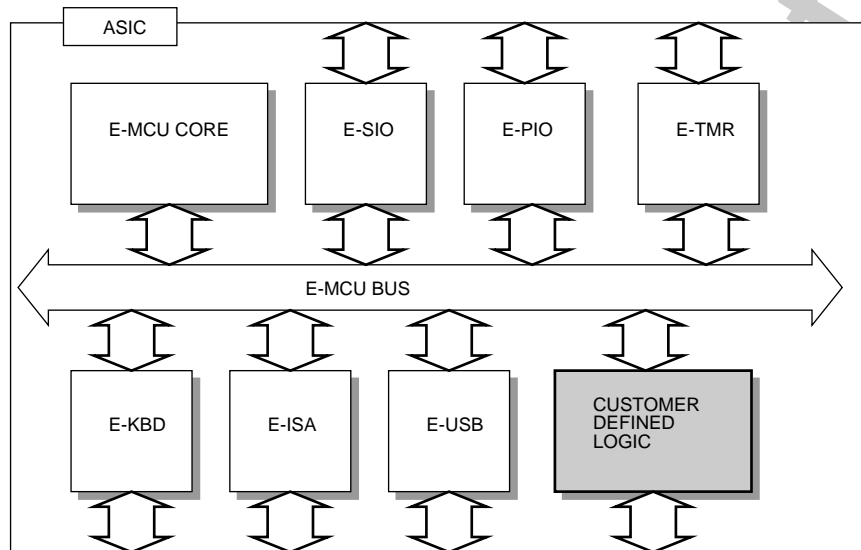


# E-MCU embedded microcontroller



## Key features of the E-MCU-8a:

- Easy to program, efficient Harvard architecture CPU
- Fully static core design
- Up to 64k of code space in ROM and 256 bytes data space in RAM
- OpCodes optimised to the application
- Single-phase CPU clock
- Custom or standard peripherals – serial I/O, parallel I/O, timers etc.

## Standard peripherals available:

- Serial I/O with modem control and baud rate generator
- Two channel system timer and prescaler
- Bidirectional parallel I/O
- ISA bus peripheral interface
- PC-AT and PS/2 keyboard/mouse interface
- USB serial bus interface.

© Future Technology Devices International Limited, 1996.

Neither the whole nor any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material or electronic form without the prior written consent of the copyright holder.

This product and its documentation are supplied on an as is basis and no warranty as to their suitability for any particular purpose is either made or implied. Future Technology Devices will not accept any claim for damages howsoever arising as a result of use or failure of this product. Your statutory rights are not affected.

This product or any variant of it is not intended for use in any medical appliance, device or system in which the failure of the product might reasonably be expected to result in personal injury.

Datasheet: E-MCU-8a  
Version: 0.5b  
Issued: 20 June 1996

**Ordering and distributor information:**

Future Technology Devices International  
St George's Studios  
93/97 St George's Road  
Glasgow  
G3 6JA  
UK

Telephone: (44) 0141 353 2565

Fax: (44) 0141 353 2656

Email: [ftdi@msn.com](mailto:ftdi@msn.com)

WWW:

Future Technology Devices International  
(US address)

Telephone:

Fax:

Email:

WWW:

---

## Contents

- 1 Overview of the E-MCU-8A 5
  - 1.1 About the FTDI embedded microcontroller 5
    - Traditional design approaches 5
    - The FTDI approach 5
    - Customisation 5
    - The E-MCU-8A core 5
  - 1.2 Key features 5
  - 1.3 Suitable applications 5
  - 1.4 About this document 6
- 2 Pin information 7
  - 2.1 Pin information 8
    - ROMD0 – ROMD7 (Active High) 8
    - ROMA0 – ROMA15 (Active High) 8
    - MEMA0 – MEMA7 (Active High) 8
    - MDO0 – MDO7 (Active High) 8
    - MDI0 – MDI7 (Active High) 8
    - MWE# (Active Low) 8
    - ROMWR# (Active Low) 8
    - MWAIT (Active High) 8
    - MRD# (Active Low) 8
    - ROMRD# (Active Low) 8
    - BLD (Active High) 8
    - BRLD (Active High) 8
    - BINC (Active High) 8
    - ROMD0 – ROMD7 (Active High) 8
    - IRQ (Active High) 8
    - DIRQ/DIRQEXT (Active High) 8
    - CLKIN (Active High) 8
    - RSTIN (Active High) 8
    - RST8K (Active High) 8
    - External wait state generator: 8
- 3 Architecture Overview - Programmers' Model 9
  - 3.1 Major blocks 9
    - Arithmetic Logic Unit (ALU) 9
    - Flags register 9
    - Program Address counter 9
    - Stack pointer 10
    - Timing and sequence block 10
    - Control logic 10
    - Temporary registers 10
  - 3.2 Address space 10
    - Program Space 10
    - Data Space 10
    - Reset Condition 11
  - 3.3 Interrupts 11
    - Interrupt 11
    - Debug interrupt 11

- 
- 3.4 Flags register 11
  - 4 Instruction set 12
    - Opcodes 12
    - Length in bytes 12
    - Example syntax 12
  - 5 Timing diagrams 18
    - 5.1 Program ROM/RAM data reads of 0, 1, 2 and 3 wait states 18
    - 5.2 Data memory I/O reads 18
    - 5.3 Data memory I/O write timing 19
    - 5.4 ROM write timings 20
    - 5.5 Reset operation (zero wait states ROM read) 21
    - 5.6 Subroutine call taken (Zero wait states ROM read) 22
    - 5.7 Return from subroutine 23
    - 5.8 Interrupt call 24
    - 5.9 Interrupt routine return 25

Preliminary information

# 1 Overview of the E-MCU-8A

This document provides preliminary information on the Future Technology ASIC Embedded Microcontroller E-MCU-8A.

## 1.1 About the FTDI embedded microcontroller

### Traditional design approaches

Traditional microcontrollers are based on old 8-bit CPU designs. These require specialist knowledge of the CPU, its architecture and limitations, and programming environment. Because the controller is not optimised for the application it can require a large amount of program space, reducing efficiency and increasing the costs of ASIC development.

### The FTDI approach

The FTDI embedded microcontroller provides an alternative approach with the emphasis on:

- Customisable CPU functions
- Ease of programming, removing the need for a detailed understanding of the architecture
- Supplied in standard VHDL format for portability between silicon vendors
- Efficient code
- Energy-efficient implementation

### Customisation

The FTDI embedded microcontroller provides greater code efficiency and cost savings by featuring a set of CPU functions which can be customised for each application. Program memory within the CPU dispenses with the need for registers, so that no detailed understanding of the CPU architecture is needed to program it.

Compound instructions (for example  $A=B+C$ ) mean that throughput is very high. A single core can therefore be used to provide many different functions without duplication of the core logic.

### The E-MCU-8A core

The E-MCU-8A core can typically be clocked at up to 33 MHz using a 0.8micron or better ASIC process. With an average instruction cycle of five to six clocks this gives a speed of up to 6MIPS.

The E-MCU-8A core is energy efficient. It is interrupt driven with an interrupt latency of a few hundred nanoseconds (at 33MHz). When not processing the E-MCU-8A sits in a low-power HALT state with only a few gates clocked, in order to minimise power consumption.

## 1.2 Key features

Key features of the E-MCU-8A are:

- Easy to program, efficient Harvard architecture CPU
- Fully static core design
- Up to 64k of code space in ROM and 256 bytes data space in RAM
- OpCodes optimised to the application
- Single-phase CPU clock
- Custom or standard peripherals – serial I/O, parallel I/O, timers etc.

Standard peripherals available:

- Serial I/O with modem control and baud rate generator
- Two channel system timer and prescaler
- Bidirectional parallel I/O
- ISA bus peripheral interface
- PC-AT and PS/2 keyboard/mouse interface.

Our design service provides support for custom peripherals and other logic.

## 1.3 Suitable applications

The FTDI E-MCU-8A is suitable for many ASIC/custom logic designs requiring a small, fast embedded CPU, for example:

- cost-effective USB and PC peripherals
- consumer markets
- automotive markets
- control

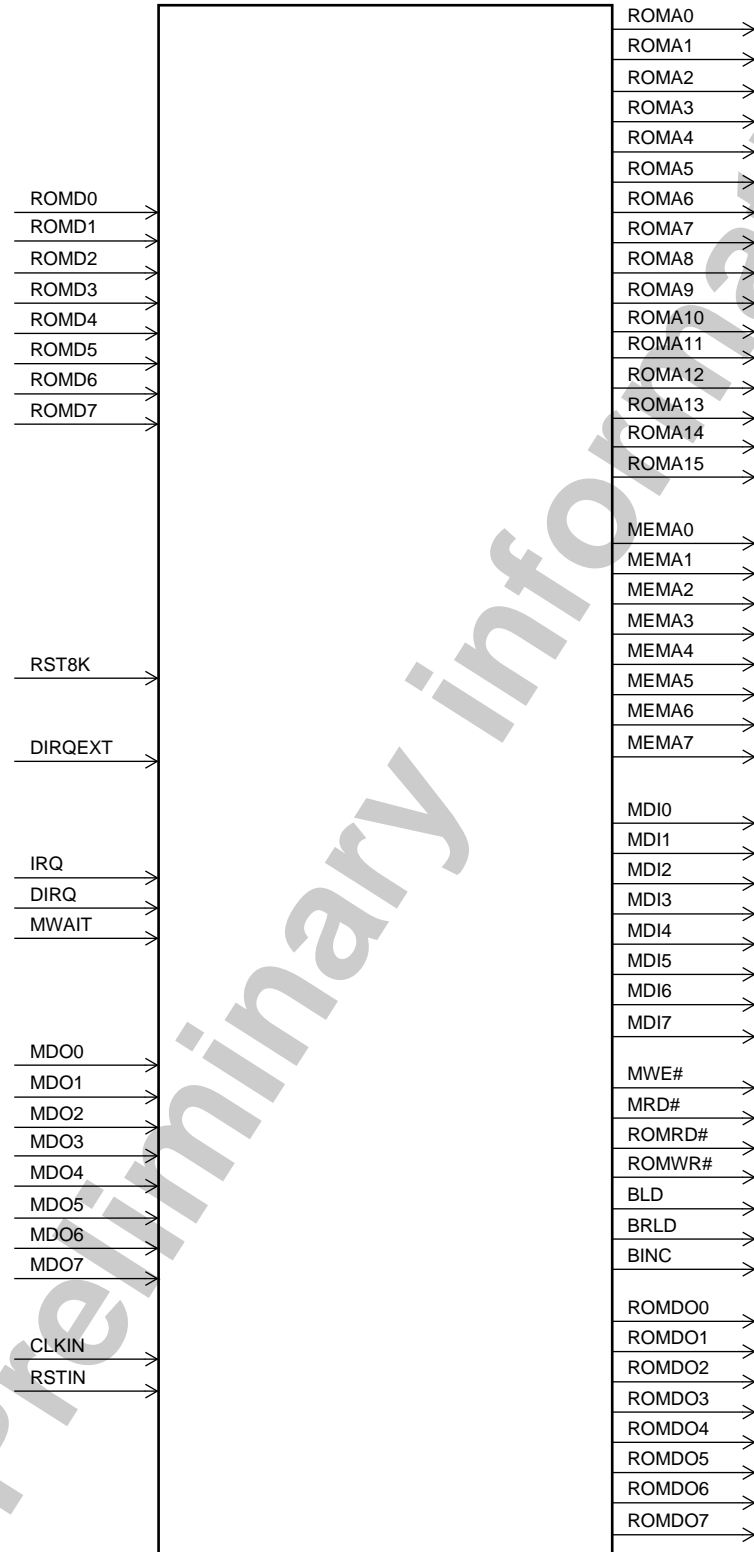
#### 1.4 About this document

This document provides an introduction to the E-MCU-8A microcontroller. Before using this preliminary information as the basis of any product please check with FTDI/F-Tek that it has not been superseded.

Preliminary information

## 2 Pin information

The following diagram shows the standard Signal Interface to the E-MCU-8A Core Macro.



## 2.1 Pin information

### **ROMD0 – ROMD7 (Active High)**

CPU data bus for the program memory.

### **ROMA0 – ROMA15 (Active High)**

CPU address bus for the program memory allowing up to 64k of program addressing space.

### **MEMA0 – MEMA7 (Active High)**

Address bus for the internal data memory and peripherals – up to 256 bytes.

### **MDO0 – MDO7 (Active High)**

Data output from the internal data memory RAM and peripherals. This is an input to the microcontroller.

### **MDI0 – MDI7 (Active High)**

Data input to the internal data memory RAM and peripherals. This is an output from the microcontroller.

### **MWE# (Active Low)**

Memory write status. Enables data writes to data memory RAM. Active low. Data is valid by the next positive CPU clock edge.

### **ROMWR# (Active Low)**

Used to write to RAM-based program memory.

### **MWAIT (Active High)**

Asserting MWAIT causes the E-MCU-8A core to remain in its current state on all clock edges where MWAIT is active. This can be used by external logic to generate 'wait states' for access to slow memory and peripherals.

### **MRD# (Active Low)**

Memory read status. Indicates that the E-MCU-8A will read data from the data RAM on the next positive CPU clock edge.

### **ROMRD# (Active Low)**

Indicates read from E-MCU-8A program memory on next active CPU positive clock edge.

### **BLD (Active High)**

Internal status indicating program counter load on the next positive CPU clock edge.

### **BRLD (Active High)**

Internal status indicating program counter reload on the next positive CPU clock edge.

### **BINC (Active High)**

Internal status indicating program counter increment on the next positive CPU clock edge.

### **ROMD0 – ROMD7 (Active High)**

Output data to the CPU program memory. Used to write data if the memory is implemented using RAM or flash memory technology.

### **IRQ (Active High)**

E-MCU-8A interrupt line, level triggered.

### **DIRQ/DIRQEXT (Active High)**

Debug interrupt. Can be connected to address decoding logic to provide hardware breakpoints at specific addresses under control of a debugging program.

### **CLKIN (Active High)**

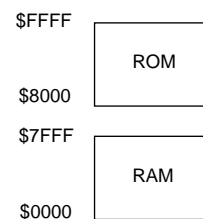
CPU clock input: clocks the CPU core on each rising edge of this signal.

### **RSTIN (Active High)**

Resets the E-MCU-8A core to a known state. The value of each part of the core on reset is described in section 3.1 *Major blocks* on page 9.

### **RST8K (Active High)**

Used to cause the CPU to execute program code at \$8000 (32Kbyte) on reset instead of \$00. This is used for a system configuration of:



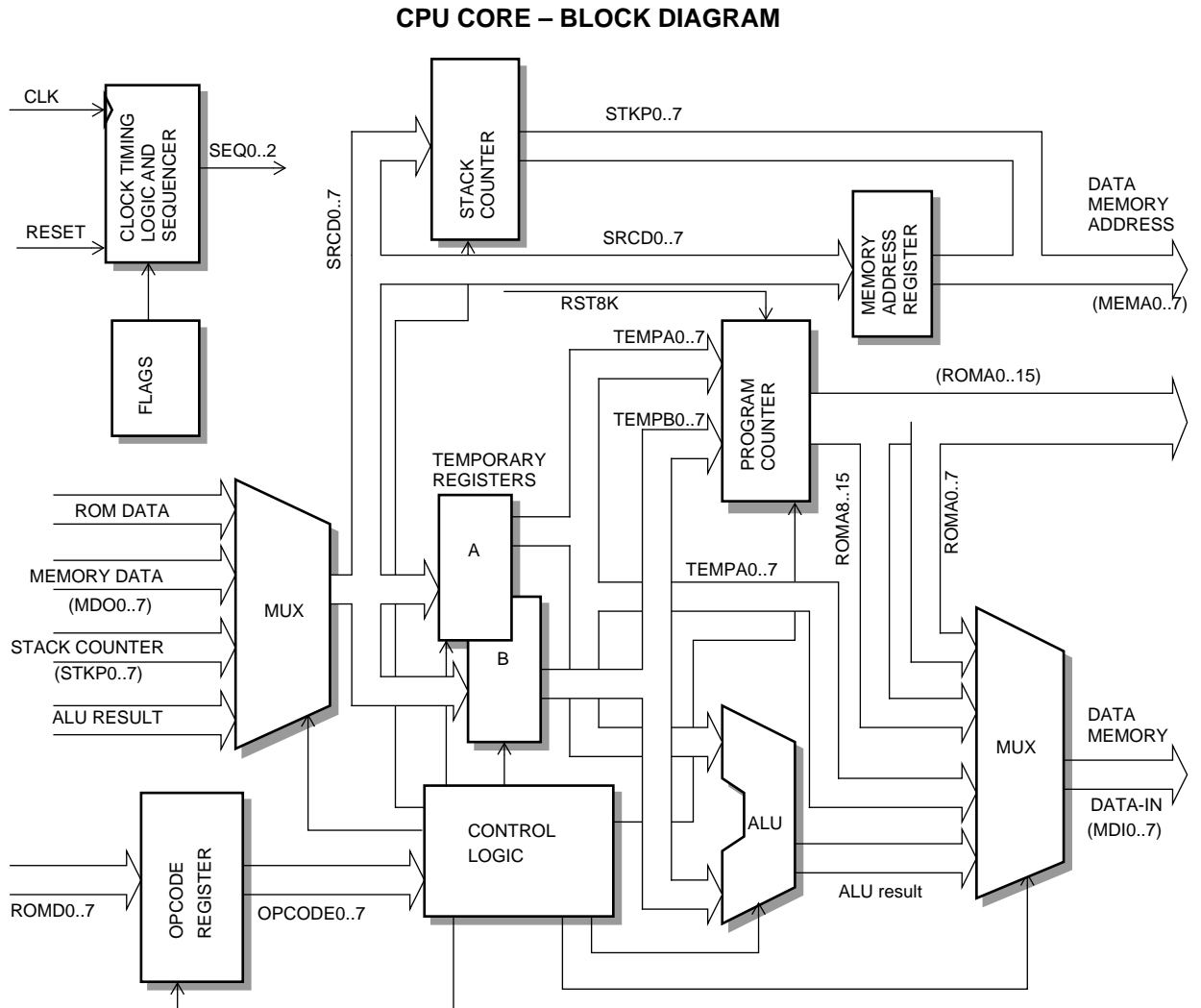
The ROM program can contain a loader program to load the RAM with another program.

### **External wait state generator:**

Uses MWE#, ROMWR#, BLD, BRLD, BINC to generate read and write strobes. It also controls MWAIT to allow 0-3 wait states to be added for ROM reads. The write strobes indicate valid data on their rising edge. This is for an easy connection to standard ROM/RAM technology. It can be used to control buffer direction for a single bi-directional data bus.

### 3 Architecture Overview - Programmers' Model

The following diagram represents the internal structure of the E-MCU-8A main CPU:



#### 3.1 Major blocks

The CPU core consists of the following major blocks:

##### Arithmetic Logic Unit (ALU)

This performs all the operations which change data including the arithmetic, the Boolean, shift and rotate operations.

##### Flags register

The flags are set or cleared by various instructions so that the Jump or Jump to subroutine instructions may make use of them.

See 3.4 *Flags register* on page 11 for more details.

##### Program Address counter

This is the address pointer for the program code. At reset its value is set to \$0000 or \$8000 depending on the signal RST8K. On exiting the RESET state, the CPU will commence executing instructions from this address.

The program counter will fetch sequential instructions as code is executed. It will be loaded on Jump/Jump to subroutine

instructions or an unmasked external interrupt.

**Stack pointer**

A counter which keeps track of the CPU stack operations. On reset its value is \$80. On subroutine/interrupt calls it decrements and stores the return address at that location in data memory. On return it loads the program counter from the data memory then increments.

The stack pointer may be initialized to a different value with the load stack pointer instruction. This would be of use if more or less than 128 Bytes of RAM is in the system.

**Timing and sequence block**

This block controls the reset and internal sequence counter of the CPU core. The sequence counter allows up to eight internal operations per instruction opcode.

**Control logic**

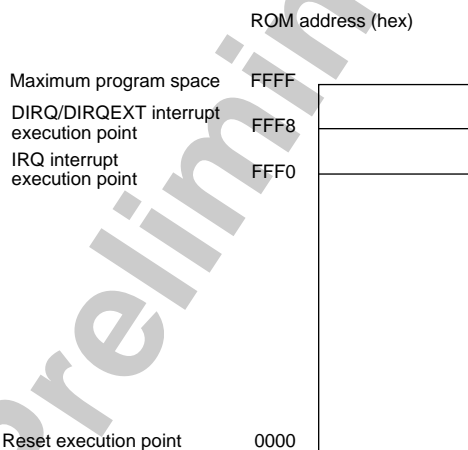
This block decodes the opcodes and controls the sequence counter, ALU and other internal functions in the CPU core.

**Temporary registers**

These are internal registers, not accessible to the programmer, and used to hold intermediate results during instruction execution.

3.2 Address space

**Program Space**



The program address space for a program memory size of up to 64K. In an embedded application this would typically be ROM, but

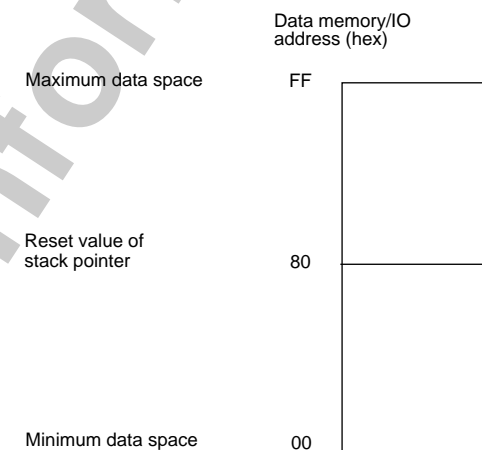
in some applications the program code could also be contained in SRAM or flash memory.

For some applications a much smaller physical memory would be used, which will be aliased throughout the address space.

The E-MCU-8A assembler has define constructs which inform the assembler of the size and location of the program memory, in order to calculate the physical address correctly and to reposition the interrupt execution code if required.

**Note:** smaller ROM sizes than 64k can be accommodated by leaving the upper ROM address lines disconnected.

**Data Space**



It is assumed that normally 128 bytes of RAM will be used in the system. This RAM is used for data and stack operations. Data variables are normally assigned from address 0 upwards, while the stack descends from the top or available data memory.

A subroutine call will use 2 bytes of stack space. An interrupt call will use 3 bytes of stack space. If the program to be written is well understood and 128 Bytes is felt to be an unnecessary use of silicon, then the data RAM can be reduced.

The stack pointer should be set to the address above the top of this new size of RAM if any stack operations are to be used. Conversely if more RAM is needed for more variables or deeper subroutine nesting then

it can be increased up to a maximum of 256 bytes.

The I/O ports are memory mapped into this data memory region and would normally start above address \$80 (if 128 bytes of RAM is used). A combined maximum of 256 byte-wide locations is allowed for RAM and I/O.

**Reset Condition**

The RESET signal is used to force the core into a known state. On reset, the program address counter is set to \$0000 (or \$8000 if RST8K is asserted) and the stack pointer is set to \$80.

Program execution begins at the reset address when RESET is de-asserted. On reset, the interrupt signals are masked and the Flags register is cleared (Set to \$00).

3.3 Interrupts

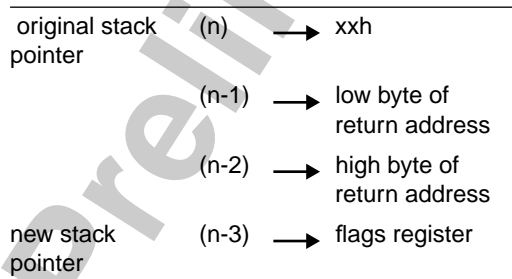
There are two maskable interrupt lines into the CPU.

**Interrupt**

The standard interrupt is enabled by the SETI instruction and disabled by the CLRI instruction.

More than one interrupt source may be allowed by ORing the interrupt sources together and providing an I/O port to determine where the interrupt came from.

If interrupts are enabled and become active then the CPU will stop at the next instruction boundary. The address of the instruction which would have been executed and the flags register are pushed onto the stack. The stack will now look like this:



The CPU program counter is now loaded with FFF0h and execution from this address

now takes place. This may be an aliased address as explained in the Address Space section. At this location there would normally be a jump instruction to the actual interrupt routine.

After the interrupt has been serviced an IRET instruction should be issued. This will pull the stack contents off the stack and increment its value back to (n) as in the previous value.

**Debug interrupt**

The DIRQEXT pin can be used as a normal interrupt of higher priority than IRQ. In addition (or instead of IRQ) a debug control block can be added to DIRQ.

This provides the capability to trap on a program address or read/write of a memory data address and/or data with or without a bit mask applied to the data.

The debug vector will go to \$FFF8 and execute from there: this will normally be a jump to a debug routine.

The debug interrupt is of higher priority than IRQ. It is possible for the debug interrupt to interrupt the IRQ interrupt routine. The IRQ interrupt will not interrupt the debug routine unless the SETI instruction is issued in the debug routine.

3.4 Flags register

The Flags register is a byte wide. It is moved on and off the stack by interrupt calls or specific instructions. Its form is as follows:

Flags

Bit	Mnemonic	Flag
0	ZF	Zero flag
1	CF	Carry flag
2	OV	Overflow flag
3	SF	Sign flag
4	IF	Interrupt flag
5	HF	Halt flag
6	DF	Debug Interrupt flag
7		reserved

## 4 Instruction set

The E-MCU-8A instruction set is summarised in the following set of tables. For each form of each instruction the mnemonic, opcode, length in bytes and syntax are given.

Please note the following:

### Opcodes

These are shown in hexadecimal.

### Length in bytes

The number of bytes of code required for the instruction in the form shown. Note that

some variants of instructions require more bytes than others.

### Example syntax

The symbol ^ after a variable implies indirection.

The symbol ^ before a variable implies a constant with the value of the variable address.

*Var1*, *Var2*, *Var3* are byte variables.

*Const* is a byte constant.

### Arithmetic instructions

Instruction	Mnemonic	Opcode	Length in bytes	Example
Add	ADD	\$02	3	ADD <i>Var1</i> <i>Const</i> ;
	ADD	\$62	3	ADD <i>Var1</i> ^ <i>Const</i> ;
	ADD	\$22	4	<i>Var2</i> = <i>Var1</i> + <i>Const</i> ;
	ADD	\$82	4	<i>Var2</i> = <i>Var1</i> ^ + <i>Const</i> ;
	ADD	\$A2	4	<i>Var2</i> ^ = <i>Var1</i> + <i>Const</i> ;
	ADD	\$42	4	<i>Var3</i> = <i>Var1</i> + <i>Var2</i> ;
Add with carry	ADC	\$03	3	ADC <i>Var1</i> <i>Const</i> ;
	ADC	\$63	3	ADC <i>Var1</i> ^ <i>Const</i> ;
	ADC	\$23	4	<i>Var2</i> = <i>Var1</i> ADC <i>Const</i> ;
	ADC	\$83	4	<i>Var2</i> = <i>Var1</i> ^ ADC <i>Const</i> ;
	ADC	\$A3	4	<i>Var2</i> ^ = <i>Var1</i> ADC <i>Const</i> ;
	ADC	\$43	4	<i>Var3</i> = <i>Var1</i> ADC <i>Var2</i> ;
Subtract	SUB	\$32	4	<i>Var2</i> = <i>Var1</i> - <i>Const</i> ;
	SUB	\$12	3	SUB <i>Var1</i> <i>Const</i> ;
	SUB	\$52	4	<i>Var3</i> = <i>Var1</i> - <i>Var2</i> ;
	SUB	\$72	3	SUB <i>Var1</i> ^ <i>Const</i> ;
	SUB	\$92	4	<i>Var2</i> = <i>Var1</i> ^ - <i>Const</i> ;
	SUB	\$B2	4	<i>Var2</i> ^ = <i>Var1</i> - <i>Const</i> ;
Subtract with carry	SBC	\$13	3	SBC <i>Var1</i> <i>Const</i> ;
	SBC	\$33	4	<i>Var2</i> = <i>Var1</i> SBC <i>Const</i> ;
	SBC	\$53	4	<i>Var3</i> = <i>Var1</i> SBC <i>Var2</i> ;
	SBC	\$73	3	SBC <i>Var1</i> ^ <i>Const</i> ;
	SBC	\$93	4	<i>Var2</i> = <i>Var1</i> ^ SBC <i>Const</i> ;
	SBC	\$B3	4	<i>Var2</i> ^ = <i>Var1</i> SBC <i>Const</i> ;
Increment by 1	INC	\$41	3	<i>Var2</i> = INC <i>Var1</i> ;
	INC	\$61	3	<i>Var2</i> = INC <i>Var1</i> ^;

## Arithmetic instructions

Instruction	Mnemonic	Opcode	Length in bytes	Example
Decrement by 1	INC	\$81	3	<i>Var2^ = INC Var1;</i>
	INC	\$A1	3	<i>Var2^ = INC Var1^;</i>
	INC	\$01	2	<i>INC Var1;</i>
	INC	\$21	2	<i>INC Var1^;</i>
	DEC	\$51	3	<i>Var2 = DEC Var1;</i>
	DEC	\$71	3	<i>Var2 = DEC Var1^;</i>
	DEC	\$91	3	<i>Var2^ = DEC Var1;</i>
	DEC	\$B1	3	<i>Var2^ = DEC Var1^;</i>
	DEC	\$11	2	<i>DEC Var1;</i>
	DEC	\$31	2	<i>DEC Var1^;</i>

## Logical instructions

Instruction	Mnemonic	Opcode	Length in bytes	Example
Boolean AND	AND	\$14	4	<i>Var2 = Var1 AND Const;</i>
	AND	\$44	4	<i>Var2 = Var1^ AND Const;</i>
	AND	\$54	4	<i>Var2^ = Var1 AND Const;</i>
	AND	\$24	4	<i>Var3 = Var1 AND Var2;</i>
	AND	\$04	3	<i>AND Var1 Const;</i>
	AND	\$34	3	<i>AND Var1^ Const;</i>
Boolean OR	OR	\$15	4	<i>Var2 = Var1 OR Const;</i>
	OR	\$45	4	<i>Var2 = Var1^ OR Const;</i>
	OR	\$55	4	<i>Var2^ = Var1 OR Const;</i>
	OR	\$25	4	<i>Var3 = Var1 OR Var2;</i>
	OR	\$05	3	<i>OR Var1 Const;</i>
	OR	\$35	3	<i>OR Var1^ Const;</i>
Exclusive OR	XOR	\$16	4	<i>Var2 = Var1 XOR Const;</i>
	XOR	\$46	4	<i>Var2 = Var1^ XOR Const;</i>
	XOR	\$56	4	<i>Var2^ = Var1 XOR Const;</i>
	XOR	\$26	4	<i>Var3 = Var1 XOR Var2;</i>
	XOR	\$06	3	<i>XOR Var1 Const;</i>
	XOR	\$36	3	<i>XOR Var1^ Const;</i>

## Control instructions

Instruction	Mnemonic	Opcode	Length	Example
Compare	CMP	\$64	3	<i>CMP Var1 Const;</i>

## Control instructions

Instruction	Mnemonic	Opcode	Length	Example
	CMP	\$65	3	CMP Var1 Var2;
	CMP	\$66	3	CMP Var1^ Const;
	CMP	\$67	3	CMP Var1^ Var2;
	CMP	\$68	3	CMP Var1^ Var2^;
Unconditional jump	JP	\$FC		JP Var1 Var2;
Conditional jump: on zero	JZ	\$F0	3	JZ Var1 Var2;
on not zero	JNZ	\$F1	3	JNZ Var1 Var2;
on carry	JC	\$F2	3	JC Var1 Var2;
on not carry	JNC	\$F3	3	JNC Var1 Var2;
on overflow	JO	\$F4	3	JO Var1 Var2;
on not overflow	JNO	\$F5	3	JNO Var1 Var2;
on sign	JS	\$F6	3	JS Var1 Var2;
on not sign	JNS	\$F7	3	JNS Var1 Var2;
greater than	JG	\$F8	3	JG Var1 Var2;
greater than or equal to	JGE	\$F9	3	JGE Var1 Var2;
less than	JL	\$FA	3	JL Var1 Var2;
less than or equal to	JLE	\$FB	3	JLE Var1 Var2;
Unconditional jump to sub-routine	JSR	\$DC	3	JSR Var1 Var2;
Conditional jump to sub- routine: on carry	JSRC	\$D2	3	JSRC Var1 Var2;
greater than	JSRG	\$D8	3	JSRG Var1 Var2;
greater than or equal to	JSRGE	\$D9	3	JSRGE Var1 Var2;
less than	JSRL	\$DA	3	JSRL Var1 Var2;
less than or equal to	JSRLE	\$DB	3	JSRLE Var1 Var2;
on not carry	JSRNC	\$D3	3	JSRNC Var1 Var2;
on not overflow	JSRNO	\$D5	3	JSRNO Var1 Var2;
on not sign	JSRNS	\$D7	3	JSRNS Var1 Var2;
on not zero	JSRNZ	\$D1	3	JSRNZ Var1 Var2;
on overflow	JSRO	\$D4	3	JSRO Var1 Var2;
on sign	JSRS	\$D6	3	JSRS Var1 Var2;
on zero	JSRZ	\$D0	3	JSRZ Var1 Var2;
Jump indirect	JPI	\$EC	3	JPI Var1^ Var2^;
Return from sub-routine	RTS	\$CC	1	RTS;
Bit test	BTST	\$74	3	BTST Var1 Const;
	BTST	\$84	3	BTST Var1^ Const;

## Stack and miscellaneous instructions

Instruction	Mnemonic	Opcode	Length	Example
Increment stack pointer	INCSP	\$E3	1	INCSP;
Decrement stack pointer	DECSP	\$E4	1	DECSP;
Pop flags register	POPF	\$E5	1	POPF;
Push flags register	PUSHF	\$E6	1	PUSHF;
Push variable	PUSHV	\$DD	2	PUSHV <i>Var1</i> ;
Push constant	PUSHC	\$DE	2	PUSHC <i>Const</i> ;
Pop variable	POPV	\$DF	2	POPV <i>Var1</i> ;
No operation	NOP	\$FF	1	NOP;
Translate	TXL	\$E7	4	<i>Var3</i> = TXL <i>Var1</i> <i>Var2</i> ;
Write code area	WRCODE	\$EA	4	<i>Var1</i> <i>Var2</i> = WRCODE <i>Var</i> ;

## Shift and rotate instructions

Instruction	Mnemonic	Opcode	Length	Example
Invert	INV	\$07	2	INV <i>Var1</i> ;
	INV	\$17	2	INV <i>Var1</i> <sup>^</sup> ;
	INV	\$27	3	<i>Var2</i> = INV <i>Var1</i> ;
	INV	\$37	3	<i>Var2</i> = INV <i>Var1</i> <sup>^</sup> ;
	INV	\$47	3	<i>Var2</i> <sup>^</sup> = INV <i>Var1</i> ;
	INV	\$57	3	<i>Var2</i> <sup>^</sup> = INV <i>Var1</i> <sup>^</sup> ;
Shift right	SHR	\$08	2	SHR <i>Var1</i> ;
	SHR	\$18	2	SHR <i>Var1</i> <sup>^</sup> ;
	SHR	\$28	3	<i>Var2</i> = SHR <i>Var1</i> ;
	SHR	\$38	3	<i>Var2</i> = SHR <i>Var1</i> <sup>^</sup> ;
	SHR	\$48	3	<i>Var2</i> <sup>^</sup> = SHR <i>Var1</i> ;
	SHR	\$58	3	<i>Var2</i> <sup>^</sup> = SHR <i>Var1</i> <sup>^</sup> ;
Rotate right	ROR	\$09	2	ROR <i>Var1</i> ;
	ROR	\$19	2	ROR <i>Var1</i> <sup>^</sup> ;
	ROR	\$29	3	<i>Var2</i> = ROR <i>Var1</i> ;
	ROR	\$39	3	<i>Var2</i> = ROR <i>Var1</i> <sup>^</sup> ;
	ROR	\$49	3	<i>Var2</i> <sup>^</sup> = ROR <i>Var1</i> ;
	ROR	\$59	3	<i>Var2</i> <sup>^</sup> = ROR <i>Var1</i> <sup>^</sup> ;
Shift arithmetic right	SAR	\$0A	2	SAR <i>Var1</i> ;
	SAR	\$1A	2	SAR <i>Var1</i> <sup>^</sup> ;
	SAR	\$2A	3	<i>Var2</i> = SAR <i>Var1</i> ;
	SAR	\$3A	3	<i>Var2</i> = SAR <i>Var1</i> <sup>^</sup> ;
	SAR	\$4A	3	<i>Var2</i> <sup>^</sup> = SAR <i>Var1</i> ;
	SAR	\$5A	3	<i>Var2</i> <sup>^</sup> = SAR <i>Var1</i> <sup>^</sup> ;

## Shift and rotate instructions

Instruction	Mnemonic	Opcode	Length	Example
Rotate right through carry	RORC	\$0B	2	<i>RORC Var1;</i>
	RORC	\$1B	2	<i>RORC Var1^;</i>
	RORC	\$2B	3	<i>Var2 = RORC Var1;</i>
	RORC	\$3B	3	<i>Var2 = RORC Var1^;</i>
	RORC	\$4B	3	<i>Var2^ = RORC Var1;</i>
	RORC	\$5B	3	<i>Var2^ = RORC Var1^;</i>
Rotate left	ROL	\$0D	2	<i>ROL Var1;</i>
	ROL	\$1D	2	<i>ROL Var1^;</i>
	ROL	\$2D	3	<i>Var2 = ROL Var1;</i>
	ROL	\$3D	3	<i>Var2 = ROL Var1^;</i>
	ROL	\$4D	3	<i>Var2^ = ROL Var1;</i>
	ROL	\$5D	3	<i>Var2^ = ROL Var1^;</i>
Shift arithmetic left	SAL	\$0E	2	<i>SAL Var1;</i>
	SAL	\$1E	2	<i>SAL Var1^;</i>
	SAL	\$2E	3	<i>Var2 = SAL Var1;</i>
	SAL	\$3E	3	<i>Var2 = SAL Var1^;</i>
	SAL	\$4E	3	<i>Var2^ = SAL Var1;</i>
	SAL	\$5E	3	<i>Var2^ = SAL Var1^;</i>
Rotate left through carry	ROLC	\$0F	2	<i>ROLC Var1;</i>
	ROLC	\$1F	2	<i>ROLC Var1^;</i>
	ROLC	\$2F	3	<i>Var2 = ROLC Var1;</i>
	ROLC	\$3F	3	<i>Var2 = ROLC Var1^;</i>
	ROLC	\$4F	3	<i>Var2^ = ROLC Var1;</i>
	ROLC	\$5F	3	<i>Var2^ = ROLC Var1^;</i>

## Load instructions

	Mnemonic	Opcode	Length	Example
Load	LD	\$00	2	<i>Var1 = STKTR;</i>
	LD	\$20	3	<i>STKPTR = Const;</i>
	LD	\$40	3	<i>Var2 = Var1;</i>
	LD	\$60	3	<i>Var2 = Var1^;</i>
	LD	\$80	3	<i>Var2^ = Var1;</i>
	LD	\$A0	3	<i>Var2^ = Var1^;</i>
	LD	\$C0	3	<i>Var1 = Const;</i>
	LD	\$E0	3	<i>Var1^ = Const;</i>

## Flag instructions

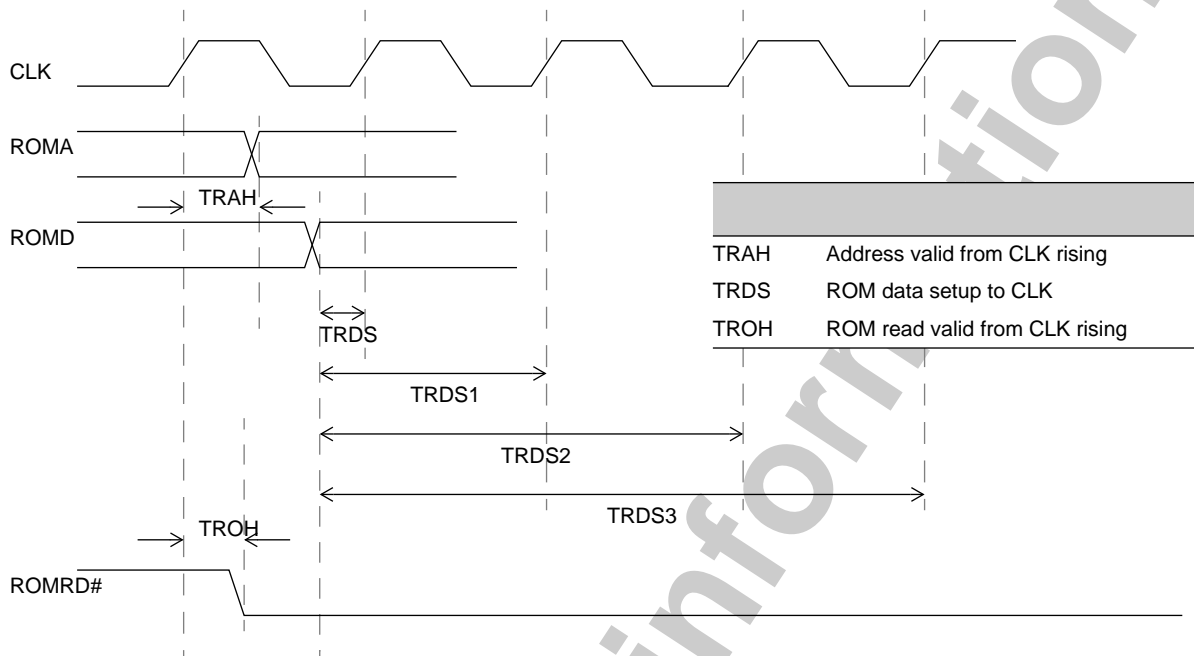
Instruction	Mnemonic	Opcode	Length	Example
Set Zero flag	SETZ	\$B4	1	SETZ;
Set Carry flag	SETC	\$B5	1	SETC;
Set overflow flag	SETO	\$B6	1	SETO;
Set sign flag	SETS	\$B7	1	SETS;
Clear Zero flag	CLRZ	\$B8	1	CLRZ;
Clear Carry flag	CLRC	\$B9	1	CLRC;
Clear Overflow flag	CLRO	\$BA	1	CLRO;
Clear Sign flag	CLRS	\$BB	1	CLRS;

## Interrupts

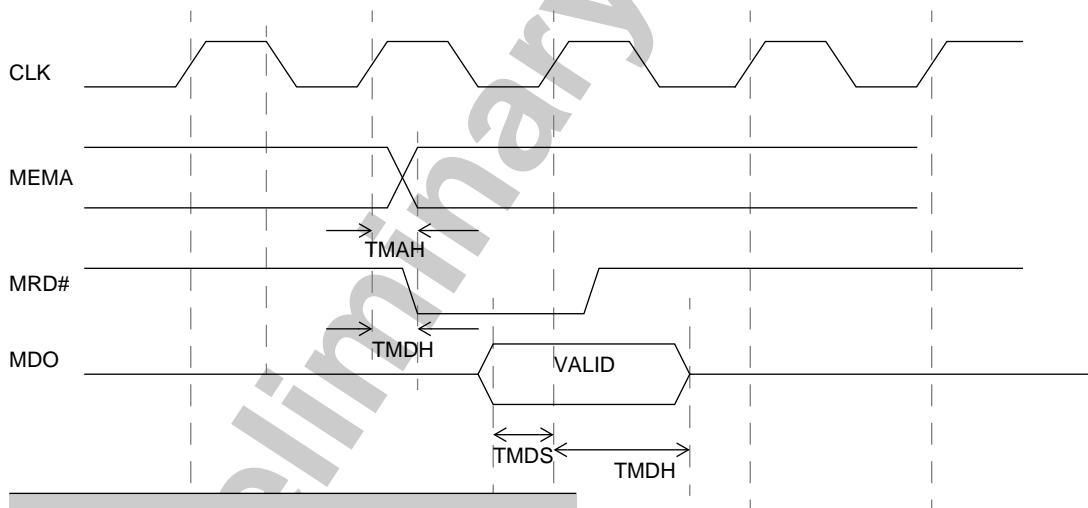
Instruction	Mnemonic	Opcode	Length	Example
Return from interrupt	IRET	\$FD	1	IRET;
Set interrupt flag	SETI	\$E1	1	SETI;
Clear interrupt flag	CLRI	\$E2	1	CLRI;
Halt execution	HALT	\$BC	1	HALT;
Set debug interrupt flag	SETD	\$E8	1	SETD;
Clear debug interrupt flag	CLRD	\$E9	1	CLRD;

## 5 Timing diagrams

### 5.1 Program ROM/RAM data reads of 0, 1, 2 and 3 wait states

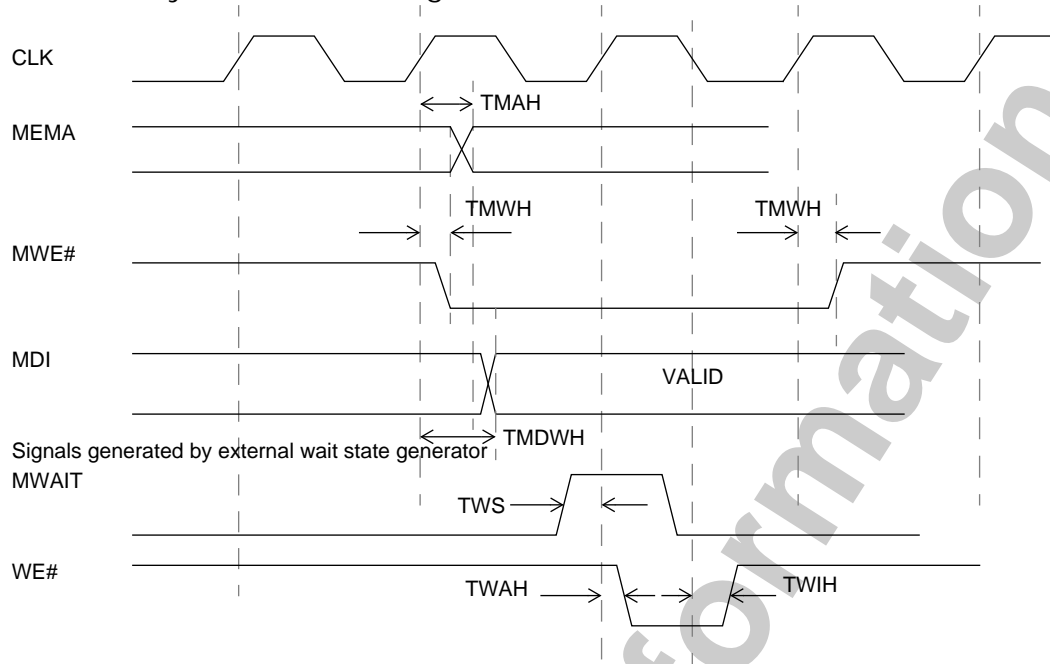


### 5.2 Data memory I/O reads



TMAH	Data address valid from rising edge of CLK
TMDH	Data read valid from CLK edge
TMDH	Read data setup to CLK rising
TMDH	Data read hold time from CLK rising

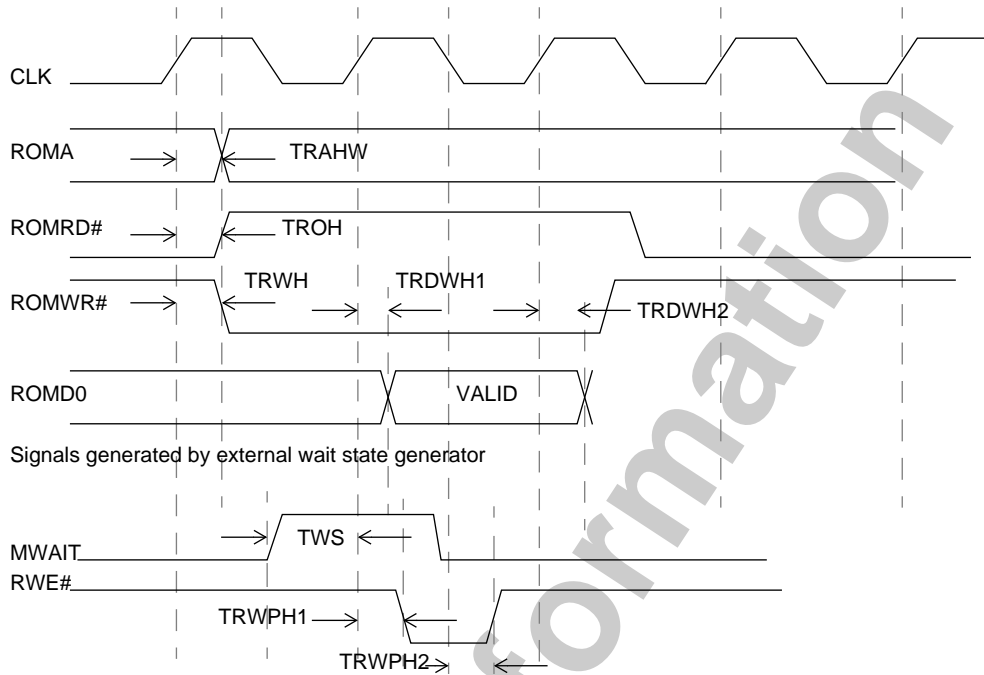
### 5.3 Data memory I/O write timing



Effect	
TMWH	Memory write valid from rising edge of CLK
TMDWH	Data out valid delay from CLK rising
TWS	MWAIT set up to CLK rising
TWAH	External data write hold time from CLK rising
TWIH	External data write hold time from CLK falling

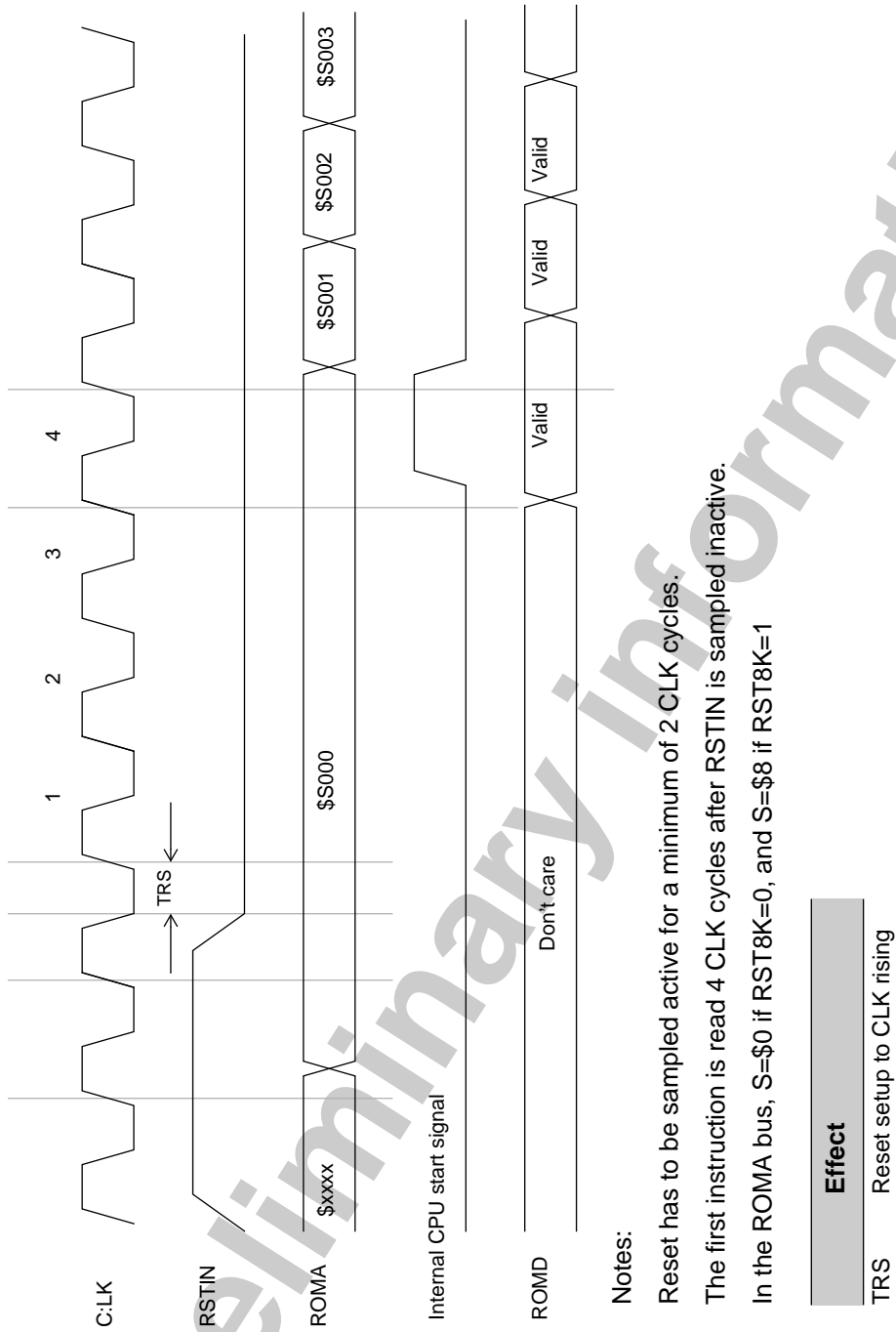
### 5.4 ROM write timings

(see next page)

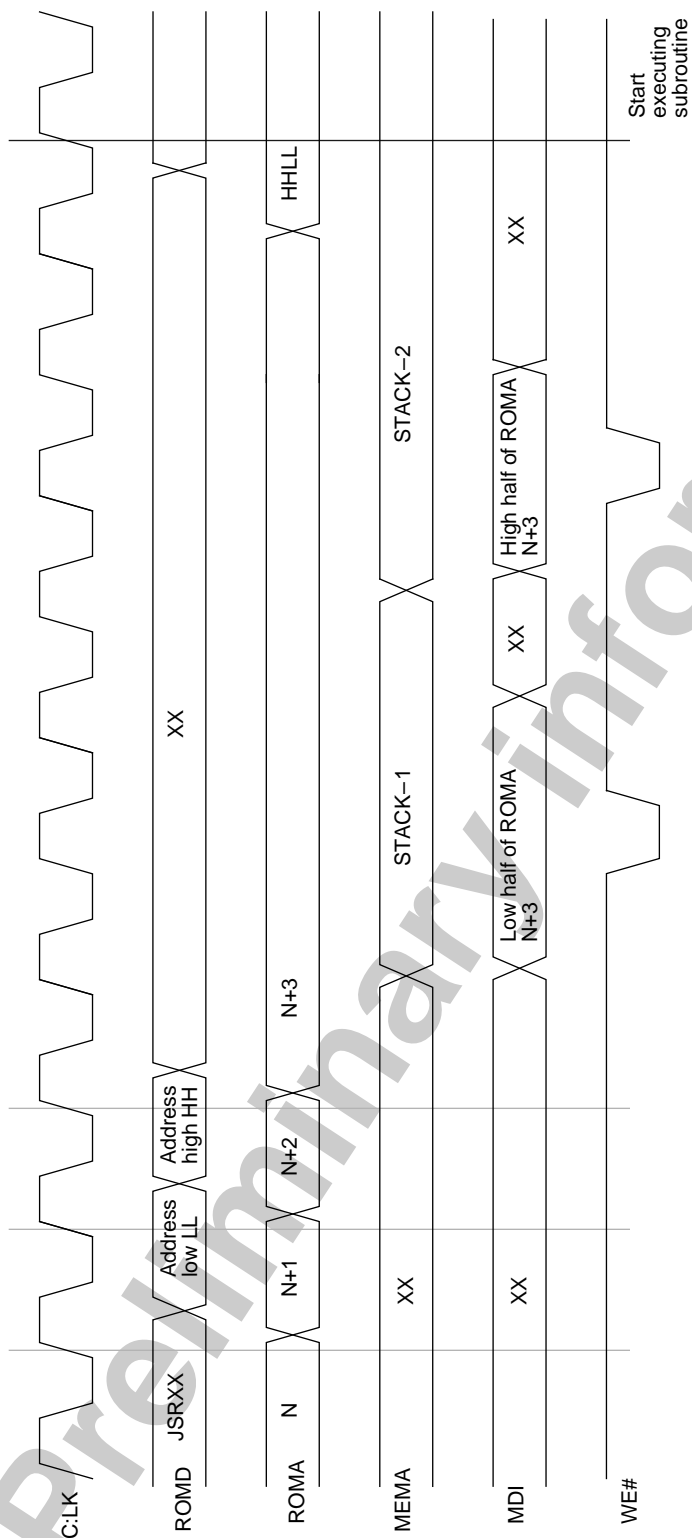


TRAHW	ROM address valid from CLK rising
TROH	ROM read valid from CLK rising
TRWH	ROM write valid from CLK rising
TRDWH1	ROM write data valid from CLK rising
TRDWH2	ROM write data hold from CLK rising
TWS	MWAIT setup to CLK rising
TRWPH1	External ROM write hold from CLK rising
TRWPH2	External ROM write hold from CLK falling.

5.5 Reset operation (zero wait states ROM read)



5.6 Subroutine call taken (Zero wait states ROM read)

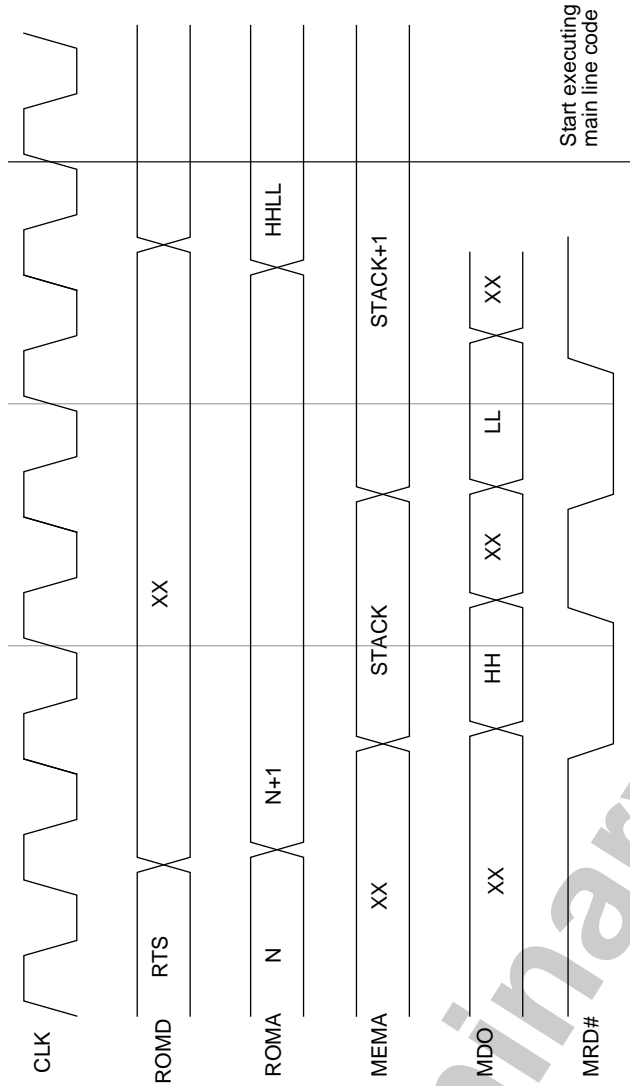


**Notes:**

If subroutine call is not taken (ie conditional call based on flags) then code execution continues with the instruction at ROM address N+3.

When the call is taken, the stack is decremented before its value is placed on the memory address bus. It will end with its original value minus 2.

5.7 Return from subroutine

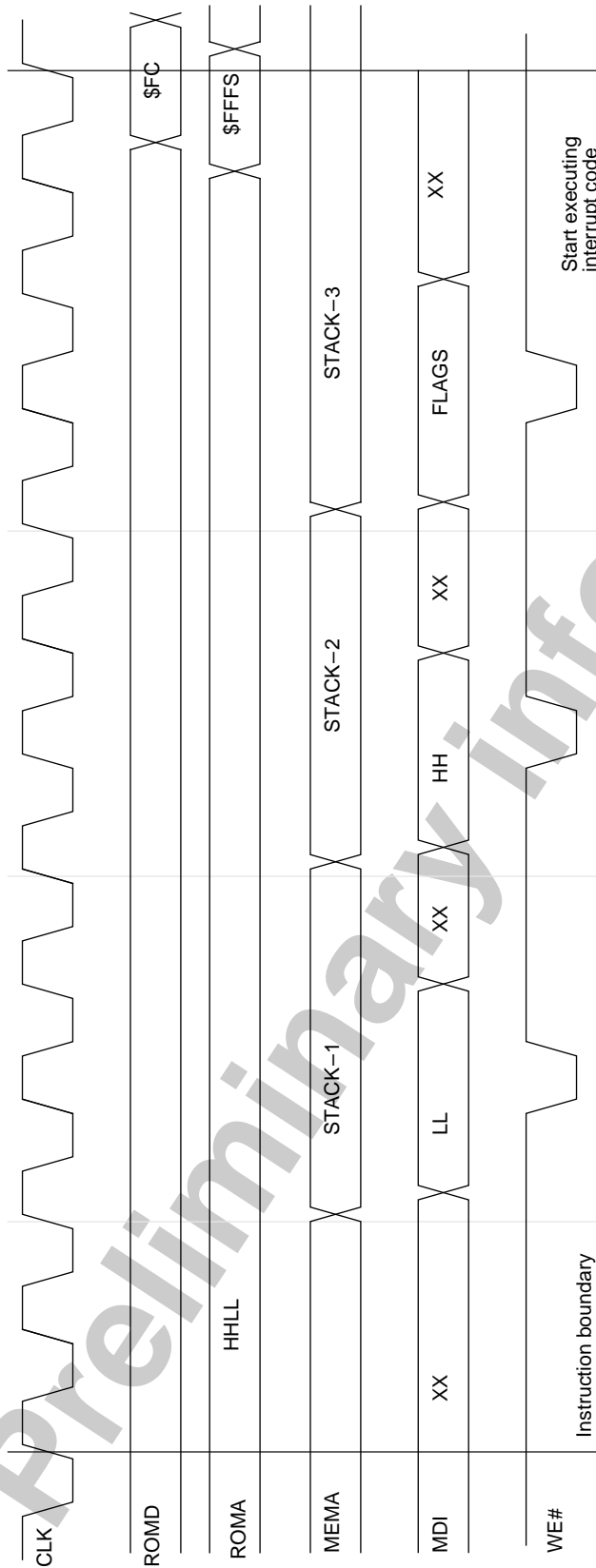


**Notes:**

When returning, the stack address is placed on the memory address bus and then incremented after the data at that location has been read. It will then end up with its original value +2.

Preliminary Information

5.8 Interrupt call



Notes:

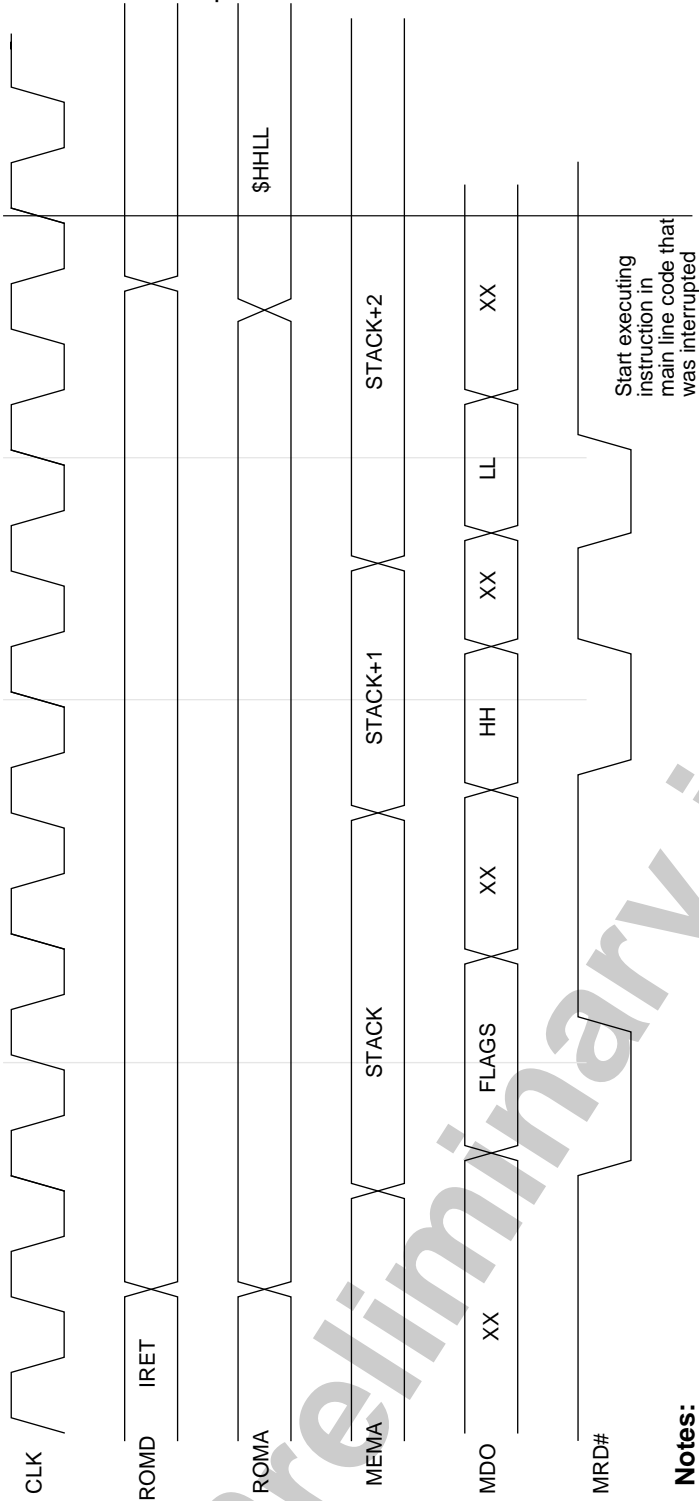
The stack is decremented before its value is placed in the memory address bus. It will end with its original value minus 3.

In the ROM address bus:

S=\$0 on IRQ active

S=\$8 on DIRQ/DIRQEXT active

### 5.9 Interrupt routine return



**Notes:**

The stack address is placed on to the memory address bus and then incremented after the data at that location has been read. It will end up with its original value +3.

*Preliminary information*