphores, variables or peripheral resources by differently prioritized interrupt routines will require a DI/EI sequence.

Note: The generation of interrupts and latching in the interrupt pending register is not disabled.

**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

## DI

**Example:**

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

### 4.8.97 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                     )
_DMAX3:    DROPR          ( skip compares values from RET stack    )
           2DROP          ( d1 --                                  )
           2R>            ( -- d2                                  )
           EXIT
 _DMAX1:   2DROP          ( skip compares values from EXP stack    )
 _DMAX2:   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

# DMAX

**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

**4.8.98 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 compares values from RET stack         )
             2DROP        ( d1 --                                       )
             2R>          ( -- d2                                       )
             EXIT
_DMIN1:      2DROP        ( skip compares 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 are 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

# DMIN

**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

## 4.8.99 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

# DNEGATE

**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

## 4.8.100 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

## DO

**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

## 4.8.101 DROP

| | | |
|---|---|---|
| **Purpose:** | Removes one 4-bit value from the 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 | |

# DROP

**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

## 4.8.102 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 |

# DROPR

**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 determines 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 start 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@

### 4.8.103 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 microcode execution.             )
           [ 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

# DTABLE@

**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.        )
;
```

# DTOGGLE

## 4.8.104 D-TOGGLE

**Purpose:** TOGGLEs [exclusive ors] a byte at a given address with a specified bit pattern. The address of the 8-bit variable is on top of the Expression Stack.

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

**Library Implementation:**

```
: DTOGGLE   Y!      ( d addr -- nh nl   )
[+Y]@ XOR           ( nh nl -- nr rl'   )
[Y-]!
[Y]@ XOR            ( nh -- rl'         )
[Y]!                ( rl' --            )
;
```

**Stack Effect:** EXP ( d addr -- )

RET (-- )

**Stack Changes:** EXP: 4 elements are popped from the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag set, if higher nibble gets zero

**X Y Registers:** The contents of the Y register are changed

**Bytes Used:** 8

**See Also:** TOGGLE XOR

# DTOGGLE

**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

### 4.8.105 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 |

# DUP

**Example:**

```
: ONE-DUP
  9          ( -- 9         )
  5          ( 9 -- 9 5     )
  DUP        ( 9 5 -- 9 5 5 )
  3DROP      ( 9 5 5 --     )
;
```

# EI

### 4.8.106 Enable-Interrupt or E-I

**Purpose:** Sets the INTERRUPT_ENABLE flag in the condition code register. Use EI/DI only, if different tasks use the same resources; i.e. two tasks both use a peripheral EEPROM or RAM - without semaphore handling.

Note: Under normal circumstances, the programer 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 INT0 .. INT7 SWI0 .. SWI7

# EI

**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

**4.8.107 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

# 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. )
;
```

# END-CODE

## 4.8.108 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 ';' |

# END-CODE

**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

### 4.8.109 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. Please note, however, that a 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

# ENDCASE

**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       )
  2 OF TWO   ENDOF             ( activated.                       )
  3 OF THREE ENDOF             (                                  )
  ERROR                        ( otherwise ...                    )
ENDCASE                        ( n --                             )
;
```

# ENDOF

## 4.8.110 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 0 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

# 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 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      )
  2 OF TWO   ENDOF            ( activated.                      )
  3 OF THREE ENDOF            (                                 )
  ERROR                       ( otherwise ...                   )
ENDCASE                       ( n --                            )
;
```

# ERASE

## 4.8.111 ERASE

**Purpose:** Resets n digits in a block of memory (RAM) to zero. Whereas n is less than 16; if n is zero, then 16 nibbles of RAM is set to 0.

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

**Library Implementation:**

```
: 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

# ERASE

**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

### 4.8.112 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, ';' is 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

# EXIT

**Example 1:**

```
: EXIT-Example
  6 2 3              (      -- 6 2 3                  )
  CMP_NE             ( 6 2 3 -- 6 2                   )
  IF SWAP 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
;
```

# EXECUTE

### 4.8.113 Execute

**Purpose:** Transfers control to the colon definition whose ROM code address is on the EXP stack.

**Category:** Control structure

**Library Implementation:**

```
: EXECUTE
        3>R
        EXIT
;
```

**Stack Effect:** EXP ( ROM addr - - )

RET ( - - ROM addr - - )

**Stack Changes:** EXP: 3 elements are popped from the stack

RET: 1 entry is used intermediately during execution

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** Not affected

**See Also:** ’ ;;

# EXECUTE

**Example:**

```
: Do_Incr
          DROPR                      \ Skip return address
          Time_count 1*!
[N];

: Do_Decr
          DROPR
          Time_count 1-!
[N];

: Do_Reset
          DROPR
          0 Time_Count !
[N];

:
  Jump_Table
          Do_Nothing
          Do_Inrc
          Do_Decr
          Do_Reset
;;  AT FF0h                          \ Do not generate an EXIT

: Exec_Example ( n  - - )
          >R ' Jump_Table   R>
          2*   M+                    \ calculate vector address
  EXECUTE
;
```

# FILL

### 4.8.114 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 are changed

**Bytes Used:** 16 + '2SWAP'

**See Also:** ERASE

# FILL

**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@

### 4.8.115 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

## I R@

**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

### 4.8.116 IF

**Purpose:** Begins an 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

# IF

**Example:**

```
: IfElseThen
  1 2 <=          ( is 1 <= 2 ?                              )
  IF              ( yes, so the If block is 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 is executed                   )
  THEN 2DROP      ( the results from the expression stack.   )
;
```

# IN

## 4.8.117 IN

| | | |
|---|---|---|
| **Purpose:** | Read data from an I/O ports. | |
| | Note: | Before changing the direction of a bi-directional (nibble-wise I/O) port from output to input, first a value of 'Fh' should be written to that port. After power-on-reset, all bi-directional 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 port = 0 ! |
| **X Y Registers:** | Not affected | |
| **Bytes Used:** | 1 | |
| **See Also:** | OUT | |

# IN

**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

### 4.8.118 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 |

# INDEX

**Example:**

```
10 ARRAY 10 Nibbles

: 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'

### 4.8.119 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 at the following fixed addresses by the compiler:

| Interrupt | Priority | ROM Address | Interrupt Opcode (Acknowledge) | Size (Bytes) |
|---|---|---|---|---|
| INT0 | lowest | 040h | C8h (SCALL 040h) | 64 |
| INT1 | | 080h | D0h (SCALL 080h) | 64 |
| INT2 | | 0C0h | D8h (SCALL 0C0h) | 64 |
| INT3 | | 100h | E0h (SCALL 100h) | 64 |
| INT4 | | 140h | E8h (SCALL 140h) | 64 |
| INT5 | | 180h | F0h (SCALL 180h) | 64 |
| INT6 | | 1C0h | F8h (SCALL 1C0h) | 32 |
| INT7 | highest | 1E0h | FCh (SCALL 1E0h) | unlimited |

During runtime the PC is 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 information 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: 1 entry (old PC) is pushed onto the stade

**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.

If the compiler directive “$OPTIMIZE - SAVECONTXT” is used, all needed register must be saved manually.

**X Y Registers:** Not affected

**Bytes Used:** 0

**See Also:** SWI0 .. SWI7 RTI $AUTOSLEEP

# 'INT0 ... INT7'

**Example:**

```
: INT5
    CCR@  Y@ X@      ( instructions will be inserted by the compiler
                        automatically                              )
    IncTime          ( activate the time-increment module every 125 ms )
    X! Y! CCR!       ( store the register data back automatically  )
;                    ( an RTI will occur in the opcode.            )
```

## J

### 4.8.120 J

**Purpose:** Leaves the 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

## J

**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

## 4.8.121 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 the time of compilation, LARRAY adds <name> to the dictionary and ALLOTs memory for storage of <number> single-length values. At execution time, <name> leaves the RAM 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

# LARRAY

**Example:**

```
12          ARRAY ShortArray                        ( normal array example.    )
64          LARRAY LongArray AT 68h
: ArrayExample
  ShortArray   [ShortArray Length] ERASE           ( set all 12 nibbles to 0    )
7           LongArray  [12]  !
8           #DO 0 0 LongArray 0 I 1- 2*
                                        M+ 2!
#LOOP
;
```

# LOOP

## 4.8.122 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

# 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                (                                         )
  DO
     I              (                                         )
     TestPort1 OUT  ( write 2, 4, 6, 8 to the 'TestPort1'.    )
  2 +LOOP           (                                         )
;
```

## M+

### 4.8.123 M-plus

**Purpose:** M+ adds the digit (4 bit) on top of the data stack to the 8-bit value below that.

**Category:** Arithmetic/logical (double-length)/FORTH colon definition

**Library Implementation:**

```
: 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 overflow

BRANCH flag = CARRY flag

**X Y Registers:** Not affected

**Bytes Used:** 5

**See Also:** D- D+ D2* D2/ M-

## M+

**Example:**

```
: MPlusMinus
  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.                              )
;
```

# M-

### 4.8.124 M-minus

**Purpose:** M- subtracts a nibble on the data stack from the 8-bit value below that.

**Category:** Arithmetic/logical (double-length)/FORTH colon definition

**Library Implementation:**

```
: 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 underflow

BRANCH flag = CARRY flag

**X Y Registers:** Not affected

**Bytes Used:** 5

**See Also:** D- D+ D2* D2/ M+

## M-

**Example:**

```
: MPlusMinus
  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

## 4.8.125 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

## MAX

**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

### 4.8.126 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

# MIN

**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

### 4.8.127 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.

**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

# MOVE

**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>

### 4.8.128 MOVE-greater

**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 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.

**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 are affected

**Bytes Used:** 10

**See Also:** MOVE FILL ERASE

# MOVE>

**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

### 4.8.129 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

# NEGATE

**Example:**

```
code true 1 end-code          ( this is only a simple example for 'true'   )
code false 0 end-code

: Negator
  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

### 4.8.130 NOP

| | |
|---|---|
| **Purpose:** | No operation; one instruction cycle of time is used. This is 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 |

# NOP

**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

### 4.8.131 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 |

# NOT

**Example:**

```
: NOT_Example
  9              (   -- 1001b    )
  NOT            ( 9 -- 0110b    )
  DROP           ( 6 --          )
;
```

# OF

### 4.8.132 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, on match

ET: not affected

**Flags:** CARRY and BRANCH flags are affected

**X Y Registers:** Not affected

**Bytes Used:** 3 - 4

**See Also:** ENDOF CASE ENDCASE

# OF

**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        )
      2 OF TWO   ENDOF                ( activated.                        )
          3 OF THREE ENDOF
      ERROR                           ( otherwise ...                     )
  ENDCASE                             ( n --                              )
;
```

# OR

### 4.8.133 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

# OR

**Example:**

```
: ERROR                    ( what happens in case of errors:      )
  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

## 4.8.134 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 |

## OUT

**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

### 4.8.135 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 |

# OVER

**Example:**

```
: OVER-Example
  3 7 4                  ( -- 3 7 4          )
  OVER                   ( 3 7 4 -- 3 7 4 7  )
  2DROP 2DROP            ( 3 7 4 7 --        )
;
```

# PICK

## 4.8.136 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 <= x <= 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 SPl -- SPh' SPl'   )
       Y! [Y-]@          ( SPh' SPl' -- 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

# PICK

**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>

### 4.8.137 R-from

**Purpose:** Removes the top 4-bit value from the Return Stack and puts the value on the Expression 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>

## R>

**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

## 4.8.138 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:** | see below |
| **Stack Effect:** | EXP ( -- n )<br>RET ( -- ) |
| **Stack Changes:** | EXP: 1 element is pushed onto the stack.<br>RET: not affected |
| **Flags:** | CARRY and BRANCH flags are affected |
| **X Y Registers:** | Not affected |
| **Bytes Used:** | 12 |
| **See Also:** | DEPTH RFREE INT0 ... INT7 |

```
: RDEPTH RP@        ( -- RPh RPl                   )
         D2/ D2/    (compute the modulo 4 number   )
         NIP        (of entries on the RET stack   )
;                   (forget entry of RDEPTH itself )
```

# RDEPTH

**Example**:

```
: Call_Again2
  RDEPTH            ( result is 2.                     )
;

: Call_Again1
  RDEPTH            ( result is 1.                     )
  Call_Again2       ( next level - new address to RET. )
;

: 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

## 4.8.139 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 are set

**X Y Registers:** Not affected

**Bytes Used:** 2 - 3

**See Also:** BEGIN WHILE UNTIL AGAIN

# REPEAT

**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

### 4.8.140 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 Stack Pointer 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

# RFREE

**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.                          )
;

: INT0
  RFREE DROP                ( result value is: 7                          )
  RFree1                    ( “call” next level.                          )
;
```

# ROL

### 4.8.141 Rotate-left

**Purpose:** Rotate the TOS left through CARRY.

| | | | TOS | | | | | |
|---|---|---|---|---|---|---|---|---|
| ← | C | ← | 3 | 2 | 1 | 0 | ← | + |

**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 = Bit3 of TOS - before operation
BRANCH flag = CARRY flag

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** ROR SHR SHL D2*

## ROL

**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

### 4.8.142 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   )
      1+ PICK >R              ( move PICKed value on RET             )
      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

## ROLL

**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 = NOP          )
  1 ROLL                             ( ..2 3 4 -- ..2 4 3 = SWAP         )
  13 ROLL                            ( ..2 4 3 -- ..2 4 3 7              )
  10 #DO DROP DROP #LOOP             ( clear the stack with 20*DROP      )
;
```

# ROMByte@TABLE

**4.8.143 ROM-byte-fetch**

**Purpose:** Fetches an 8-bit constant from ROM onto TOS, 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 the microcode execution         )
   [ 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

# ROMByte@TABLE

**Example:**

```
ROMCONST DigitTable 10h , 1 , 2 , 3 , 4 , 45h , 6 , 7, 8 , 9, Ah , Bh , Ch , Dh , Eh , 0Fh ,
( Pay 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

### 4.8.144 ROM-CONSTANT

**Purpose:** Defines 8-bit constants in the ROM for look-up tables or as ASCII string constants.

Pay 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 table items (bytes)

**See Also:** DTABLE@ ROMByte@

Please note, that only a macro, colon or VARIABLE definition is allowed following a table definition

```
ROMCONST Tab1 10h, 13h, 55,
23h 2CONSTANT #Apples
```

is not allowed since a ’,’ is exspected after 23h.

# ROMCONST

**Example 1:**

```
13 CONSTANT TextLength 03h CONSTANT LCD_Data
ROMCONST LCD_Text TextLength , " MARC4 Test "

: ExampleText
  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
;
```

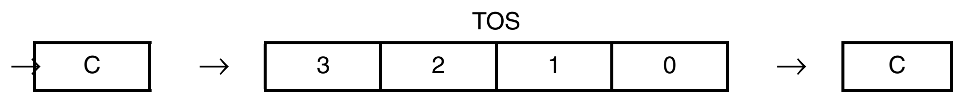
**Example 2:**

```
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     )
: Exam2     Ab P4 L/U-table
;
```

# ROR

### 4.8.145 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 = bit0 of TOS - before operation
BRANCH flag = CARRY flag

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** ROL SHR SHL

## ROR

**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

### 4.8.146 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 |

# ROT

**Example:**

```
: ROTATE_Example
  4 6 9            ( -- 4 6 9       )
  ROT              ( 4 6 9 -- 6 9 4 )
  3DROP            ( 6 9 4 --       )
;
```

# RP@

### 4.8.147 R-P-fetch

| | |
|---|---|
| **Purpose:** | Fetch the Return Stack pointer. |
| **Category:** | Assembler instruction |
| **MARC4 Opcodes:** | 71 hex |
| **Stack Effect:** | EXP ( -- RPh RPl ) |
| | RET ( -- ) |
| **Stack Changes:** | EXP: 2 elements are pushed onto the stack |
| | RET: not affected |
| | RET: new base address |
| **Flags:** | Not affected |
| **X Y Registers:** | Not affected |
| **Bytes Used:** | 1 |
| **See Also:** | SP! SP@ >SP $xx RP! >RP FCh R0 |

# RP@

**Example:**

```
VARIABLE R0 47 ALLOT          ( Return Stack: 48 nibbles 13 entry levels        )
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.               )
;

: $RESET
   > SP S0                    ( initialize the stack pointers.                   )
   > RP FCh                   ( set RP to autosleep address.                     )
   RDEPTH DROP                ( result is 0                                      )
;
```

# RP!

### 4.8.148 R-P-store

| | |
|---|---|
| **Purpose:** | Restore the Return Stack pointer. |
| **Category:** | Assembler instruction |
| **MARC4 Opcodes:** | 75 hex |
| **Stack Effect:** | RP!: EXP ( RPh RPl -- ) |
| | RET ( -- ) |
| **Stack Changes:** | RP! EXP: 2 elements are popped from the stack |
| | RET: Return Stack pointer modified |
| **Flags:** | Not affected |
| **X Y Registers:** | Not affected |
| **Bytes Used:** | 1 |
| **See Also:** | SP! SP@ >SP $xx RP@ >RP FCh R0 |

# RP!

**Example:**

```
VARIABLE R0 47 ALLOT            ( Return Stack: 48 nibbles 13 entry levels       )
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.             )
;

: $RESET
  > SP S0                       ( initialize the stack pointers.                 )
  > RP FCh                      ( set RP to autosleep address.                   )
  RDEPTH DROP                   ( result is 0                                    )
;
```

# >RP FCh, R0

## 4.8.149 To-RP Address

**Purpose:** Initialization of the Return Stack pointer.

**Category:** Assembler instruction

**MARC4 Opcodes:** R0: predefined data structure
>RP $xx = 79xx hex

**Stack Effect:** >RP $xx: EXP ( -- )
RET ( -- )   RP := $xx

**Stack Changes:** >RP $xx EXP: not affected
RET: new base address

**Flags:** Not affected

**X Y Registers:** Not affected

**Bytes Used:** 2

**See Also:** SP! SP@ >SP $xx RP@ RP!

# >RP FCh, R0

**Example:**

```
VARIABLE R0 47 ALLOT          ( Return Stack: 48 nibbles 13 entry levels       )
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.             )
;



: $RESET
   > SP S0                    ( initialize the stack pointers.                 )
   > RP FCh                   ( set RP to autosleep address.                   )
   RDEPTH DROP                ( result is 0                                    )
;
```

# S>D

## 4.8.150 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

# S>D

**Example:**

```
: Bytes & Nibbles

  4            (             --  4     )
  S>D          (          4  --  0  4  )
  25h D+       (  0  4  2  5  --  2  9  )
  D>S          (     2     9  --  9     )
  DROP         (           9  --        )
;
```

# SP@

## 4.8.151 S-P-fetch

**Purpose:** Fetch the Expression Stack pointer onto the TOS.

**Category:** Assembler instruction

**MARC4 Opcodes:** 70 hex

**Stack Effect:** EXP ( -- SPh SPl )

RET ( -- )

**Stack Changes:** EXP: 'stack pointer + 1' is pushed onto the stack

RET: not affected

**Flags:** Not affected

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** RP! RP@ >RP FCh SP! >SP S0

# SP@

**Example 1:**

```
VARIABLE S0 34 ALLOT \ Define EXP stack depth
VARIABLE R0 80 ALLOT \ Define 21 RET stack entries

( The qFORTH word DEPTH uses 'SP@' to push the EXP stack      )
( pointer onto the Expression Stack                           )

: DEPTH   SP@              ( -- SPh SPl                                 )
      S0                   ( SPh SPl -- SPh SPl S0h S0l                 )
      D-                   ( SPh SPl S0h S0l -- diffh diffl             )
      NIP                  ( 0 n -- n ) ( result only on nibble, max Fh )
      1-                   ( the value of SP on stack is incremented.   )
;

: $RESET
  > SP S0                  ( initialize the stack pointers.             )
  > RP NoRAM               ( Set RP to autosleep address.               )
  DEPTH  DROP              ( result is 0                                )
;
```

**Example 2:**

Purpose: Add the 4-bit number on TOS to a 8-bit byte in RAM

```
:  M+                      ( n addr - - )
  X! [+X]@ + [X-]!
  [X]@ 0 +C [X]!
;
:  SRAMINIT                (  - -  )
  >SP F9h
  SP@ X!
  0
  begin    DROP            Fh [X]! [X-]@ NOT
                           0h [X]! [X-]@ OR
                           TOG_BF ?LEAVE DROP
  X@ 1- OR                         ( Test all addresses except 00 & 01    )
  until                            ( Attention: RETURN address stack!     )
  Error_Flag !                     ( Save result in Error_Flag at FFh     )
  >SP      S0                      ( Reset STACKPOINTER again !!          )
  [ E O N ] ;                      ( Don't place in 'ZERO PAGE'           )
```

# SLEEP

## 4.8.152 Sleep

**Purpose:** The SLEEP instruction forces the MARC4 CPU into sleep mode, whereby the internal CPU clock is halted.

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 interrupt service routine is placed.

Note: It is not recommended to use the SLEEP instruction other than in the $RESET level or the $AUTOSLEEP routine because it might result in unwanted side effects within other interrupt routines. If any interrupt is active or pending, the SLEEP instruction will be executed in the same way as an NOP

**Category:** Interrupt handling/assembler instruction

**MARC4 Opcode:** 0F hex

**Stack Effect:** EXP ( - - )

RET ( - - )

**Stack Changes:** EXP: not affected

RET: not affected

**Flags:** I_ENABLE flag: set

CARRY and BRANCH flags: not affected

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** DI, EI, RTI, $AUTOSLEEP

# SLEEP

**Example:**

```
\ After POR, wait in power-down mode until a key is pressed to start the application

: System _Init
          Setup_Peripherals
          Enable_KeyInt              \ Enable INT1 for nest key input
          SLEEP                      \ Wait for INT1 to happen here
          NOP NOP                    \Allow INT processing
BEGIN     Port5 IN                   \ Wait until no key is pressed
          Port5 IN AND Fh =
UNTIL
          Choose_Display             \ Show power-up display
          1_Hz .Set_BaseTimer        \ Setup 1 Hz INT5
;
```

# SP!

## 4.8.153 S-P-store

**Purpose:** Restore the Expression Stack pointer.

**Category:** Assembler instruction

**MARC4 Opcodes:** 74 hex

**Stack Effect:** EXP ( SPh SPl -- )

RET ( -- )

**Stack Changes:** EXP: 2 elements are popped into the stack pointer

RET: not affected

**Flags:** Not affected

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** RP! RP@ >RP $xx SP@ >SP S0

# SP!

**Example:**

```
VARIABLE S0 34 ALLOT              \ Define EXP stack depth
VARIABLE R0 80 ALLOT              \ Define 21 RET stack entries


: SP_ex         DUP 0<>           (n - - n)
        IF      2Ah               \ This is a stupid example to re-adjust SP
            ELSE   2Fh
            THEN          SP!
;
```

# >SP S0

### 4.8.154 To-SP S-zero

| | |
|---|---|
| **Purpose:** | Initialization of the Expression Stack pointer. |
| **Category:** | Assembler instruction |
| **MARC4 Opcodes:** | S0: predefined data structure |
| | >SP $xx = 78xx |
| **Stack Effect:** | EXP ( -- ) |
| | RET ( -- ) |
| **Stack Changes:** | EXP: the Expression Stack pointer is initialized |
| | RET: not affected |
| **Flags:** | Not affected |
| **X Y Registers:** | Not affected |
| **Bytes Used:** | 2 |
| **See Also:** | RP! RP@ >RP $xx SP@ SP! |

# >SP SO

**Example 1:**

```
VARIABLE S0 34 ALLOT \ Define EXP stack depth
VARIABLE R0 80 ALLOT \ Define 21 RET stack entries


( The qFORTH word DEPTH uses 'SP@' to push the EXP stack                 )
( pointer onto the Expression Stack                                      )
: DEPTH   SP@               (               -- SPh SPl                   )
     S0                     ( SPh SPl       -- SPh SPl S0h S0l           )
     D-                     ( SPh SPl S0h S0l -- diffh diffl             )
     NIP                    ( 0 n -- n ) ( result only on nibble, max Fh )
     1-                     ( the value of SP on stack is incremented.   )
;


: $RESET
  > SP S0                   ( initialize the stack pointers.             )
  > RP NoRAM                ( Set RP to autosleep address.               )
  DEPTH DROP                ( result is 0                                )
;
```

**Example 2:**

Purpose: Add the 4-bit number on TOS to a 8-bit byte in RAM

```
:  M+                       ( n addr - - )
  X! [+X]@ + [X-]!
  [X]@ 0 +C [X]!
;
:  SRAMINIT                 ( - - )
  >SP F9h
  SP@ X!
  0
  begin   DROP
          Fh [X]! [X-]@ NOT
          0h [X]! [X-]@ OR
          TOG_BF ?LEAVE DROP
          X@ 1- OR                  ( Test all addresses except 00 & 01   )
  until                             ( Attention: RETURN address stack!    )
  Error_Flag !                      ( Save result in Error_Flag at FFh    )
  >SP     S0                        ( Reset STACKPOINTER again !!         )
[ E O N ] ;                         ( Don't place in 'ZERO PAGE'          )
```

# SET_BCF

### 4.8.155 Set-BRANCH-and-CARRY-flag

| | |
|---|---|
| **Purpose:** | Set the state of the BRANCH and CARRY flag 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 flag set |
| | BRANCH flag set |
| **X Y Registers:** | Not affected |
| **Bytes Used:** | 1 |
| **See Also:** | TOG_BCF CLR_BCF |

# SET_BCF

**Example:**

```
: Error                     ( what happens in case of error:        )
  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 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                      )
;
```

# SWAP

## 4.8.156 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 OVER |

# SWAP

**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'

**4.8.157 Software-interrupt -zero ... Software -interrupt-seven**

**Purpose:** These qFORTH words generate a software interrupt. They allow the programmer to postpone less important tasks 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 (NOP gives time for task switch )

END-CODE

**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

# 'SWI0 ... SW17'

**Example:**

```
: INT3                    ( software triggered interrupt #3          )
  UpdateLCD               ( update LCD display                       )
;

: INT5                    ( external hardware interrupt - level 5    )
  ScanKeys                ( do most important action here and now    )
  SWI3                    ( more action later, after this job is     )
  Count 1+!               ( finished; activated from the interrupt   )
  Count @ 0=              ( logic after the higher prioritised       )
  IF Overflow THEN        ( job is completed.                        )
;                         ( compiler translates ';' to 'RTI'.        )
```

# T+!

## 4.8.158 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-!

# T+!

**Example:**

```
3 ARRAY 3Nibbles AT 40h

: Triples
  123h 3Nibbles 3!          ( store 123 in the 3 nibbles array.          )
  321 3Nibbles T+!          ( 123 + 321 = 444                            )
  3Nibbles 3@ 3>R           ( save the result onto Return Stack.         )
  123h 3Nibbles T-!         ( 444 - 123 = 321                            )
  DROPR                     ( forget the saved result on Return Stack.   )
;
```

# T-!

### 4.8.159 T-minus-store

**Purpose:** Subtracts the TOS 12-bit value from a 12-bit variable in RAM and stores 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 are changed

**Bytes Used:** 17 + ’M+’

**See Also:** 3@ 3! T+!

# T-!

**Example:**

```
3 ARRAY 3Nibbles AT 40h

: Triples
  123h 3Nibbles 3!            ( store 123 in the 3 nibbles array.         )
  321 3Nibbles T+!            ( 123 + 321 = 444                           )
  3Nibbles 3@ 3>R             ( save the result onto Return Stack.        )
  123h 3Nibbles T-!           ( 444 - 123 = 321                           )
  DROPR                       ( forget the saved result on Return Stack   )
;
```

# TD+!

**4.8.160 T-D-plus-store**

**Purpose:** ADD the 8-bit number on the stack to a 12-bit element in RAM and store the result in that variable. On function entry, the start base address of the array is the TOS value.

**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 are changed

**Bytes Used:** 15 + ’M+’

# TD+!

**Example:**

```
3 ARRAY 3Nibbles

: TripDigi
  F87h 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-!

### 4.8.161 T-D-minus-store

**Purpose:** SUBTRACT the 8-bit number on the stack from an 12-bit element in RAM and store the result in that variable. On function entry, the start base address of the array is the TOS value.

**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 are changed

**Bytes Used:** 17 + 'M+'

**See Also:** 3@ 3! T-!

# TD-!

**Example:**

```
3 ARRAY 3Nibbles

: TripDigi
  F87h 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

### 4.8.162 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 is 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 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:** 0

**See Also:** IF   ELSE

# THEN

**Example:**

```
: IfElseThen
  1 2 <=               ( is 1 <= 2 ?                              )
  IF                   ( yes, so the If block is 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

## 4.8.163 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

# TOGGLE

**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

### 4.8.164 Toggle-BRANCH -flag

**Purpose:** Toggle the state of the BRANCH flag in the MARC4 condition code register.

If the BRANCH flag = 1 THEN reset 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

# TOG_BF

**Example:**

```
: Error                        ( what happens in case of error:       )
  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 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                     )
;
```

# TUCK

## 4.8.165 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

# TUCK

**Example:**

```
: TUCK-Example
  1 2                    (  -- 1 2      )
  TUCK                   ( 1 2 -- 2 1 2 )
  3DROP                  ( 2 1 2 --     )
;
```

# UNTIL

## 4.8.166 UNTIL

**Purpose:** art 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 BEGIN.

**Category:** Control structure

**Library Implementation:**

```
CODE UNTIL          ( complement BRANCH flag        )
        TOG_BF
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

# UNTIL

**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

## 4.8.167 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 the time of compilation, 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+1> 4-bit memory locations is 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

# VARIABLE

**Example:**

```
3 CONSTANT TimeLen

VARIABLE R0 27 ALLOT               ( return stack: 28 nibbles, used 21 for     )
                                   ( 7 levels of task switching.               )
                                   ( 7 nibbles are free for 1nibble variables. )
VARIABLE S0 19 ALLOT               ( 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.                     )

TimeLen ARRAY ActualTime           ( 3 nibble array.                           )
```

# WHILE

## 4.8.168 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 instructions 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              ( complement BRANCH flag    )
       TOG_BF
           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

# WHILE

**Example:**

```
: Example
  10 BEGIN              ( decrement the TOS , from 10 to 0.          )
       1- DUP           ( save TOS                                   )
       0<> WHILE        ( REPEAT decrement while TOS not equal 0     )
       DUP 1 OUT
  REPEAT                ( after each decrement the TOS contains      )
                        ( the value of the present index             )
;
```

# X@

### 4.8.169 X fetch

| | |
|---|---|
| **Purpose:** | The X register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing). |
| **Category:** | Assembler instruction |
| **MARC4 Opcode:** | 72 hex |
| **Stack Effect:** | EXP ( - - x_h x_l ) |
| | RET ( -- ) |
| **Stack Changes:** | EXP: 2 elements are pushed onto the stack |
| | RET: not affected |
| **Flags:** | CARRY flag not affected |
| | BRANCH flag not affected |
| **X Y Registers:** | Not affected |
| **Bytes Used:** | 1 |
| **See Also:** | X! |



4747A–4BMCU–01/04

## X@

**Example:**

```
: SRAMINIT                      ( -- )
  >SP F9h
  SP@ X!
  0
  begin     DROP        Fh [X]! [X-]@ NOT
                        0h [X]! [X-]@ OR
                        TOG_BF  ?LEAVE DROP
            X@ 1- OR                ( Test all addresses except 00 & 01       )
  until                             ( Attention: RETURN address stack!        )
  Error_Flag !                      ( Save result in Error_Flag at FFh        )

  >SP           S0                  ( Reset STACKPOINTER again !!             )
[ E O N ] ;                         ( Don't place in 'ZERO PAGE'              )
```

## X!

### 4.8.170 X store

**Purpose:** The X register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 76 hex

**Stack Effect:** EXP ( x_h x_l - - )

RET ( -- )

**Stack Changes:** EXP: 2 elements are popped from the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** X@ Y! Y@

# X!

**Example:**

Purpose: Add the 4-bit number on TOS to a 8-bit byte in RAM

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



:  SRAMINIT                          (  - -  )
   >SP F9h
   SP@ X!
   0
   begin        DROP          Fh [X]! [X-]@ NOT
                              0h [X]! [X-]@ OR
                              TOG_BF  ?LEAVE DROP
                X@ 1- OR                   ( Test all addresses except 00 & 01        )
   until                                   ( Attention: RETURN address stack!         )
   Error_Flag !                            ( Save result in Error_Flag at FFh         )

   >SP          S0                         ( Reset STACKPOINTER again !!              )
[ E O N ] ;                                ( Don't place in 'ZERO PAGE'               )
```

# [X]@

## 4.8.171 indirect X fetch

**Purpose:** The X register can be used as a pointer to access variables or arrays in the RAM. For the compilation of a qFORTH word which uses these MARC4 instructions directly, the qFORTH compiler switch $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 30 hex

**Stack Effect:** EXP ( - - n )

RET ( -- )

**Stack Changes:** EXP: 1 element is pushed onto the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** Used, but not changed

**Bytes Used:** 1

**See Also:** [X]! [Y]! [Y]@

## [X]@

**Example:**

Purpose: Add the 4-bit number on TOS to a 8-bit byte in RAM

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

## [+X]@

### 4.8.172 pre increment indirect X fetch

**Purpose:** The X register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 31 hex

**Stack Effect:** EXP ( - - n )

RET ( - - )

**Stack Changes:** EXP: 1 element is pushed onto the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** The X register is changed by this instruction

**Bytes Used:** 1

**See Also:** X! X@ [x]! [X]@ [+X]! [+Y]@

# [+X]@

**Example:**

Purpose: Add the 4-bit number on TOS to an 8-bit byte in RAM

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

# [X-]@

### 4.8.173 post-decrement indirect X fetch

**Purpose:** The X register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 32 hex

**Stack Effect:** EXP ( - - n )

RET ( - - )

**Stack Changes:** EXP: 1 element is pushed onto the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** The X register is changed by this instruction

**Bytes Used:** 1

**See Also:** X! X@ [X]@ [+X]@ [Y]@ [+Y]@ [Y-]@

## [X-]@

**Example:**

```
( >Array=   Compares two arrays, starting with the last element   )
( down to lower addresses. Maximal length: 16 elements            )
( n Addr1 Addr2 result is in branch flag                          )

: >Array=
  X! Y! 0 SWAP
  #DO [X-]@ [Y-]@ - OR
  #LOOP 0=
;
```

# [X]!

### 4.8.174 indirect X store

**Purpose:** The X register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 38 hex

**Stack Effect:** EXP ( n - - )

RET ( - - )

**Stack Changes:** EXP: 1 element is popped from the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** The X register is not changed by this instruction

**Bytes Used:** 1

**See Also:** X! X@ [+X]! [X-]! [Y]! [+Y]! [Y-]!

# [X]!

**Example:**

Purpose: Add the 4-bit number on TOS to an 8-bit byte in RAM

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

# [+X]!

### 4.8.175 pre increment indirect X store

**Purpose:** The X register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 39 hex

**Stack Effect:** EXP ( n - - )

RET ( - - )

**Stack Changes:** EXP: 1 element is popped from the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** The X register is changed by this instruction

**Bytes Used:** 1

**See Also:** X! X@ [X]! [X-]! [Y]! [+Y]! [Y-]!

# [+X]!

**Example:**

Purpose: Add the 4-bit number on TOS to a 8-bit byte in RAM

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

# [X-]!

### 4.8.176 post decrement indirect X store

**Purpose:** The X register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 3A hex

**Stack Effect:** EXP ( n - - )

RET ( - - )

**Stack Changes:** EXP: 1 element is popped from the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** The X register is changed by this instruction

**Bytes Used:** 1

**See Also:** X! X@ [X]@ [+X]@ [Y]! [+Y]! [Y-]!

# [X-]!

**Example:**

Purpose: Add the 4-bit number on TOS to a 8-bit byte in RAM

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

# XOR

### 4.8.177 XOR

| | |
|---|---|
| **Purpose:** | XOR leaves the bit-wise logical exclusive-or of the top two values on the Expression 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 |

# XOR

**Example:**

```
: Error                            ( what should happen in case of error:    )
  3R@   3
  #DO                              ( show PC, where CPU fails                )
      I OUT [ E 0 ]                ( suppress compiler warnings.             )
  #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 is not        )
      Error                        ( activated                               )
  THEN                             (                                         )
  1100b 1100b XOR                  ( BRANCH flag is   set [result = 0]       )
  0000b <> IF                      ( XOR top two values; error is not        )
      Error                        ( activated.                              )
  THEN                             (                                         )
;
```

# Y@

### 4.8.178 Y fetch

| | |
|---|---|
| **Purpose:** | The Y register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing). |
| **Category:** | Assembler instruction |
| **MARC4 Opcode:** | 73 hex |
| **Stack Effect:** | EXP ( - - x_h x_l ) |
| | RET ( -- ) |
| **Stack Changes:** | EXP: 2 elements are pushed onto the stack |
| | RET: not affected |
| **Flags:** | CARRY flag not affected |
| | BRANCH flag not affected |
| **X Y Registers:** | Not affected |
| **Bytes Used:** | 1 |
| **See Also:** | Y! |

# Y@

**Example 1:**

```
4 ARRAY Ramaddr AT 30h

$OPTIMIZE - XYTRACE, -XY@!
:  Y-STORE
          Ramaddr [3] Y!          ( assign the variable Ramaddr to the Y register)
          5 [Y-]!                 ( store 5 in the RAM location 33               )
          6 [Y-]!                 ( store 6 in the RAM location 32               )
          2 [Y-]!                 ( store 2 in the RAM location 31               )
          7 [+Y]!                 ( store 7 in the RAM location 31               )
;
$OPTIMIZE +XY@!, +XYTRACE
```

**Example 2:**

Purpose: Substract a 4-bit number on TOS from 8-bits in RAM

```
: M-!                                          ( n addr - - )
  Y! [+Y]@ - [Y-]! [Y]@ 0 -c [Y]!
;
```

# Y!

### 4.8.179 Y store

**Purpose:** The Y register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 77 hex

**Stack Effect:** EXP ( x_h x_l - - )

RET ( -- )

**Stack Changes:** EXP: 2 elements are popped from the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** Not affected

**Bytes Used:** 1

**See Also:** Y@ Y! X@

## Y!

**Example 1:**

```
4 ARRAY Ramaddr AT 30h

$OPTIMIZE - XYTRACE, -XY@!
: Y-STORE
        Ramaddr [3] Y!      ( assign the variable Ramaddr to the Y register)
        5 [Y-]!             ( store 5 in the RAM location 33            )
        6 [Y-]!             ( store 6 in the RAM location 32            )
        2 [Y-]!             ( store 2 in the RAM location 31            )
        7 [+Y]!             ( store 7 in the RAM location 31            )
;
$OPTIMIZE +XY@!, +XYTRACE
```

**Example 2:**

Purpose: Substract a 4-bit number on TOS from 8-bits in RAM

```
: M-!                                   ( n addr - - )
  Y! [+Y]@ - [Y-]! [Y]@ 0 -c [Y]!
;
```

## [Y]@

### 4.8.180 indirect Y fetch

**Purpose:** The Y register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 34 hex

**Stack Effect:** EXP ( - - n )

RET ( -- )

**Stack Changes:** EXP: 1 element is pushed onto the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** Used, but not changed

**Bytes Used:** 1

**See Also:** [Y]! [Y]! [Y]@

# [Y]@

**Example 1:**

```
4 ARRAY Ramaddr AT 30h

$OPTIMIZE - XYTRACE, -XY@!
: Y-STORE
        Ramaddr [3] Y!      ( assign the variable Ramaddr to the Y register)
        5 [Y-]!             ( store 5 in the RAM location 33              )
        6 [Y-]!             ( store 6 in the RAM location 32              )
        2 [Y-]!             ( store 2 in the RAM location 31              )
        7 [+Y]!             ( store 7 in the RAM location 31              )
;
$OPTIMIZE +XY@!, +XYTRACE
```

**Example 2:**

Purpose: Substract a 4-bit number on TOS from 8 bits in RAM

```
: M-!                                       ( n addr - - )
  Y! [+Y]@ - [Y-]! [Y]@ 0 -c [Y]!
;
```

## [+Y]@

### 4.8.181 pre increment indirect Y fetch

| | |
|---|---|
| **Purpose:** | The Y register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing). |
| **Category:** | Assembler instruction |
| **MARC4 Opcode:** | 35 hex |
| **Stack Effect:** | EXP ( - - n ) |
| | RET ( - - ) |
| **Stack Changes:** | EXP: 1 element is pushed onto the stack |
| | RET: not affected |
| **Flags:** | CARRY flag not affected |
| | BRANCH flag not affected |
| **X Y Registers:** | The Y register is changed by this instruction |
| **Bytes Used:** | 1 |
| **See Also:** | Y! Y@ [Y]! [Y]@ [+Y]! [+Y]@ |

# [+Y]@

**Example 1:**

```
4 ARRAY Ramaddr AT 30h

$OPTIMIZE - XYTRACE, -XY@!
: Y-STORE
        Ramaddr [3] Y!        ( assign the variable Ramaddr to the Y register)
        5 [Y-]!               ( store 5 in the RAM location 33      )
        6 [Y-]!               ( store 6 in the RAM location 32      )
        2 [Y-]!               ( store 2 in the RAM location 31      )
        7 [+Y]!               ( store 7 in the RAM location 31      )
;
$OPTIMIZE +XY@!, +XYTRACE
```

**Example 2:**

Purpose: Substract a 4-bit number on TOS from 8-bits in RAM

```
: M-!                                    ( n addr - - )
  Y! [+Y]@ - [Y-]! [Y]@ 0 -c [Y]!
;
```

# [Y-]@

### 4.8.182 post decrement indirect Y fetch

**Purpose:** The Y register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing).

**Category:** Assembler instruction

**MARC4 Opcode:** 36 hex

**Stack Effect:** EXP ( - - n )

RET ( - - )

**Stack Changes:** EXP: 1 element is pushed onto the stack

RET: not affected

**Flags:** CARRY flag not affected

BRANCH flag not affected

**X Y Registers:** The Y register is changed by this instruction

**Bytes Used:** 1

**See Also:** Y! Y@ [Y]@ [+Y]@ [X]@ [+X]@ [X-]@

# [Y-]@

**Example:**

```
( >Array=   Compares two arrays, starting with the last element        )
( down to lower addresses. Maximal length: 16 elements                  )
( n Addr1 Addr2 result is in branch flag                                )

: >Array=
   X! Y! 0 SWAP
   #DO [X-]@ [Y-]@ - OR
   #LOOP 0=
;
```

# [Y]!

### 4.8.183 indirect Y store

| | |
|---|---|
| **Purpose:** | The Y register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing). |
| **Category:** | Assembler instruction |
| **MARC4 Opcode:** | 3C hex |
| **Stack Effect:** | EXP ( n - - ) |
| | RET ( - - ) |
| **Stack Changes:** | EXP: 1 element is popped from the stack |
| | RET: not affected |
| **Flags:** | CARRY flag not affected |
| | BRANCH flag not affected |
| **X Y Registers:** | The Y register is not changed by this instruction |
| **Bytes Used:** | 1 |
| **See Also:** | Y! Y@ [+Y]! [Y-]! [X]! [+X]! [X-]! |

# [Y]!

**Example 1:**

```
4 ARRAY Ramaddr AT 30h

$OPTIMIZE - XYTRACE, -XY@!
:  Y-STORE
          Ramaddr [3] Y!          ( assign the variable Ramaddr to the Y register)
          5 [Y-]!                 ( store 5 in the RAM location 33              )
          6 [Y-]!                 ( store 6 in the RAM location 32              )
          2 [Y-]!                 ( store 2 in the RAM location 31              )
          7 [+Y]!                 ( store 7 in the RAM location 31              )
;
$OPTIMIZE +XY@!, +XYTRACE
```

**Example 2:**

Purpose: Substract a 4-bit number on TOS from 8 bits in RAM

```
: M-!                                              ( n addr - - )
  Y! [+Y]@ - [Y-]! [Y]@ 0 -c [Y]!
;
```

# [+Y]!

### 4.8.184 pre increment indirect Y store

| | |
|---|---|
| **Purpose:** | The Y register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing). |
| **Category:** | Assembler instruction |
| **MARC4 Opcode:** | 3D hex |
| **Stack Effect:** | EXP ( n - - ) |
| | RET ( - - ) |
| **Stack Changes:** | EXP: 1 element is popped from the stack |
| | RET: not affected |
| **Flags:** | CARRY flag not affected |
| | BRANCH flag not affected |
| **X Y Registers:** | The Y register is changed by this instruction |
| **Bytes Used:** | 1 |
| **See Also:** | Y! Y@ [Y]! [Y-]! [X]! [+X]! [X-]! |

# [+Y]!

**Example 1:**

```
4 ARRAY Ramaddr AT 30h

$OPTIMIZE - XYTRACE, -XY@!
: Y-STORE
        Ramaddr [3] Y!        ( assign the variable Ramaddr to the Y register)
        5 [Y-]!               ( store 5 in the RAM location 33              )
        6 [Y-]!               ( store 6 in the RAM location 32              )
        2 [Y-]!               ( store 2 in the RAM location 31              )
        7 [+Y]!               ( store 7 in the RAM location 31              )
;
$OPTIMIZE +XY@!, +XYTRACE
```

**Example 2:**

Purpose: Substract a 4-bit number on TOS from 8 bits in RAM

```
: M-!                                          ( n addr - - )
  Y! [+Y]@ - [Y-]! [Y]@ 0 -c [Y]!
;
```

# [Y-]!

**4.8.185 post decrement indirect Y store**

| | |
|---|---|
| **Purpose:** | The Y register can be used as a 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 $OPTIMIZE-XYTRACE should be turned off (no optimizing). |
| **Category:** | Assembler instruction |
| **MARC4 Opcode:** | 3E hex |
| **Stack Effect:** | EXP ( n - - ) |
| | RET ( - - ) |
| **Stack Changes:** | EXP: 1 element is popped from the stack |
| | RET: not affected |
| **Flags:** | CARRY flag not affected |
| | BRANCH flag not affected |
| **X Y Registers:** | The Y register is changed by this instructions |
| **Bytes Used:** | 1 |
| **See Also:** | Y! Y@ [Y]@ [+Y]@ [Y]! [+X]! [X-]! |

# [Y-]!

**Example 1:**

```
4 ARRAY Ramaddr AT 30h

$OPTIMIZE - XYTRACE, -XY@!
:  Y-STORE
          Ramaddr [3] Y!        ( assign the variable Ramaddr to the Y register   )
          5 [Y-]!               ( store 5 in the RAM location 33                  )
          6 [Y-]!               ( store 6 in the RAM location 32                  )
          2 [Y-]!               ( store 2 in the RAM location 31                  )
          7 [+Y]!               ( store 7 in the RAM location 31                  )
;
$OPTIMIZE +XY@!, +XYTRACE
```

**Example 2:**

Purpose: Substract a 4-bit number on TOS from 8 bits in RAM

```
: M-!                                                   ( n addr - - )
  Y! [+Y]@ - [Y-]! [Y]@ 0 -c [Y]!
;
```

# Index of “Detailed Description of the qFORTH Language”

## Symbols



## Numerics



## A



## B



## C



## D






4747A–4BMCU–01/04

4747A–4BMCU–01/04

## Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

## Regional Headquarters

*Europe*
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

*Asia*
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

*Japan*
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

## Atmel Operations

*Memory*
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

*Microcontrollers*
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

*ASIC/ASSP/Smart Cards*
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

*RF/Automotive*
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

*Biometrics/Imaging/Hi-Rel MPU/*
*High Speed Converters/RF Datacom*
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80


*Literature Requests*
www.atmel.com/literature




4747A–4BMCU–01/04 /xM