Technical Reference Manual

for

6502 Assembler

Printed Date: 99/05/26

Ver.98-01

Technical Reference Manual for 6502 Assembler

Contents

Part I 6502 Microprocessor

Chapter 1-1 Introduction

The 6502 microprocessor (6502 *u*P) can execute 56 different types of instructions (65C02 can execute 71 instructions). The various combinations of addressing that are available for use by individual instruction types give the microprocessor a total of 151 executable instructions (65C02 has 212).

Instructions are comprised of one, two, or three bytes. The first byte always holds the machine code equivalent of the operation code (**op code**), we summarize all instructions op code matrix in section 1-2-2. There is an instruction decode logic in 6502 that can decode the op-code and issue appropriate internal-control signals to all other elements of the microprocessor, and possibly to external circuits in the system. The second and third bytes, if has, are gated into the **data bus buffer**, from which they are routed into either the **Arithmetic Logic Unit** (ALU) if they represent data, or into the **program counter** (PC) if they represent an address.

1-1-1 Register Structures

Fig. 1-1-1 shows a register structures diagram of the 6502 microprocessor. There are 6 main registers: a 16-bit program counter (PC), an 8-bit accumulator (A or AC), two 8-bit index registers (X and Y), an 8-bit stack pointer register (S or SP), and an 8-bit status register (P).

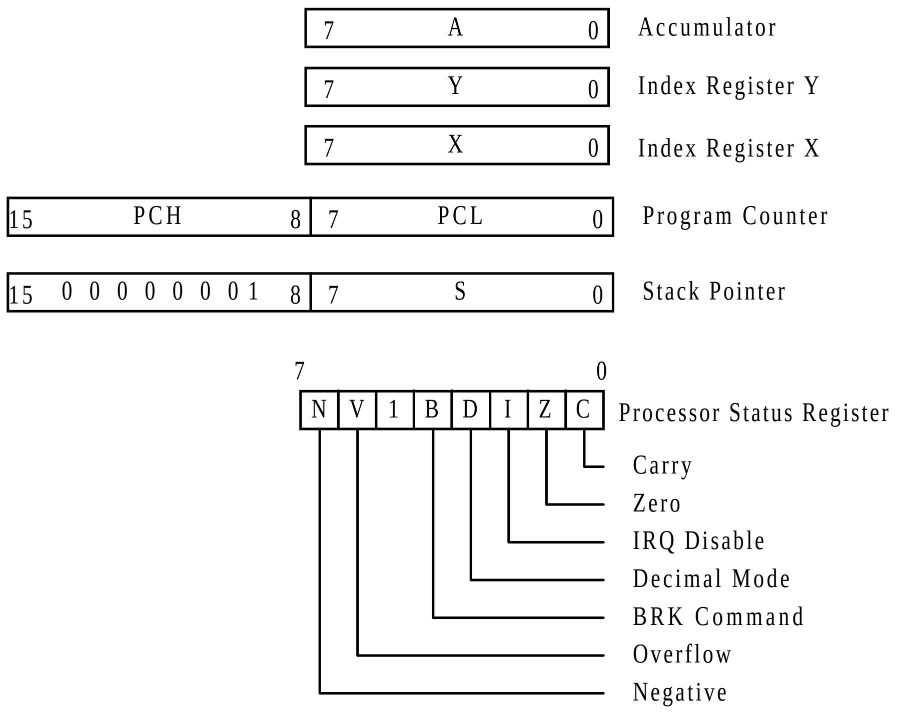


Fig. 1-1-1 Register structures of the 6502 microprocessor

General Purpose Registers

Three 8-bit registers of the six main registers, which are the accumulator and the X, Y registers, can be used to save temporary data values, to communicate with memory, to maintain counters, or for a variety of other applications. The three registers are called **general-purpose registers**.

The accumulator is the only register in which arithmetic and logical operations can be performed, and it holds one of the operands for each add, subtract, AND, OR, and Exclusive-OR instruction. The 6502 also has instructions to logically shift the contents of the accumulator to the right or to the left.

The X and Y registers are primarily employed as index registers to access sequential values in memory, but the fact that they can be incremented and decremented under program control also makes them popular as general purpose counters.

Status Register

The **processor status register** (P), see Fig. 1-1-2, contains seven usable bits. Five of these bits are "status flags"; they provide information on the result of a previously executed instruction (in most cases, the preceding instruction). The two other bits are "control bits".

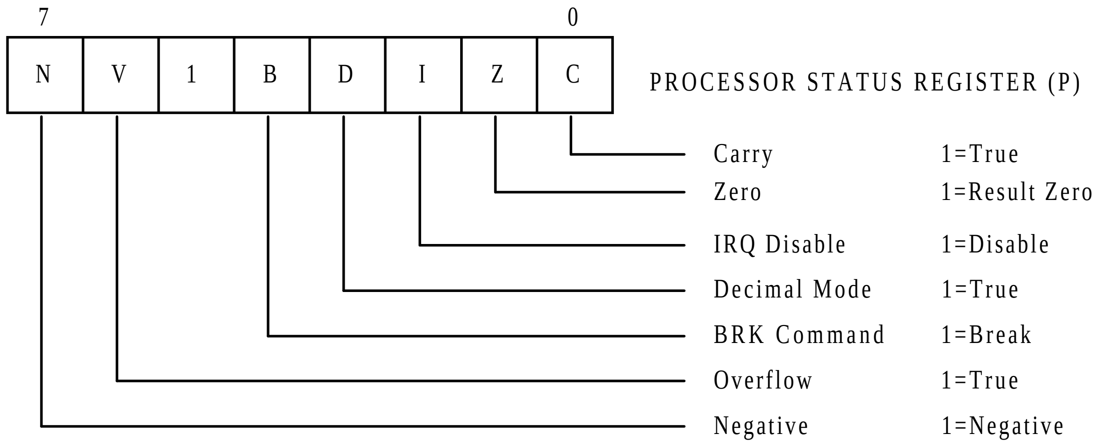


Fig. 1-1-2 Status Register of the 6502

Carry flag (C)	is used to save any carry produced by an add operation, any borrow produced by a subtract operation, or the value of a bit after a shift operation. The carry also reflects the result of a compare operation.
Zero flag (Z)	represents whether or not the result of an operation is zero.
Break flag (B)	indicates whether an interrupt request to the 6502 was caused by a "break" instruction or by an externally generated interrupt.
Overflow flag (V)	is applicable only to arithmetic operations on signed numbers. It is set if the addition of two like-signed numbers or the subtraction of two unlike-signed numbers produces a result greater than $+127_{10}$ or less than -128_{10}.
Negative flag (N)	represents whether or not the result of a signed arithmetic operation produced a negative result. This bit is also used as a general-purpose indicator of the state of the most-significant bit position in the accumulator.

The 6502 has several instructions that can be used to check the condition, or state, of each of these flags. According to the results of these checks, the 6502 can decide whether or not to execute one of two possible sequences of instructions. All flags remain set or cleared after these “check” operations are performed. Therefore, not all 6502 instructions affect the flags.

About the control bits - the **IRQ Disable bit** (I) and the **Decimal Mode bit** (D):

IRQ Disable bit (I)	is used to “lock out” external interrupts to the 6502 at times when, for some reason, the microprocessor is not prepared to serve an interrupt. The 6502 automatically disables interrupts while it is being reset or when it is serving a previous interrupt. Programs can also disable interrupts during periods when a certain sequence of instructions must be executed uninterrupted.
Decimal Mode bit (D)	controls whether the internal ALU of the 6502 is to operate as a straight binary adder or as a decimal adder. In binary mode, the ALU treats arithmetic operands as 8-bit binary numbers. In decimal mode, the ALU treats arithmetic operands as two BCD (Binary-Coded Decimal) digits packed into one 8-bit byte. If the Decimal Mode bit set to “1” then the ALU operates in BCD type, and the results will be equivalent to them got from normal decimal system.

Stack Pointer Register

The 6502 microprocessor can be programmed so that at some point in a program, program execution can be transferred to another sequence of instructions that is stored in another part of memory. Before this transfer actually occurs, the 6502 saves the address of the next instruction in its current sequence of instructions. And, after the new sequence of instructions has been executed, the program control would be returned to the point that is just after the instruction that caused transfer operation. The return address is saved in an area of memory called a **stack**.

The stack is also a portion of memory that is designated to accept these return addresses. However, the stack must be implemented in Page 1 of the 6502 address space – the addresses from 0100 to 01FF (hexadecimals). Since this design restriction and the address register for the stack (**the Stack Pointer**) is only 8 bits wide that ensures the high-order two digits of the address are always 01.

Data are entered onto, and extracted from, the stack of the 6502, in memory, the same way that we stack papers on the desk. The last item to be placed on the stack is also the first item to be removed from it. This type of stack is usually referred to as “last in, first out”. As return addresses are entered onto the stack, they are really stored in R/W memory at lower and lower memory addresses; the stack “builds” toward address 0. The Stack Pointer, therefore, is automatically decremented by 1 as each new address byte is pushed onto the stack, and is automatically incremented by 1 as each address byte is pulled off of the stack. The A and P registers can also be saved on the stack, if desired.

Program Counter

The last (sixth) register described is the **program counter** that is a 16-bit counter that determines which memory location will be accessed next. It is automatically incremented after each memory access so that it addresses the next consecutive memory location. Since the program counter is 16 bits wide, it can address any location in the 64K-byte address space of the 6502.

1-1-2 Reset and Interrupt Signals

The 6502 has three separate signals, $\overline{\text{RES}}$, $\overline{\text{IRQ}}$, and $\overline{\text{NMI}}$, by which external devices can cause an executing program to be interrupted.

① **Reset** ($\overline{\text{RES}}$) is used to initialize the 6502 to a known state, or to start the 6502 when power is first applied to the system. While $\overline{\text{RES}}$ is active, the 6502 can neither transmit nor receive information. When $\overline{\text{RES}}$ goes high level, the microprocessor loads the program counter with the contents of memory locations FFFC and FFFD (hexadecimal), the address from which the first (or initial) instruction will be fetched.

② **Interrupt ReQuest** ($\overline{\text{IRQ}}$) is an input signal by which most peripheral devices request service from the 6502. Here, “request” is the keyword here. Unlike the Reset signal, which interrupts the 6502 unconditionally $\overline{\text{IRQ}}$ simply informs the microprocessor that some peripheral device in the system is waiting to send or receive information. An interrupt request will be acknowledged only if the Interrupt Disable bit (I) of the processor status register is reset to a logic zero. If “I” is reset, the 6502 will load the contents of the two uppermost memory locations, FFFE and FFFF (hexadecimal), into the program counter. If “I” is set when the interrupt request is received, the microprocessor ignores the request, and continues executing as if no request has been made. The 6502 does not “remember” the request, but when the “I” bit is reset, the 6502 is interrupted.

③ **Non-Maskable Interrupt** ($\overline{\text{NMI}}$) is an interrupt that cannot be disabled. The $\overline{\text{IRQ}}$ input signal is *maskable*; that is, it can be enabled or disabled, depending on the state of the “I” bit in the processor status register. Non-maskable interrupt, $\overline{\text{NMI}}$, does not merely request to interrupt the microprocessor, like $\overline{\text{RES}}$, it **does** interrupt the microprocessor each time it is activated. The $\overline{\text{NMI}}$ line is designed to interrupt the 6502 under some condition that requires immediate attention, such as a power failure. The address of the sequence of instructions that serve the $\overline{\text{NMI}}$ interrupt is stored in two consecutive memory locations, FFFA and FFFB (hexadecimal). Fig.1-1-3 shows the Reset and Interrupt vectors locations of the 6502.

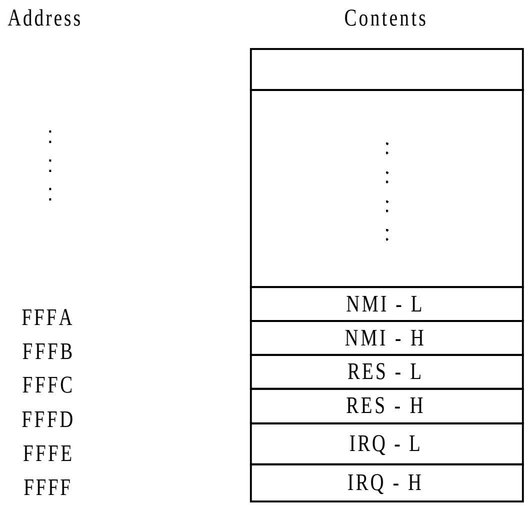


Fig. 1-1-3 Vectors location of the 6502

Chapter 1-2 6502 Instruction Set

The 6502 has 56 different instructions and 13 addressing modes, making it popular to be used. The chapter will introduce the 13 addressing modes and 56 instructions and lead the reader to write simple program.

1-2-1 Instruction Formats

All programs in this Part are given in the assembler format. This format divides each line in the program (i.e., each line of program *code*) into four fields: **label**, **op code**, **operand** and **comment**.

format:	**Label**	**Op code**	**Operand**	**Comment**
	KEYIN3	LDY	#00	；Used to fetch element count.

① **Label field** is used to mark a symbolic name or label to the location of an instruction, so that it can be referenced by other instructions in the program. The label field is an optional field; In fact, most instructions will not be labeled. However, if an instruction is labeled, it should follow the restrictions: ❶The label must begin in the leftmost column of the line (column 1); ❷The label must begin with an alphabetic character (A through Z); ❸The label must be no longer than six characters; ❹And the labels must be different. For example, the instruction JUMP KEYIN3 will cause the program counter to be unconditionally loaded with the memory address that has been assigned the label KEYIN3. The instruction at label KEYIN3 will be the next instruction to be executed after the JUMP (JMP) instruction is executed.

② **Op code field** is mandatory for every line in the program that contains an instruction, and must contain one of the 56 valid mnemonics of the 6502 (see 1-2-2). The op code may begin in any column except column 1, and must be separated from a label (if present) by at least one space.

③ **Operand field** is used to specify data or an address for instructions that require an operand. The operand must be separated from the op code by at least one space. The assembler will accept operands in any of five forms. Applying an appropriate prefix character to the operand, as followings to specify the five forms:

Prefix	**Operand Form**
None	Base 10 (Decimal)
$	Base 16 (Hexadecimal)
@	Base 8 (Octal)
%	Base 2 (Binary)
'	ASCII

④ **Comment field** is always optional, and is used to add an explanatory note to a statement. The contents of the comment field are not executed, so you can write what kind of comment you want. However, the text of the comment should be preceded by a semicolon (;). Comments may be used alone, too, without being appended to a line that contains an instruction.

1-2-2 Instructions Op Code Matrix

We summarize all the instructions and op codes here, for user to refer conveniently.

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BRK 7,1	ORA ind,X 6,2				ORA zpg 3,2	ASL zpg 5,2		PHP 3,1	ORA imm 2,2	ASL A 2,1			ORA abs 4,3	ASL abs 6,3	
1	BPL rel 2,2 ♠	ORA ind,Y 5,2 ※				ORA zpg,X 4,2	ASL zpg,X 6,2		CLC 2,1	ORA abs,Y 4,3 ※				ORA abs,X 4,3 ※	ASL abs,X 6,3	
2	JSR abs 6,3	AND ind,X 6,2			BIT zpg 3,2	AND zpg 3,2	ROL zpg 5,2		PLP 4,1	AND imm 2,2	ROL A 2,1		BIT abs 4,3	AND abs 4,3	ROL abs 6,3	
3	BMI rel 2,2 ♠	AND ind,Y 5,2 ※				AND zpg,X 4,2	ROL zpg,X 6,2		SEC 2,1	AND abs,Y 4,3 ※				AND abs,X 4,3 ※	ROL abs,X 6,3	
4	RTI 6,1	EOR ind,X 6,2				EOR zpg 3,2	LSR zpg 5,2		PHA 3,1	EOR imm 2,2	LSR A 2,1		JMP abs 3,3	EOR abs 4,3	LSR abs 6,3	
5	BVC rel 2,2 ♠	EOR ind,Y 5,2 ※				EOR zpg,X 4,2	LSR zpg,X 6,2		CLI 2,1	EOR abs,Y 4,3 ※				EOR abs,X 4,3 ※	LSR abs,X 6,3	
6	RTS 6,1	ADC ind,X 6,2				ADC zpg 3,2 ↑	ROR zpg 5,2		PLA 4,1	ADC imm 2,2 ↑	ROR A 2,1		JMP ind 6,3	ADC abs 4,3 ↑	ROR abs 6,3	
7	BVS rel 2,2 ♠	ADC ind,Y 5,2 ※↑				ADC zpg,X 4,2 ↑	ROR zpg,X 6,2		SEI 2,1	ADC abs,Y 4,3 ※↑				ADC abs,X 4,3 ※↑	ROR abs,X 6,3	
8		STA ind,X 6,2			STY zpg 3,2	STA zpg 3,2	STX zpg 3,2		DEY 2,1		TXA 2,1		STY abs 4,3	STA abs 4,3	STX abs 4,3	
9	BCC rel 2,2 ♠	STA ind,Y 6,2			STY zpg,X 4,2	STA zpg,X 4,2	STX zpg,Y 4,2		TYA 2,1	STA abs,Y 5,3	TXS 2,1			STA abs,X 4,3		
A	LDY imm 2,2	LDA ind,X 6,2	LDX imm 2,2		LDY zpg 3,2	LDA zpg 3,2	LDX zpg 3,2		TAY 2,1	LDA imm 2,2	TAX 2,1		LDY abs 4,3	LDA abs 4,3	LDX abs 4,3	
B	BCS rel 2,2 ♠	LDA ind,Y 5,2 ※			LDY zpg,X 4,2	LDA zpg,X 4,2	LDX zpg,Y 4,2		CLV 2,1	LDA abs,Y 4,3 ※	TSX 2,1		LDY abs,X 4,3 ※	LDA abs,X 4,3 ※	LDX abs,Y 4,3 ※	
C	CPY imm 2,2	CMP ind,X 6,2			CPY zpg 3,2	CMP zpg 3,2	DEC zpg 5,2		INY 2,1	CMP imm 2,2	DEX 2,1		CPY abs 4,3	CMP abs 4,3	DEC abs 6,3	
D	BNE rel 2,2 ♠	CMP ind,Y 5,2 ※				CMP zpg,X 4,2	DEC zpg,X 6,2		CLD 2,1	CMP abs,Y 4,3 ※				CMP abs,X 4,3 ※	DEC abs,X 6,3	
E	CPX imm 2,2	SBC ind,X 6,2 ↑			CPX zpg 3,2	SBC zpg 3,2 ↑	INC zpg 5,2		INX 2,1	SBC imm 2,2 ↑	NOP 2,1		CPX abs 4,3	SBC abs 4,3 ↑	INC abs 6,3	
F	BEQ rel 2,2 ♠	SBC ind,Y 5,2 ※↑				SBC zpg,X 4,2 ↑	INC zpg,X 6,2		SED 2,1	SBC abs,Y 4,3 ※↑				SBC abs,X 4,3 ※↑	INC abs,X 6,3	

BRK imm
7, 1

7, 1 = Cycles, bytes
imm = immediate zpg = zero-page
rel = relative abs = absolue ind = indirect

↑ Add 1 to N if in decimal mode
※ Add 1 to N if page boundary is crossed
♠ Add 1 to N if branch occurs to same page,
Add 2 to N if branch occurs different page

1-2-3 Load and Store Instructions

Instruction	Codes	Cycles / Bytes	Status flags affected
LDA imm	A9xx	2/2	N， Z
LDA abs	ADaabb	4/3	N， Z
LDA zpg	A5aa	3/2	N， Z
LDA abs,x	BDaabb	4/3	N， Z
LDA abs,y	B9aabb	4/3	N， Z
LDA zpg,x	B5aa	4/2	N， Z
LDA (zpg), y	B1aa	5/2	N， Z
LDA (zpg,x)	A1aa	6/2	N， Z
LDX imm	A2xx	2/2	N， Z.
LDX abs	AEaabb	4/3	N， Z
LDX zpg	A6aa	3/2	N， Z
LDX abs,y	BEaabb	4/3	N， Z
LDX zpg, y	B6aa	4/2	N， Z
LDY imm	A0xx	2/2	N， Z
LDY abs	ACaabb	4/3	N， Z
LDY zpg	A4aa	3/2	N， Z
LDY abs,x	BCaabb	4/3	N， Z
LDY zpg,x	B4aa	4/2	N， Z

Moving data into and out of memory is the most basic operations of the 6502, due to a memory-oriented architecture. All such transfers are made via three registers – the Accumulator (A) register, the X register and the Y register. We call the process of transferring data from memory into one of the registers as "**load data**". There are three Load instructions:

Instruction	**Description**
LDA	Load Accumulator with memory
LDX	Load X register with memory
LDY	Load Y register with memory

There are two flags of the processor status register will be changed to provide some information about the value that has been loaded into the register, but the source location in memory is unaffected during the load operation. If the most significant bit (bit 7) of the loaded data is an 1 then the "negative flag (N)" of the processor status register will be set, and will be reset if bit 7 of the loaded value is a 0. Furthermore, the "zero flag (Z)" will be set if the loaded value is 0; otherwise it will be reset. Example:

```
LDA        $12C3
```

It will load the contents of memory location $12C3 into the accumulator. If the contents of memory location $12C3 is $33 then the contents of accumulator will be $33 after the instruction executed.

The process of transferring data from one of the registers of the 6502 into memory is called "store data". There are three Store instructions:

Instruction	**Description**
STA	Store Accumulator in Memory
STX	Store X Register in Memory
STY	Store Y Register in Memory

Instruction	Codes	Cycles / Bytes	Status flags affected
STA abs	8Daabb	4/3	
STA zpg	85aa	3/2	
STA abs,x	9Daabb	5/3	
STA abs,y	99aabb	5/3	
STA zpg,x	95aa	4/2	
STA (zpg), y	91aa	6/2	
STA (zpg,x)	81aa	6/2	
STX abs	8Eaabb	4/3	
STX zpg	86aa	3/2	
STX zpg, y	96aa	4/2	
STY abs	8Caabb	4/3	
STY zpg	84aa	3/2	
STY zpg,x	94aa	4/2	

The store instructions do not change the contents of the source registers (A, X, Y) and the processor status register. How can we do about storing a value in memory? Combining a "load immediate" and "store" instructions is the simplest way to store a value in memory, like:

```
LDA      #33
STA      $71
```

Which stores #33 in zero page memory location $0071.

1-2-4 Arithmetic Instructions

There are several instructions for adding and subtracting both binary-coded and binary-coded-decimal numbers in the 6502 microprocessor. All that instructions involve two operands, one in the accumulator and the other in memory or direct second byte value of the instruction (if immediate addressing mode is used) – refer to 1-2-12 "6502 Addressing Modes".

Representation of Numbers

The 6502 can add or subtract both unsigned and signed numbers. Actually, the forms make no difference to the microprocessor, but programmer should know how to interpret both forms.

In an **unsigned number**, each data bit carries a certain binary weight, according to its position within the number. Data bits are numbered from right to left, an 8-bit data with the rightmost bit labeled as b_0 and the leftmost bit labeled as b_7. The bit binary weight is correlative to the bit position that b_0 has a weight of 2^0 (decimal 1), b_1 has a weight of 2^1 (decimal 2), and b_7 has a weight of 2^7 (decimal 128) etc. Therefore, one byte can represent an unsigned number from decimal 0 (00000000) to decimal 255 (11111111). → (128 + 64 + 32 + 16 + 8 + 4 + 2 + 1=255)

In a **signed number**, the seven low-order bits (bit 0~6) represent data, and have the same weights as with unsigned numbers. The most significant bit (bit7) indicates the sign of the number. If the number is positive then the bit7 is logic 0; otherwise bit7 is logic 1. The range of the positive signed numbers is within decimal 0 (00000000) to decimal +127 (01111111). And, the negative signed numbers may be within the range of decimal –1 (11111111) to decimal –128 (10000000).

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	**bit position**
1	0	1	0	0	1	1	0	data
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	binary weight

1*128 + 0*64 + 1*32 + 0*16 + 0*8 + 1*4 + 1*2 + 0*1 = 166 equivalent decimal values(unsigned)
1(-) 0*64 + 1*32 + 0*16 + 0*8 + 1*4 + 1*2 + 0*1 = -38 equivalent decimal values(signed)

Someone might be bewildered and wonder why –1 is represented by binary 1111111, rather than by 10000001? Because we use "**two's complement**" form to represent their negative-signed numbers. And, why "two's complement"? The two's complement form was introduced to eliminate the problems that zero represented in two forms, 00000000 (the positive form) and 10000000 (negative form). In two's complement, zero is represented by only one form→00000000. To derive the negative two's complement form of a binary number; you just take the positive form of the number and reverse the sense of each bit. You change each 1 to a 0 and change each 0 to an 1, and **add 1** to the result. The following example shows the steps required in deriving the binary representation of –16 (in two's complement form).

$$\begin{array}{rl}
00010000 & +16_{10} \\
\downarrow & \\
11101111 & \text{"one's complement"} \\
+ \qquad\qquad 1 & \text{add 1} \\
\hline
11110000 & -16_{10}\text{(two's complement)}
\end{array}$$

Decimal Mode Instructions

Some instructions of the 6502 can cause ALU to operate as a binary adder or as a decimal adder during addition and subtraction. As operating as a binary adder, the ALU treats both 8-bit operands as binary numbers, with values from 00000000 to 11111111. When operating as a decimal adder, the ALU treats both operands as Binary-Coded Decimal (BCD) numbers, with two 4-bit BCD digits packed into each 8-bit operand.

We can use two instructions to choose the binary or decimal modes of the ALU. The two instructions are "**SED"** (Set Decimal Mode) and "**CLD"** (Clear Decimal Mode).

Instruction	Codes	Cycles / Bytes	Status flags affected
CLD	D8	2/1	D=0
SED	F8	2/1	D=1

The SED forces the ALU perform as a decimal (BCD) adder, and it will also set the Decimal Mode (D) control bit of the processor status register. The CLD instruction causes the ALU to perform as a binary adder, and clear the Decimal Mode bit (D) in the processor status register. The Decimal Mode bit (D) is undefined when power is applied; it must be either set or cleared by the initialization program.

Addition

Instruction	Codes	Cycles / Bytes	Status flags affected
ADC imm	69xx	2/2	N, V, Z, C
ADC abs	6Daabb	4/3	N, V, Z, C
ADC zpg	65aa	3/2	N, V, Z, C
ADC abs,x	7Daabb	4/3	N, V, Z, C
ADC abs,y	79aabb	4/3	N, V, Z, C
ADC zpg,x	75aa	4/2	N, V, Z, C
ADC (zpg),y	71aa	5/2	N, V, Z, C
ADC (zpg,x)	61aa	6/2	N, V, Z, C

Normally, there are two add operations we have ever encountered, one simply adds the operands and the other includes a carry in the addition. The former instruction, called "**ADD**" instruction, is used to add single byte numbers or to add the low-order bytes of two multi-byte operands. The latter instruction, call "**ADC** (Add with Carry)", is used to add the higher-order bytes of two multi-byte operands.

Since most additions involve multi-byte numbers, so the 6502 has only one add instruction—**ADC**. Symbolically, the instruction of ADC can be represented as:

$$A = A + M + C$$

Where,

A is the accumulator

M is the contents of memory (or an immediate value)

C is the Carry

That is, adds the operands to the accumulator with Carry flag. So, if the Carry flag has been set by some previous operation, then the addition becomes:

$$A = A + M + 1$$

If Carry is reset to a logic 0 when the ADC instruction is executed, then the addition becomes:

$$A = A + M + 0$$

For adding single-byte numbers or adding the least-significant byte of two multi-byte numbers, the Carry flag (C) must be cleared to logic 0 before performing the addition. The CLC (CLear Carry flag) instruction is used to clear Carry flag. Therefore, with the clear carry requirement the single-byte addition will look like:

```
CLC
ADC         $71
```

The contents of zero page memory location $0071 is added to the accumulator.

Instruction	Codes	Cycles / Bytes	Status flags affected
SEC	38	2/1	C=1
CLC	18	2/1	C=0

Sometimes, you don't need proceed an ADC instruction with a CLC instruction. If an immediate value is being added to the accumulator and you know the Carry flag has been set by some previous operation then you can use an immediate value, which is one less than the value you want to add, rather than using a CLC instruction before ADC instruction. For instance, to add 33 to the accumulator and the Carry is set to one, you can only code:

```
ADC        #32
```

The result will be same with coding CLC followed by ADC #33. In fact, it's not a good idea to do so.

Except absolute addressing (indexed or un-indexed, three bytes), refer to 1-2-12, the all ADC instructions are two bytes long. The ADC instructions will affect the following four flags in processor status register:

① Carry flag (C). If the sum of a binary addition exceeds decimal 255 (hex. FF) or if the sum of a binary-coded decimal addition exceeds decimal 99, then the Carry will be set to logic 1; otherwise it is reset.

② Zero flag (Z). It will be set if the sum is zero; otherwise, it is reset.

③ Negative flag (N). When this flag used as a general-purpose state indicator of the most-significant bit of the accumulator, the flag will be set if the bit7 of the result is a logic 1; otherwise it is reset. During the signed numbers are being added the N flag will be set if the result is negative and reset if it is positive.

④ Overflow flag (V). This flag is applicable only for signed numbers arithmetic operation. If both positive or both negative numbers are added and the result exceeds $+127_{10}$ or -128_{10}, which causes bit7 of the accumulator to be changed, then the V flag will be set; otherwise it is reset.

For add operation, the N and V flags are relevant only if signed numbers are being added. To add multi-byte numbers, simply clear the Carry flag before adding the low-order bytes, and then execute a series of LDA (load) / ADC (add) / STA (store) instructions, once for each byte to be added. For example, a double-precision addition routine

```
 ; This routine adds two 16-bit numbers. One number is stored in locations
 ; $30 and $31, the other is stored in locations $32 and $33.
 ; The sum replaces the number in locations $30 and $31.
 ;
DPADD     CLC                  ; CARRY = 0
          LDA    $30           ; Add low-order bytes
          ADC    $32
          STA    $30
          LDA    $31           ; Add high-order bytes
          ADC    $33
          STA    $31
```

Subtraction

Instruction	Codes	Cycles / Bytes	Status flags affected
SBC imm	E9xx	2/2	N，V，Z，C
SBC abs	EDaabb	4/3	N，V，Z，C
SBC zpg	E5aa	3/2	N，V，Z，C
SBC abs,x *	FDaabb	4/3	N，V，Z，C
SBC abs,y *	F9aabb	4/3	N，V，Z，C
SBC zpg,x	F5aa	4/2	N，V，Z，C
SBC (zpg),y *	F1aa	5/2	N，V，Z，C
SBC (zpg,x)	E1aa	6/2	N，V，Z，C

Like addition operation, most microprocessors have two subtract instructions, one just subtracts the operands and the other includes a borrow in the subtraction. The former instruction, called “**SUB**” instruction, is used to subtract single byte numbers or to subtract the low-order bytes of two multi-byte operands. The latter instruction, call “**SBC** (Subtract with Borrow – contributed by the Carry flag)”, is used to subtract the higher-order bytes of two multi-byte operands.

Since most subtractions involve multi-byte numbers, so the 6502 has only one subtract instruction — SBC. Due to the SBC always includes borrow into the subtraction, you must observe the following rule: The Carry flag (C) must be set to a 1 or accounted for before subtraction.

The instruction of setting the Carry flag is SEC (Set Carry flag).

Why set the Carry to 1? Because the 6502 uses the **complement** of Carry flag as borrow, rather than its true value, to implement the SBC instruction. Symbolically, the instruction of SBC will be represented as:

$$A = A - M - \overline{C}$$

Where, A is the accumulator

M is the contents of memory (or an immediate value)

$\overline{C}$ is the complement of the Carry

With the Carry set requirement, single-byte subtraction operations look like:

```
SEC
SBC        $71
```

The contents of zero page memory location $0071 are subtracted from the accumulator.

Except absolute addressing (indexed or un-indexed, three bytes), refer to 1-2-12, the all SBC instructions are two bytes long. The SBC instructions will affect the following four flags in processor status register:

① Carry flag (C). If the result is positive or zero then the Carry flag will be set. It will be reset if the result is negative (including a borrow).

② Zero flag (Z). It will be set if the result is zero; otherwise, it is reset. Please note that if Carry and Zero are both set, the result is zero. If Carry is set and Zero (Z) is reset, the result is positive.

③ Negative flag (N). When the flag used as a general-purpose state indicator of the most-significant bit of the accumulator, the flag will be set if the bit7 of the result is a logic 1; otherwise it is reset. During the signed numbers are being subtracted the N flag will be set if the result is negative and reset if it is positive.

④ Overflow flag (V). This flag is applicable only for signed numbers arithmetic operation. If two unlike-signed numbers (one positive and the other is negative) are subtracted and the result exceeds $+127_{10}$ or -128_{10}, which causes bit7 of the accumulator to be changed, then the V flag will be set; otherwise it is reset.

Like addition operation routine, to subtract multi-byte numbers, you just set the Carry flag before subtracting the low-order bytes, and then execute a series of LDA (load) / ADC (add) / STA (store) instructions, once for each byte to be subtracted. For example, a double-precision addition routine:

```
         ; This routine subtracts a 16-bit number stored in locations
         ; $0030 and $0031 form another number stored in locations
         ; $0032 and $0033.
         ; Locations $0030 and $0032 hold the low-order bytes of the numbers.
         ; Store the result in locations $0030 and $0031.
DPSUB    SEC                  ; Carry = 1
         LDA   $30            ; Subtract low-order bytes
         SBC   $32            ;
         STA   $30            ;
         LDA   $31            ; Subtract high-order bytes
         SBC   $33            ;
         STA   $31            ;
```

You can negate a number (in two's complement form) via the SBC instruction. It's very easy to get the negative number by subtracting the positive form of the number from zero. For instance, the following routine negates the contents of zero page memory location $0031:

```
         SEC                  ; Carry = 1
         LDA   #00            ; Accumulator = 0
         SBC   $31            ; Subtract $0031
         STA   $31            ;   and re-store in memory $0031
```

Signed Number Arithmetic

Actually, the operation of addition and subtraction makes no difference whether the numbers are signed or unsigned. However, that doesn't mean every addition and subtraction routine, mentioned above, can be used with numbers from both signed and unsigned systems. That only means the mechanics portions of the addition and subtraction of these routines are universal. The signed arithmetic routine may include some additional instructions that will treat a negative result differently from a positive result, such as an overflow conditional operation (i.e., the sign bit is altered) may be treated differently from one in which no overflow occurs.

Except as a general-purpose state indicator of the most-significant bit of the accumulator for N flag, the N and V flags are applicable only for signed numbers arithmetic operation, that is they are meaningless for un-signed numbers arithmetic operation.

In signed numbers arithmetic routine, the final status of the two flags usually used to decide how to process the result. The N flag only reflects the result is positive or negative, but the V flag presents whether the contents of the accumulator is valid or not.

If the V flag is set, the contents of the accumulator are invalid. The V flag will be set if the addition of two like-signed numbers or the subtraction of two unlike-signed numbers, which will produce a result more positive than $+127_{10}$ or more negative than -128_{10}; otherwise, the V flag is reset. Once set, the overflow flag can be reset with a special one-byte instruction – CLV (Clear Overflow(V) flag). The overflow flag will be reset automatically at the beginning of the next ADC or SBC instruction.

Instruction	Codes	Cycles / Bytes	Status flags affected
CLV	B8	2/1	V=0

1-2-5 Increment and Decrement Instructions

There are several instructions that can increment and decrement the contents of the X, Y registers, or memory location in the 6502.

Increment / Decrement Registers

Instruction	Codes	Cycles / Bytes	Status flags affected
DEX	CA	2/1	N，Z
DEY	88	2/1	N，Z
INX	E8	2/1	N，Z
INY	C8	2/1	N，Z

The X and Y registers used as index registers, which can also function as general-purpose counter in a various application.

It is easy to access consecutive memory locations by using increment and decrement instructions for both of these general-purpose registers. They are:

Instruction	**Description**
DEX	DEcrement index X by one
DEY	DEcrement index Y by one
INX	INcrement index X by one
INY	INcrement index X by one

These implied addressing instructions each occupies one byte in memory and take two cycles to execute. Furthermore, they affect two flags in the processor status register:

① The Negative flag (N). It will be set if the b7 = 1 of the result after being incremented or decremented; otherwise, N is reset. That indicates a negative result being generated if you are operating with a signed numbers system.

② The Zero flag (Z). It will be set if the result of the increment or decrement operation is zero; otherwise, it is reset.

The example below shows a routine that copies the contents of eight consecutive bytes in memory into another portion of the memory. In this example, the X register functions as an index register and Y register acts as the byte counter. The last instruction – BNE, has not been described yet, but all it does is make the 6502 loop back and load another byte (via the LDA $30,X instruction at label NXTBYT) until the byte counter – Y register has been decreased to zero.

```
; This routine copies an eight-byte block of memory, starting at location
;   $0030, into another part of memory, starting at location $0300
          LDX   #00          ; Index = 0
          LDY   #08          ; Byte counter = 8
NXTBYT    LDA   $30，X       ; Load next byte
          STA   $0300，X     ; Store next byte
          INX                ; Increment index
          DEY                ; Decrement index
          BEN   NXTBYT       ; Loop until all bytes copied
```

Increment / Decrement Memory

Instruction	Codes	Cycles / Bytes	Status flags affected
INC abs	EEaabb	6/3	N，Z
INC zpg	E6aa	5/2	N，Z
INC abs,x	FEaabb	7/3	N，Z
INC zpg,x	F6aa	6/2	N，Z
DEC abs	CEaabb	6/3	N，Z
DEC zpg	C6aa	5/2	N，Z
DEC abs,x	DEaabb	7/3	N，Z
DEC zpg,x	D6aa	6/2	N，Z

In some cases, counters are maintained in memory rather than in X or Y registers. There are two instructions allow you to perform increment or decrement operation on memory without using ADC or SBC. They are:

Instruction	**Description**
DEC	DECrement memory by 1
INC	INCrement memory by 1

The two instructions above permit four addressing modes – absolute, zero page, absolute indexed X, and zero page indexed X (refer to 1-2-12). These instructions affect the same flags in the processor status register – Negative (N) and Zero (Z), as the register increment and decrement instructions.

Unlike the arithmetic instructions – ADC and SBC, the INC and DEC instructions do not affect the Carry flag. So, If you are using a multi-byte memory counter to increment, you must check the Z flag (rather than the C flag) to decide whether or not to increment the next higher byte; if Z is set (indicates the current byte's value is zero and generates a carry), then increment. Similarly, if a multi-byte counter must be decremented, you must check the N flag to decide whether or not to decrement the next higher byte; if the DEC instruction caused N to switch from 0 to 1 (indicates a borrow happened), decrement.

1-2-6 Logic Instructions

Instruction	Codes	Cycles / Bytes	Status flags affected
AND imm	29xx	2/2	N，Z
AND abs	2Daabb	4/3	N，Z
AND zpg	25aa	3/2	N，Z
AND abs,x *	3Daabb	4/3	N，Z
AND abs,y *	39aabb	4/3	N，Z
AND zpg,x	35aa	4/2	N，Z
AND (zpg),y *	31aa	5/2	N，Z
AND (zpg,x)	21aa	6/2	N，Z

Instruction	Codes	Cycles / Bytes	Status flags affected
EOR imm	49xx	2/2	N，Z
EOR abs	4Daabb	4/3	N，Z
EOR zpg	45aa	3/2	N，Z
EOR abs,x *	5Daabb	4/3	N，Z
EOR abs,y *	59aabb	4/3	N，Z
EOR zpg,x	55aa	4/2	N，Z
EOR (zpg),y *	51aa	5/2	N，Z
EOR (zpg,x)	41aa	6/2	N，Z

Instruction	Codes	Cycles / Bytes	Status flags affected
ORA imm	09xx	2/2	N，Z
ORA abs	0Daabb	4/3	N，Z
ORA zpg	05aa	3/2	N，Z
ORA abs,x *	1Daabb	4/3	N，Z
ORA abs,y *	19aabb	4/3	N，Z
ORA zpg,x	15aa	4/2	N，Z
ORA (zpg),y *	11aa	5/2	N，Z
ORA (zpg,x)	01aa	6/2	N，Z

Sometimes you will want to check just one or more bits in a memory location or register, rather than entire 8-bit byte. In this case the logic instructions of the 6502 are useful for programmer.

There are three logic instructions, which operate on the accumulator with the contents of a memory location or immediate value specified in the operand.

Instruction	**Description**
AND	AND memory with accumulator
EOR	Exclusive-OR memory with accumulator
ORA	OR memory with Accumulator

All instructions occupy two bytes in memory if a zero page or immediate addressing mode is used, and three bytes in memory if an absolute addressing mode is used. All three instructions affect two flags:

① The Zero flag. It will be set if the result is zero; otherwise, it is reset.
② The Negative flag. It will be set if the bit7 = 1 of the result; otherwise, it is reset.

AND Instruction

AND instruction is used to filter, mask, or strip out (clear to zero) certain bits in the accumulator, so that some form of processing can be performed on the remaining bits. If each bit in which both memory and accumulator are in logic 1 then the bit in the accumulator is set to 1 after the instruction; otherwise that bit is reset to 0. Table 1-2-1 summarize the AND result:

Table 1-2-1 Logic AND operation

Memory bit	Accumulator bit	Result in Accumulator
0	0	0
0	1	0
1	0	0
1	1	1

The AND instruction is useful for testing the accumulator bits selected for a 1 value (to check a status byte, perhaps, to find out which bits are "on"), or to mask out bits that are no interest in a particular program application. The ASCII characters 0 to 9 are assigned the value listed in Table 1-2-2 (assume the most-significant bit is always a logic 0). If the four most-significant bits are masked out of the ASCII codes so that they are reset to zero, the remained four least significant bits are same with the BCD value. For example,

ASCII code of 5_{10} $\underline{0\ 0\ 1\ 1}\ 0\ 1\ 0\ 1_2$

$\downarrow$mask the four most-significant bits to zero

BCD code of 5_{10} $\underline{0\ 0\ 0\ 0}\ 0\ 1\ 0\ 1_2$

It will be very easy to mask the four most-significant bits to logic 0 by using logic AND instruction.

Table 1-2-2 ASCII codes for Character 0 to 9

Character	ASCII		BCD
	Hexadecimal	Binary	
0	30	0 0 1 1 0 0 0 0	0 0 0 0
1	31	0 0 1 1 0 0 0 1	0 0 0 1
2	32	0 0 1 1 0 0 1 0	0 0 1 0
3	33	0 0 1 1 0 0 1 1	0 0 1 1
4	34	0 0 1 1 0 1 0 0	0 1 0 0
5	35	0 0 1 1 0 1 0 1	0 1 0 1
6	36	0 0 1 1 0 1 1 0	0 1 1 0
7	37	0 0 1 1 0 1 1 1	0 1 1 1
8	38	0 0 1 1 1 0 0 0	1 0 0 0
9	39	0 0 1 1 1 0 0 1	1 0 0 1

For example, you store the ASCII code of 5_{10} in accumulator and ANDing the value with 00001111_2, and then you will get the result you want.

```
LDA     #$35
AND     #$0F
```

The accumulator will contain 00000101, or BCD 5.

OR Instruction

The ORA (OR Memory with Accumulator) instruction produces a logic 1 result in each bit of accumulator for corresponded bit of memory or accumulator contains a logic 1. The ORA instruction is usually used to set some bits of the accumulator to a logic 1, for example:

```
ORA     #01
```

It will set the least-significant bit of the accumulator to a logic 1, and leave all other bits unchanged.

Table 1-2-3 Logic OR operation

Memory bit	Accumulator bit	Result bit in Accumulator
0	0	0
0	1	1
1	0	1
1	1	1

EOR Instruction

The EOR (Exclusive-OR Memory with Accumulator) instruction is used to determine which bits differ between two operands, and it can also be used to complement selected accumulator bits. The EOR instruction produces a logic 1 result in each bit of accumulator for corresponded bit of memory or accumulator (but not both) contains a logic 1; otherwise all other bit positions are cleared to 0. For example,

```
EOR     #$0F
```

The instruction will complement the four least-significant bits of the accumulator and the four most-significant bits leave unchanged. The EOR is also used to determine whether two values are identical or not, EOR will set the Zero flag (Z) if, and only if, the contents of memory are identical to the contents of the accumulator. Table 1-2-4 summarizes the EOR conditions:

Table 1-2-4 Logic EOR operation

Memory bit	Accumulator bit	Result bit in Accumulator
0	0	0
0	1	1
1	0	1
1	1	0

1-2-7 Jump, Branch, Compare, and Bit-Check Instruction

Most of the program examples with instructions were executed in the order in which they appeared in program. Control has not been transferred to another section of a program based on the results received by executing an instruction or a series of instructions. Now, we will discuss how program execution can be transferred from one section of a program to another section, and why this is useful.

The Jump Instruction

Instruction	Codes	Cycles / Bytes	Status flags affected
JMP abs	4Caabb	3/3	
JMP (abs)	6Caabb	5/3	

Have you ever-read magazines and come across a direction, like "Jump to page 10"? That is also a jump instruction.

The jump instructions of the 6502 cause the next operation to happen at a point rather than the next consecutive memory location. And the jump instruction of the 6502 is unconditional; it occurs every time the 6502 encounters it in a program.

The operand of a jump (JMP) instruction is usually a label, so you will often see JMP instruction like:

```
                  :
                  :
                  ADC   $30
                  INX
                  JMP   THERE
         HERE     STA   $33
                  :
                  :
         THERE    SEC
                  SBC   $32
                  :
```

The program will jump to the instruction labeled THERE in the example, and it executes SEC, SBC $32…and so on. Will the STA $33 instruction located label HERE ever be executed? Yes, but only if an instruction in some portion of the program transfers control to that location.

The JMP instruction is used to skip over a group of instructions that are executed under some conditions, or a group of instructions that are executed during some other part of the program. The JMP instruction can operate with absolute addressing or indirect absolute addressing (refer to 1-2-12). Since both modes are absolute, the jump can be to any place in memory.

The JMP instruction occupies three bytes in memory and takes three cycles to execute with absolute addressing and five cycles to execute with indirect absolute addressing. Furthermore, the JMP instruction does not affect any flag in processor status register.

The Branch Instructions

Instruction	Codes	Cycles / Bytes	Status flags affected
BCS rel **	B0rr	2/2	
BCC rel **	90rr	2/2	
BEQ rel **	F0rr	2/2	
BNE rel **	D0rr	2/2	
BMI rel **	30rr	2/2	
BPL rel **	10rr	2/2	
BVS rel **	70rr	2/2	
BVC rel **	50rr	2/2	

Like the JMP instruction, the branch instructions cause the program execution to be transferred to a specific memory location. Whereas the JMP instruction transfers control to some absolute address in memory, the branch instructions transfer control to a relative address in memory – transfer control to a specified number of memory locations forward or backward from the next instruction after the branch instruction. Another difference between the JMP instruction and branch instructions is that the branch instructions are **decision-making** instruction or called as **conditional jump**. Each branch instructions check the status of a single flag in the processor status register. If the state of the flag meets the requirements specified by the branch instruction, then the instruction is executed; otherwise, execution continues with the next consecutive instruction in the program. The branch instructions and flag they check are summarized in Table 1-2-5.

Table 1-2-5 The Branch Instruction

Instructions	Description	Causes a Branch, If
BCC	Branch on Carry Clear	C = 0
BCS	Branch on Carry Set	C = 1
BEQ	Branch on result EQual zero	Z = 1
BNE	Branch on result Not Equal zero	Z = 0
BMI	Branch on result MInus	N = 1
BPL	Branch on result PLus	N = 0
BVS	Branch on oVerflow Set	V = 1
BVC	Branch on oVerflow Clear	V = 0

You will find from Table 1-2-5, the branch instructions are executed based on the status of four flags – Carry (C), Negative (N), Zero (Z), and Overflow (V).

As you know, the branch instruction transfers are taken relative to the branch instructions, so that these instructions operate in only relative addressing mode. For this reason, the branch can just stretch across to 127 bytes forward or 128 bytes backward, from the branch instruction. You can always combine a JMP instruction with the branch instruction to branch anywhere in memory.

For example, here is how a program might execute a branch on the Carry-set condition to an instruction (at CSET) that is more than 127 locations past, or 128 locations ahead of the BCC instruction:

```
                BCC     CCLEAR        ; Go to CCLEAR on Carry  = 0
                JMP     CSET          ; Go to CSET on Carry = 1
        CCLEAR  LDA     $20
```

Table 1-2-6 gives the hexadecimal equivalents for the full range of branch, both forward and backward. If you have a SYSTEM that offers the mnemonic entry mode, you can ignore this table and enter the absolute (16-bit) address itself, rather than the relative displacement; otherwise, here is how you can use Table 1-2-6.

Table 1-2-6 Hexadecimal operands for Branch Instructions

Forward Relative Branch Table

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Backward Relative Branch Table

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
1	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
2	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
3	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
4	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
5	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
6	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
7	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

If you want code a Branch on Carry Clear (BCC) instruction that will branch 89 bytes forward if the condition is met, you would look up 89 in the Forward Relative Branch Table. Position 89 is in column 9 and row 5. The column gives the least-significant hexadecimal digit value and the row gives the most-significant hexadecimal digit value, so the BCC instruction should be coded like:

```
        BCC     $59
```

All branch instructions occupy two bytes in memory. These instructions are executed in two cycles if the condition is not met and three cycles if the condition is met (four cycles if the branch crosses a page boundary).

The Compare Instruction

Up to this point, the branch instructions check on the state of a status flag that reflects the result in an arithmetically altered register.

Instruction	Codes	Cycles / Bytes	Status flags affected
CMP imm	C9xx	2/2	N, Z, C
CMP abs	Cdaabb	4/3	N, Z, C
CMP zpg	C5aa	3/2	N, Z, C
CMP abs,x *	Ddaabb	4/3	N, Z, C
CMP abs,y *	D9aabb	4/3	N, Z, C
CMP zpg,x	D5aa	4/2	N, Z, C
CMP (zpg),y *	D1aa	5/2	N, Z, C
CMP (zpg,x)	C1aa	6/2	N, Z, C
CPX imm	E0xx	2/2	N, Z, C
CPX abs	ECaabb	4/3	N, Z, C
CPX zpg	E4aa	3/2	N, Z, C
CPY imm	C0xx	2/2	N, Z, C
CPY abs	CCaabb	4/3	N, Z, C
CPY zpg	C4aa	3/2	N, Z, C

There are some situations, however, where you would like to base a branch decision on the contents of a register or memory location in its unaltered state. The 6502 microprocessor has three instructions for that purpose, which are called **compare** instructions. The compare instructions affect three flags of the processor status register – Carry, Zero, and Negative that can be check with branch instructions **without altering the contents of the operands**. The three compare instructions:

Instruction	**Description**
CMP	CoMPare memory and accumulator
CPX	ComPare memory and index X
CPY	ComPare memory and index Y

The CMP instruction supports eight different addressing modes; the same ones supported by the ADC and SBC instructions. And, the CPX and CPY instructions operate with just three addressing modes (immediate, absolute, and zero page— refer to 1-2-12), because the X and Y registers function primarily as counters and indexes that do not require much elaborate addressing capability.

The compare instructions subtract an immediate value or the contents of a memory location from the addressed register, but do not save the result in the register. The only indications of the result are the states of the three status flags – Negative (N), Zero (Z), and Carry (C). The combination of these three flags can indicate whether the register contents are **less than**, **equal to**, or **greater than** the operand "data" (immediate value or contents of the addressed memory location). Table 1-2-7 summarizes the result indicator for the compare instructions:

Table 1-2-7 Compare Instruction Result

	N	Z	C
A, X, or Y < memory	1*	0	0
A, X, or Y = memory	0	1	1
A, X, or Y > memory	0*	0	1

* Valid only for "two's complement" compare.

Example 1-2-1 contains a routine that checks whether the contents of two memory locations are identical, and sets a flag in memory.

Example 1-2-1 Checking two memory locations for equality

```
; If the contents of location $71 and $72 are same then
;   the routine sets the memory location $73 to “1”; otherwise,
;   to “0”.
            LDX   #00    ; Initialize flag to zero
            LDA   $71    ; Load first value
            CMP   $72    ; Are the two values identical?
            BEN   DONE   ; Non-equal, jump to label DONE
            INX          ; Yes, set the flag
DONE        STX   $73    ; Store flag in memory location $73
```

Example 1-2-2 contains another memory-to-memory comparison routine, and stores the greater value in the higher order location in memory. Example 1-2-3 contains a register-to-constant comparison routine in which two branch instructions are used with one compare instruction, so that the ”less than”, “equal to”, and “greater than” conditions are checked.

Example 1-2-2 Arrange two numbers in order of value

```
; This routine arranges two numbers in locations $71 and $72
;   in order of value, with the lower-valued number in location $71.
;
            LDA   $72    ; Load second number into accumulator
            CMP   $71    ; compare the two numbers
            BCS   DONE   ; DONE if first is less than or equal to second;
            LDX   $71    ; Otherwise, exchange the contents of the memory
            STA   $71    ;   location $71 and $72
            STX   $72    ;
     DONE   :
            :
```

Example 1-2-3 A three-way decision routine

```
; This routine stores the contents of the accumulator into location $71, $72
;   or $73, depending on whether the accumulator less than 7, equal to 7, or
;   greater than 7, respectively.
;
            CMP   #07    ; Compare accumulator to 7
            BCS   EQGT7  ; Equal to or greater than 7 branch to EQGT7;
            STA   $71    ; Otherwise, accumulator is less than 7,
                         ;   stored in location $71
            JMP   DONE
EQGT7       BNE   GT7
```

```
                STA    $72        ; Accumulator equals to 7
                JMP    DONE
GT7             STA    $73        ; Accumulator is greater than 7
DONE            :
                :
```

Example 1-2-4 is a routine that will move consecutive memory bytes (max. 256 bytes), with the CPX instruction doing the “all bytes moved?” check each time a byte is moved.

Example 1-2-4 A multi-byte move routine

```
; This routine moves up to 256 bytes of memory, starting at location
;   $71, to another portion of memory, starting at location $0300
; The byte count is contained in location $10
                LDX    #00              ; Index = 0
NXTBYT          LDA    $71，X           ; Load next byte
                STA    $0300，X         ; Store next byte
                INX                     ; Increment index
                CPX    $10              ; All bytes moved?
                BNE    NXTBYT           ; If not, move next byte
                :
```

The BIT Instruction

Instruction	Codes	Cycles / Bytes	Status flags affected
BIT abs	2Caabb	4/3	N=M7, V=M6, Z
BIT zpg	24aa	3/2	N=M7, V=M6, Z

The compare instructions compare two “entire” 8-bit values. If you want to compare a certain bits in a memory location, you may use AND logic instruction to mask out the unwanted bits. However, in the process of masking out the unwanted bits, the AND instruction will destroys the mask contained in the accumulator. Certainly, the mask could be reloaded, but this requires additional time. The same job can be done without altering the accumulator by executing a **BIT** instruction.

The Check Bits in memory with accumulator (**BIT**) instruction alter neither accumulator nor memory, and like the compare instructions, record status information in the processor status register.

① The Negative flag (N) receives the initial (un-ANDed) value of bit7 of the memory location being checked.
② The Overflow flag (V) receives the initial (un-ANDed) value of bit6 of the memory location being checked.
③ The Zero flag (Z) is set if the AND operation generates a zero result; otherwise, it is reset.

The BIT instruction performs the same operation as the AND instruction, but it does not enter the ANDed result in the accumulator. Furthermore, the AND instruction affects two flags – N and Z and both flags reflect the post-ANDed status, the BIT instruction affects three flags – Z, N and V, but only the Z flag reflects the post-ANDed status.

The BIT instruction only used in absolute and zero page addressing. And, it's worth to note that only the Z flag reflects the result of the simulated AND operation; the N and V flags represent the values of the two high-order memory bits in their unaltered state. For example, waiting for a memory bit to become logic0:

```
            LDA     #08          ; Mask off all bits except b3
LOOP        BIT     $3210        ; The b3 (contents of location $3210) = 0 ?
            BNE     LOOP         ; Keep checking until the b3 = 0
MEET        :                    ;   then continue here
```

In that example, If the "BNE LOOP" was "BEQ LOOP" then just check a logic 1 in b_3. The BIT instruction can check more than one bit at a time, but with some limitation. When two or more bits are set in the accumulator mask, a subsequent BIT instruction will set Z flag only if both check bits are logic 0 in memory; otherwise, will reset Z if either or both check bits are logic 1 in memory.

However, the limitation will disappear if only one bit of the check bits is in the b_0 to b_5 and the other check bit(s) are either or both in b_6 to b_7. The bit6 and bit7 can be checked with their own branch instructions (BVS and BVC for bit6, BMI and BPL for bit7) because b_6 and b_7 provide AND-independent status indicator. For example, waiting for any of three memory bits to become logic 0.

```
            LDA     #08          ; Select bit3 for checking with mask
LOOP        BIT     $3210        ; checked memory
            BEQ     MEET         ; Branch on b3  = 0
            BVC     MEET         ; Branch on b6 = 0
            BMI     LOOP         ; LOOP if b7 = 1
MEET        :                    ; Continue here when b3 or b6 or b7 = 0
```

It is useless if you are checking elements of tables or any other indexed or indirect-accessed data because the BIT instruction is restricted to absolute and zero page addressing.

However, the BIT instruction can often replace a compare instruction for data at known addresses. The easy access to bit6 and bit7 (via V and N flags) makes these bits ideal for storing status information. And, the N and V may be widely used in **interrupt request scheme** of the input / output structure of the 6502 system

1-2-8 SHIFT and ROTATE Instructions

Instruction	Codes	Cycles / Bytes	Status flags affected
ASL abs	0Eaabb	6/3	N, Z, C
ASL zpg	06aa	5/2	N, Z, C
ASL acc	0A	2/1	N, Z, C
ASL abs,x	1Eaabb	7/3	N, Z, C
ASL zpg,x	16aa	6/2	N, Z, C

Instruction	Codes	Cycles / Bytes	Status flags affected
LSR abs	4Eaabb	6/3	N, Z, C
LSR zpg	46aa	5/2	N, Z, C
LSR acc	4A	2/1	N, Z, C
LSR abs,x	5Eaabb	7/3	N, Z, C
LSR zpg,x	56aa	6/2	N, Z, C

Instruction	Codes	Cycles / Bytes	Status flags affected
ROL abs	2Eaabb	6/3	N, Z, C
ROL zpg	26aa	5/2	N, Z, C
ROL acc	2A	2/1	N, Z, C
ROL abs,x	3Eaabb	7/3	N, Z, C
ROL zpg,x	36aa	6/2	N, Z, C

Instruction	Codes	Cycles / Bytes	Status flags affected
ROR abs	6Eaabb	6/3	N, Z, C
ROR zpg	66aa	5/2	N, Z, C
ROR acc	6A	2/1	N, Z, C
ROR abs,x	7Eaabb	7/3	N, Z, C
ROR zpg,x	76aa	6/2	N, Z, C

The 6502 microprocessor has four instructions that cause the 8-bit contents of an operand (a memory location or the accumulator) to be displaced 1 bit position to the left or to the right. Two of these instructions “shift” the operand, the other two “rotate” the operand.

For all the four instructions, the Carry flag acts as a “ninth bit” of the operand in that it receives the value of the bit that is displaced out of one end of the accumulator or memory location (bit0 for a right shift, bit7 for a left shift). In a **shift** operation, the vacated bit position at the opposite end of the operand (bit0 for a left shift, bit7 for a right shift) is reset to 0. In a **rotate** instruction, the vacated bit position at the opposite end of the operand receives the value of the Carry flag before rotation.

Instruction	**Description**
ASL	Accumulator Shift Left
LSR	Logical Shift Right
ROL	ROtate Left
ROR	ROtate Right

Operating on the accumulator, you just put an “A” in the operand field of the instruction, like:

ASL	A	(accumulator)	occupy 1 byte in memory

Operating on memory can be conducted in any of four addressing modes – absolute, zero page, zero page indexed X, and absolute indexed X. The formats like:

ASL	$1234	(absolute)	occupy 3 bytes in memory
ASL	$2A	(zero page)	occupy 2 bytes in memory
ASL	$2A，X	(zero page indexed X)	occupy 2 bytes in memory
ASL	$1234，X	(absolute indexed Z)	occupy 3 bytes in memory

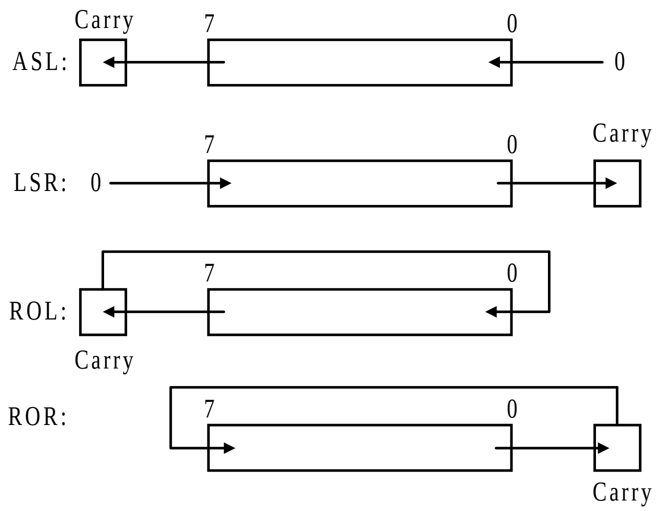


Fig. 1-2-1 Diagram of the SHIFT and ROTATE Instructions

Besides the Carry flag is affected, the shift and rotate instructions affect two other flags in the processor status register:

① ASL, ROL, and ROR cause the Negative flag (N) to be set if bit7 of the shifted result is set to logic 1; otherwise, it is reset. The LSR instruction always causes the Negative flag to be reset because it shifts a "0" into bit7.

② The Zero flag (Z) will be set if the shifted result is "0"; otherwise, it is reset.

Carry Flag	**Bit position 7 6 5 4 3 2 1 0**		
1	0 1 1 0 1 0 0 0	before shift (hex 68, decimal 104)	$\boxed{\text{Original value}}$
0	1 1 0 1 0 0 0 0	after ASL (hex D0 decimal 208)	Multiplies original value by 2
0	0 0 1 1 0 1 0 0	after LSR (hex 34, decimal 52)	Divides original value by 2
0	1 1 0 1 0 0 0 1	after ROL (hex D1 decimal 209)	
0	1 0 1 1 0 1 0 0	after ROR (hex B4 decimal 180)	

Unsigned Numbers Shift

Single-byte numbers can be shifted using only the shift instruction, ASL or LSR. Multi-byte numbers require a combination of these shift and rotation instructions. In multi-byte shift operation, the Carry flag is used to propagate bit values that have been displaced out of previously shifted bytes.

For example, a 24 bits (three bytes) unsigned numbers stored in location $71 (low-order byte), $72, and $73 (high-order byte) can be left-shifted with:

```
ASL     $71        ; First shift low-order byte
ROL     $72        ; Shift middle byte w/Carry flag
ROL     $73        ; Shift high-order byte w/Carry flag
```

Multi-byte numbers right shifts operation in left-to-right order, where the high-order byte is shifted first, with an LSR instruction.

The right-shift routine for three bytes numbers starting at location $71 is:

```
          LSR     $73          ；First shift high-order byte
          ROR           $72          ；Shift middle byte w/Carry flag
          ROR           $71          ；Shift high-order byte w/Carry flag
```

Signed Numbers Shift

The four shift and rotate instructions perform as “logic” shifts without regard to sign. If a signed number is right-shifted, the sign bit will be shifted 1 bit position to the right, and its value will be replaced with a 0. If a signed number is left-shifted, the sign will be shifted into the Carry flag, and its value will be replaced by bit6. Your program must restore the **displaced sign** value. Example:

Example 1-2-5 A left-shift routine for multi-precision signed numbers

```
;This routine left-shifts a multi-precision signed number stored in memory
;  starting at location $71. The length of the number, in bytes, is contained
;  in location $2F
;
          LDY   $2F        ；Load byte count into Y register
          ASL   $71        ；Left shift low-order byte
          LDX   #01        ；Byte index  = 1
          DEY              ；Decrement byte count
NXTBYT    ROL   $71，X     ；Shift next byte
          INX              ；Update byte index
          DEY              ；   and byte count
          BNE   NXTBYT     ；LOOP until all bytes shifted
;
;The code that follows restores the sign to the most-significant byte (MSBY)
;
          DEX              ；Make index point to MSBY
          LDA   $71，X     ；Load MSBY into accumulator
          BCC   MSB0       ；Sign = 1？
          ORA   #$80       ；If YES, put a 1 in sign bit
          JMP   SOVER
MSB0      AND   #$7F       ；If NO, put a 0 in sign bit
SOVER     STA   $71，X     ；Return MSBY to memory
          :
```

In a logical right shift, the sign is vacated by the operation. You can preserve the sign by recording its original value in the Carry flag and by using an ROR instruction to shift it into the most-significant byte.

Example 1-2-6 A right-shift routine for multi-precision signed numbers

```
;This routine right-shifts a multi-precision signed number stored in memory
;  starting at location $71. The length of the bytes number is contained in
;  location $2F
           CLC                  ;Prepare for sign = 0 shift
           LDX    $2F           ;Load byte count into X
           DEX                  ;Set index for MSBY
           LDA    $71，X        ;Load high-order byte
           BPL    MSB0          ;Sign = 1？
           SEC                  ;If YES, prepare for sign = 1 shift
MSB0       ROR    A             ;Shift high-order byte, and
           STA    $71，X        ;   return it to memory
           DEX                  ;Decrement index for next byte
NXTBYT     ROR    $71，X        ;Shift next highest byte
           DEX
           BPL    NXTBYT        ;LOOP until all bytes shifted
```

1-2-9 Register Transfer Instructions

Instruction	Codes	Cycles / Bytes	Status flags affected
TAX	AA	2/1	N，Z
TXA	8A	2/1	N，Z
TAY	A8	2/1	N，Z
TYA	98	2/1	N，Z
TSX	BA	2/1	N，Z
TXS	9A	2/1	

Although the 6502 is a memory-oriented architecture, it also has 6 one-byte instructions that allow you to copy the contents of one register into another without disturbing the source register.

Instruction	**Description**
TAX	Transfer Accumulator to index X
TAY	Transfer Accumulator to index Y
TSX	Transfer Stack pointer to index X
TXA	Transfer index X to Accumulator
TXS	Transfer index X to Stack pointer
TYA	Transfer index Y to Accumulator

Arithmetic Operation on the X and Y Registers

For X and Y registers, the 6502 just can increment or decrement them. The transfer instructions provide a simple way to copy X and Y into the accumulator for a more complex arithmetic operation, and have the result returned. Considering a case in which you want to access every tenth element in a list rather than consecutive elements.

The access instruction would be an index instruction, such as LDA LIST, Y, in which LIST is a label assigned to the starting location of the list. To access every tenth element, you must add 10 to the index Y between load operations.

How can you add 10 to Y register? One way is to code ten consecutive INY instructions. A more efficient way is to transfer Y to the accumulator, add 10 (immediate), and return the sum to Y for the next list access.

1-2-10 Stack Instruction

The stack of the 6502 is of the “last-in-first-out” variety. That is, the last item (a data byte) that is entered onto the stack is the first item to be extracted from the stack. This scheme causes data to be retrieved in the reverse order from which it was stored.

A stack address register called the **Stack Pointer** (SP or S), which always points to the next free memory location on the stack, used to access stack information. The Stack Pointer is automatically decremented after a byte is pushed onto the stack, and is automatically incremented before a byte is pulled from the stack, so the stack built in the direction of address 0.

The stack is implemented in page 1 (location $0100 ~ $01FF) of the address space. Thus, the Stack Pointer must be initialized by the user’s program to address $01FF when the power is applied. The stack can hold up to 256 bytes of information because a page comprised of 256 byte locations. The purposes of using the stack: (1) to save interrupt or subroutine return addresses, and (2) to temporarily save register content

Stack Pointer Instructions

The contents of SP are undefined when power is applied, and must be initialized by the user’s program. You can use a special instruction – TXS to initialize the Stack Pointer.

Instruction	**Description**
TXS	Transfer index X to Stack-pointer

That is an one-byte instruction with two cycles execution.

Since the stack starts at memory location $01FF, most system programs initialize the Stack Pointer to $FF (recall that the high-order address byte $01 is automatically supplied by the 6502). For example,

```
LDX     #$FF               ; Load $FF into X register.
TXS                        ; Transfer X register to Stack Pointer
```

A program, using another implied-addressing instruction, can also read the contents of the Stack Pointer:

Instruction	**Description**
TSX	Transfer Stack-pointer to index X

That is also an one-byte instruction with two cycles execution.

This instruction is rarely used because you do not care where in the stack information is being stored in the most cases.

PUSH and PULL Instructions

Instruction	Codes	Cycles / Bytes	Status flags affected
PHA	48	3/1	
PLA	68	4/1	N, Z
PHP	08	3/1	
PLP	28	4/1	N, V, B=1, D, I, Z, C

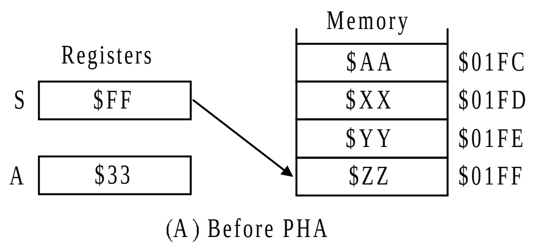
The 6502 microprocessor has four instructions that allow the contents of both the accumulator and the processor status register to be stored in the stack.

Instruction	**Description**
PHA	PusH Accumulator on Stack
PHP	PusH processor status on Stack
PLA	PulL Accumulator from stack
PLP	PulL Processor status from stack

All the four instructions are implied addressing instructions, which occupy only one byte in memory. The PHA and PHP instructions work identically, except they push different registers onto stack. In each case, the contents of the register are pushed onto the stack at the location being pointed to by the Stack Pointer. Then, the Stack Pointer is decreased by one to the next lower address. Neither instruction changes the contents of its source register.

Similarly, the PLA and PLP instructions increment the Stack Pointer to the next higher address, and load the contents of the memory location, addressed by the Stack Pointer, into the appropriate destination register.

Figure 1-2-2 shows the effect of the PHA instruction on the Stack. In Fig. 1-2-2 (A), the Stack Pointer (S) is pointing to the top of the stack (location $01FF), and the accumulator contains $33. After PHA is executed (Fig. 1-2-2 (B)), the Stack Pointer has been decremented to $FE and the contents of the accumulator have been stored in location $01FF. After the PUSH, if you execute a PLA instruction, the Stack Pointer will be incremented to point to $01FF and the accumulator will be loaded with the contents of location $01FF.



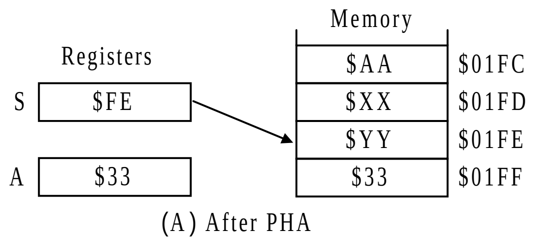


Fig. 1-2-2 How a register is pushed onto the stack

Why someone needs to save registers on the stack? The reason is to preserve their contents while they are being manipulated by other programming operations. This is particularly true in subroutines.

It looks as if only the accumulator and the processor status register can be saved on the stack. That is true based on the instruction functions, but by using the register transfer instructions, you can move the X and Y registers into the accumulator, push them onto the stack, and later pull them off and restore them with additional register transfer instructions. Example 1-2-7 shows the coding required saving not only the accumulator and the processor status register, but also the X and Y registers on the stack. Note that the registers are pulled in the reverse order from which they were pushed.

Example 1-2-7 Saving all registers on the Stack

```
 ; Push registers onto the Stack instructions
PHP                        ; Save processor status register
PHA                        ; Save accumulator
TXA                 ; Save X register
PHA
TYA                        ; Save Y register
PHA
  :
  :           Some routine is executing here
  :
 ; Pull registers from Stack
PLA                        ; Restore Y register
TAY
PLA                        ; Restore X register
TAX
PLA                        ; Restore accumulator
PLP                        ; Restore processor status register
```

In most cases, the accumulator contents must always be pushed onto the stack before either X or Y register is pushed because the X and Y pushes must be preceded by a TXA or TYA instruction, which destroys the contents of the accumulator. Furthermore, if the processor status register is to be saved, it must also be pushed onto the stack before either X or Y is pushed because both TXA and TYA affect the processor status register's Negative (N) and Zero (Z) flags.

1-2-11 NO OPERATION Instruction

The No Operation (NOP) instruction is a simple one byte implied addressing instruction, which is generally used during program development. The NOP instruction performs no operation – it does not change any status flags, registers, or memory locations, but it does perform the very useful function of reserving space in memory. Since the NOP instruction occupies only one byte in memory, at least two NOP instructions should be inserted at the spot where space is to be reserved.

NOP instructions may be used to replace instructions that have been deleted, without requiring all of the branch and jump instruction addresses to be changed.

Instruction	Codes	Cycles / Bytes	Status flags affected
NOP	EA	2/1	

1-2-12 6502 Addressing Modes

One reason for the wide popularity of the 6502 microprocessor is the flexibility it offers in addressing. The 6502 has 13 addressing modes. We introduce the 13 addressing modes respectively, as following:

Immediate Addressing

In immediate addressing, the operand resides in the second byte of the instruction. An immediate operand is specified by placing a # prefix before the operand. For example,

```
LDA         #$33
```

is an instruction that loads hexadecimal 33 (decimal 51) into the accumulator. All instructions that use immediate addressing are two bytes long.

Absolute Addressing

Absolute addressing allows the direct addressing of any of the 65,536 memory locations in the address space of the 6502. All instructions that use absolute addressing require three consecutive memory locations for storage. The first byte is the op code of the instruction; the second and third bytes are the low-order and high-order bytes of the operand address, respectively. For instance,

```
LDA         $12C3
```

is an instruction that loads the contents of memory location $12C3 into the accumulator. If memory location $12C3 contains hexadecimal 3C, the accumulator will contain hexadecimal 3C after the LDA instruction is executed. The example instruction looks like these in memory:

Location	**Contents**	**Description**
nnnn	$AD	Op code for LDA with absolute addressing
nnnn+1	$C3	Low-order byte of address
nnnn+2	$12	High-order byte of address

As you can see, the instruction occupies three consecutive addresses. In other words, the instruction has three bytes long. As an assembler rule, the absolute address doesn’t use any sign to mark usually.

Zero Page Addressing

Zero page addressing is a form of absolute addressing in which the 6502 accesses only the first 256 locations in memory. These are hexadecimal addresses 0000 through 00FF (decimal addresses 0 through 255). Because the high-order byte of a zero page address is always zero, instructions that use zero page addressing are two-byte instructions; the first byte is the op code, the second byte is the low-order byte of a zero page address (00 through FF). The 6502 microprocessor will treat a two-digit operand as a zero page address. For example,

```
LDA         $2A
```

This is a zero-page addressing mode, loading the contents of memory location 002A into the accumulator. Except for two instructions, JMP (Jump) and JSR (Jump to Subroutine), all 6502 instructions that can use absolute addressing can also use zero page addressing. Considering the inherent saving in both storage space and execution time, the zero page should be used, whenever possible, to hold frequently accessed data. ("Zero page" instructions occupy one less byte in memory and take one less cycle to execute than their "absolute address" counterparts.) The zero page is also useful to store temporary data values.

Implied Addressing

The 6502 microprocessor has many instructions that don't need any operand, such as setting or clearing a bit in the processor status register, increasing or decreasing a register, or copying the contents of one register into another. The 6502 receives enough information from the op code alone – and employ what is called (appropriately) "implied addressing". Some examples are:

Mnemonic	**Description**
CLC	CLear Carry flag.
DEX	DEcrement the X register.
TAX	Transfer Accumulator to X register.

All implied addressing instructions occupy one byte memory location.

Indirect Absolute Addressing

Indirect absolute addressing is used by only one 6502 instruction – the Jump (JMP) instruction. The JMP instruction loads the program counter with a new address at which the 6502 is to fetch its next instruction. The JMP instruction can use either absolute addressing or indirect absolute addressing. With absolute addressing, the operand of the JMP instruction is the destination address that is to be put into the program counter. With indirect absolute addressing, the operand of the JMP instruction is the address of the first of two memory locations that contain the 16-bit destination address. In other words, the real 16 bits destination addresses are stored in memory, program should according to the operand of the JMP to fetch them. It' a rule that an indirect absolute operand is specified by enclosing it in parentheses.

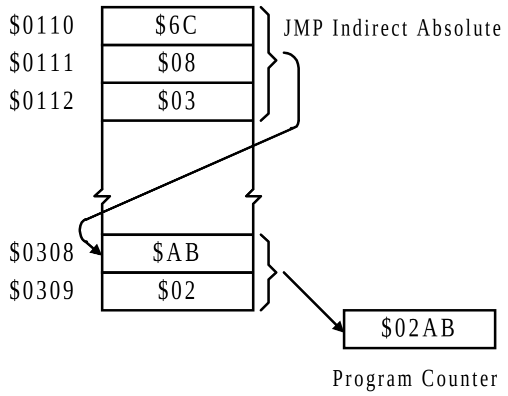


Fig. 1-2-3 Indirect absolute addressing

For example,

```
JMP     ($0308)
```

causes the program counter to be loaded with the low-order address contained in memory location $0308 and the high-order address contained in memory location $0309. Fig. 1-2-3 illustrates this example, with $02AB as the final (effective) address and with the instruction stored in memory location $0110, $0111, and $0112.

If your destination is address $02AB, why not just use absolute addressing to store it in the program counter? The answer is that indirect absolute addressing allows us to work with variable destination addresses. For example, if the 6502 is installed in a system in which it must serve several peripheral devices, an indirect absolute addressing JMP instruction can be used to access a sequence of instructions appropriate to the peripheral that requires service.

We can prepare the series instructions for serving each peripheral device, and store them in various locations of memory respectively. For each device, the series instructions are put together continually. And then we choose the fixed consecutive two addresses, like $0308 and $0309, to store the begin addresses of each device. Whenever we want to serve any peripheral device, we just execute the instruction below:

```
JMP     ($0308)
```

Because we can use other instructions to put the begin addresses of any device, which requires service, into $0308 and $0309. In this case, the indirect absolute JMP instruction would always fetch the 16-bit address from the same pair of memory locations, but the 6502 would change the contents of these locations, depending on which peripheral device requires service. However, in absolute addressing mode, the program just jumps to the unique address specified in operand.

In a data processing application, a single 6502 microprocessor might be accepting from operators at several keyboards. In this case, the destination address that the 6502 is to jump to depend on which keyboard the 6502 is accepting data from at any particular time. Input from keyboard No. 1 will be stored in one place, input from keyboard No. 2 will be stored in another place, and so on. Indirect absolute addressing also allows the effective address to be in RAM even when the program is in ROM or PROM.

Absolute Indexed Addressing

In absolute indexed addressing, the effective address of the operand is computed by adding the contents of the X or Y register to the absolute address in the instruction. That is,

Effective address = Absolute address + X

or Effective address = Absolute address + Y

All absolute indexed instructions occupy three consecutive memory locations. The first location is for operand, second and third locations are for absolute addresses – low-order byte is prior to the high-order byte. The format of absolute indexed addressing is to attach a “X” or a “Y” to the address to specify absolute indexed operands. For example,

```
LDA     $12C3，X
```

That is an absolute indexed LDA instruction. If the X register contains $05, the instruction loads the contents of memory location $12C8 (i.e., $12C3 + $05) into the accumulator.

Absolute indexed addressing is especially useful for accessing data in a list. For this application, you would use the starting address of the list as the operand of the instruction and use the index register (X or Y) to store "displacement" - to specify the particular element in the list that you want to access. For example, if the content of index register is $05 then you will access the 6th element in the list. If you establish a loop in which X is incremented after each access, you can access a series of consecutive elements in the list.

Zero Page Indexed Addressing

Zero page indexed addressing is to zero page addressing as absolute indexed addressing is to absolute addressing. With zero page indexed addressing, the effective zero page address of the operand is computed by adding the contents of the X or Y register to the zero page base address contained in the second byte of the instruction. All zero page indexed instructions are two-byte instructions (one byte less than their absolute-indexed counterparts). Attaching a "X" or a "Y" to the address specifies zero page indexed operands. For example,

```
        LDA        $33，X
```

If the X register contains $05, the instruction loads the contents of location $0038 (i.e., $0033 + $05) into the accumulator. Like absolute indexed addressing, zero page indexed addressing offers the potential for list applications. By using zero page indexed addressing, the instruction requires only two memory locations for storage, while absolute indexed addressing instructions require three memory locations for storage.

One important point to stress that is the effective address is restricted to Page 0 (location 0 through $FF). If the addition of the index register produces an address larger than $FF, the 6502 will disregard any carry out of the low-order byte. In the preceding example, X is restricted to values of $CC (decimal 204) or less; X = $CC will produce an effective address of $FF, while X = $CD will produce a "wrap-around" address of $00.

Zero-page Indexed Indirect Addressing

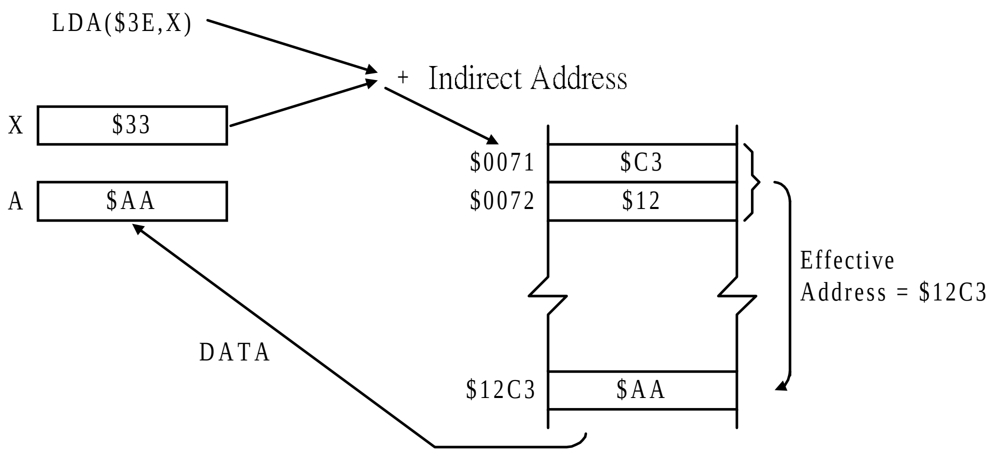


Fig. 1-2-4 Zero-page indexed indirect addressing

Zero-page Indexed indirect addressing is a combination of zero-page indexed addressing and indirect addressing modes. Remember that zero-page indexed addressing gets the effective address from adding an index register displacement to a base address contained in the instruction. In indirect addressing, the operand contained in the instruction is the first address of two memory locations that contain the address of the data, rather than the data itself. These two concepts can be combined. Using zero-page indexed indirect addressing, a displacement in the X register is added to the zero page operand in the instruction, to produce an indirect zero page address. Treating the new indirect zero page address as indirect address to fetch the low address byte of effective address. The effective absolute address is contained in the memory location addressed by the computed indirect address and the next consecutive memory location (high-address byte).

Since the operand address is a zero page address, so all indexed indirect instructions occupy only two bytes in memory. The charming of "zero-page indexed indirect addressing" is that the whole memory space of the 6502 (65,536 bytes) can be accessed with a two-byte instruction since the effective address is a 16-bit absolute address. However, all the zero-page indexed indirect addressing instructions need six cycles to execute, three more than the zero page form of the same instruction and two more than the absolute form. Zero-page indexed indirect operands are in the (aa, X) form. For example,

```
        LDA        ($3E，X)
```

If the X register contains $33, the instruction causes the 6502 to compute an address of $71 ($3E + 33) and fetch the effective memory address from zero page location $71 (low address byte) and $72 (high address byte). If memory location $71 contains $C3 and location $72 contains $12, the contents of location $12C3 will be loaded into the accumulator. As you can see in Fig. 1-2-4.

Zero-page Indirect Indexed Addressing

Zero-page indirect indexed addressing combines the same two addressing modes as "zero-page indexed indirect addressing", but applies them in reverse order. We call the zero-page indexed indirect addressing as "pre-indexing", and zero-page indirect indexed addressing as "post-indexing". Because in zero-page indirect indexed addressing mode, the index is added to the 16-bit memory address after the indirect addressing is performed.

Zero-page indirect indexed operands are of the form (aa), Y. For instance,

```
        LDA        ($3E)，Y
```

If the Y register contains $33, the instruction fetch its base address from zero page locations $3E and $3F. And, if location $3E contains $C3 and location $3F contains $12, the base address of the data table is $12C3. The value of location $12F6 (i.e., $12C3 + $33) will be loaded into the accumulator. Fig. 1-2-5 illustrates the example.

Zero-page indirect indexed addressing is used for accessing a certain known element in one of a number of liked-structured data tables. For instance, this mode might be used in an instruction sequence (like subroutine) that is shared by several users. Before using the zero-page indirect indexed instruction, the users calling program stores a unique base address into the zero page operand location and the consecutive location (like $3E/$3F in above example), so that the instruction can use the correct data table.

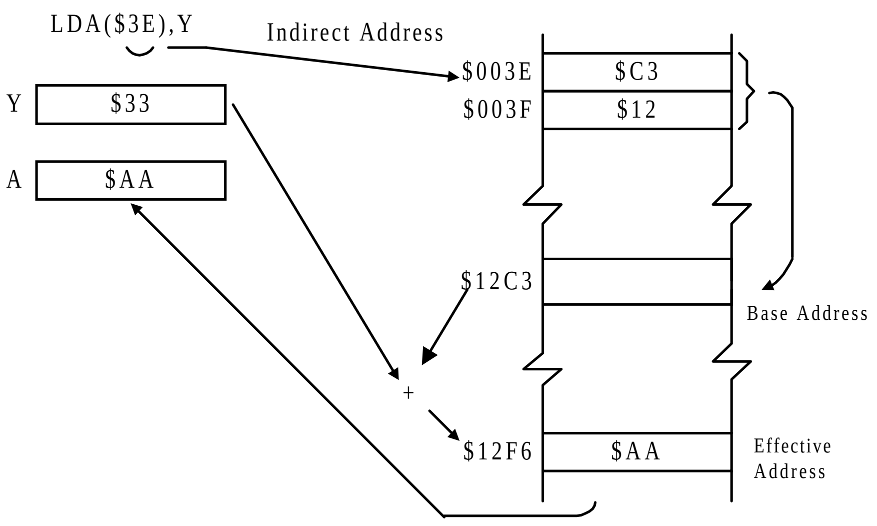


Fig. 1-2-5 Zero-page indirect indexed addressing

Relative Addressing

In relative addressing mode, the effective address is specified relative to the address of the next instruction to be executed. That is, adding the current value of the program counter a positive or negative displacement gets the effective address.

Adding a positive displacement will address a location following the current instruction (i.e., higher in memory), and a negative displacement will address a location preceding the current instruction (lower in memory). Relative addressing is used only by 8 branch instructions. Branch instruction makes program control to transfer forward or backward if a certain condition is met; otherwise, execution proceeds to the next sequential instruction. That is, we usually call them as "conditional jump instructions". For example,

```
        BCC     NEXT
                LDA     #$33
```

The BCC instruction (Branch on Carry Clear) will cause CPU to branch to the instruction at label "NEXT" if the carry bit is clear (reset). If the carry bit is set, the branch will not be performed and CPU executes the LDA instruction.

All branch instructions occupy two bytes in memory, the first byte is op code and the second byte contains the displacement. The displacement is limited to the range of +127 bytes (forward), to –128 (backward) from the branch instruction because the displacement is just 8 bits long.

Accumulator Addressing

The 6502 microprocessor has four instructions that allow shifting or rotating the contents of the accumulator or a memory location one bit position to the right or to the left. If an "A" operand is specified in these instructions, the CPU shifts or rotates the accumulator rather than the memory. Accumulator addressing is an implied type of addressing and it occupies only one byte in memory.

```
        ASL     A
```

That's an example of accumulator addressing. The instruction shifts the contents of accumulator to the left by one bit position.

The 13 addressing modes of the 6502 have been described completely. The Table 1-2-8 summarizes the addressing modes. In this table, “a” represents a hexadecimal address number.

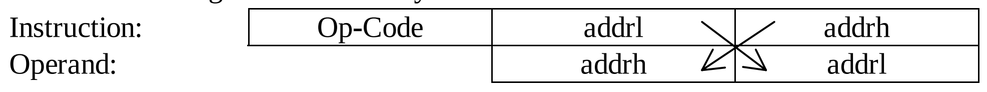
Table 1-2-8 Addressing modes of 6502

Addressing Modes	Operands types
Immediate	#aa
Absolute	aaaa
Zero Page	aa
Implied	
Indirect Absolute	(aaaa)
Absolute Indexed，X	aaaa，X or aaaa X
Absolute Indexed，Y	aaaa，Y or aaaa Y
Zero Page Indexed，X	aa，X or aa X
Zero Page Indexed，Y	aa，Y or aa Y
Zero Page Indexed Indirect	(aa，X) or (aa X)
Zero Page Indirect Indexed	(aa),Y or (aa) Y
Relative	aa or aaaa
Accumulator	A

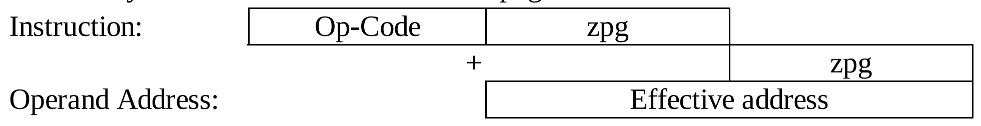
1-2-13 6502 Addressing Modes Summary

Immediate Addressing-# —The operand is the second byte of the instruction.

Absolute-a —With absolute addressing the 2nd and 3rd bytes of the instruction form the 16-bit address.



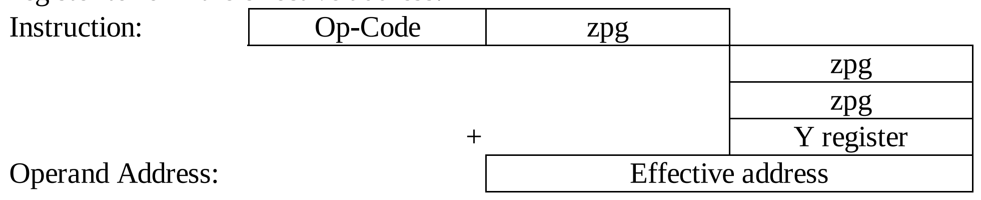
Zero Page-zpg —The second byte of the instruction is the zero page address.



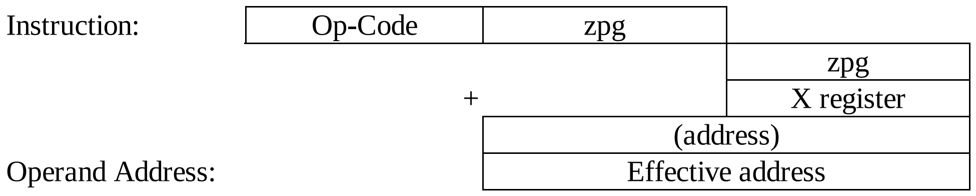
Accumulator-A —It’s a single byte instruction. The operand is the Accumulator.

Implied-i —Implied addressing uses a single byte instruction. The operand is implicitly defined by the instruction.

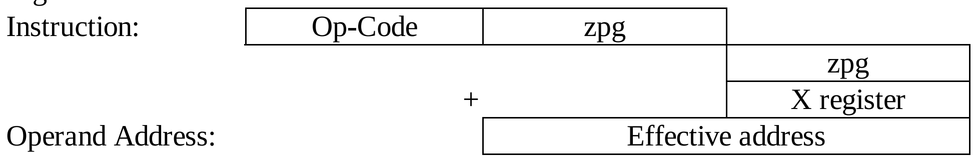
Zero Page Indirect Indexed-(zpg),y—This addressing mode is often referred to as indirect, Y. The second byte of the instruction is the zero page address and the contents of the zero page location are added to the Y index register to form the effective address.



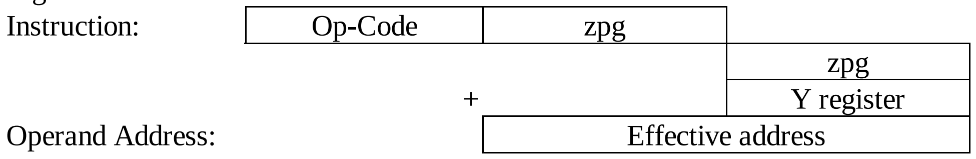
Zero Page Indexed Indirect-(zpg,x)—This addressing mode is often referred to as indirect, X. The second byte of the instruction is the zero page address and is added to the X index register. The result points to the 16-bit effective address.



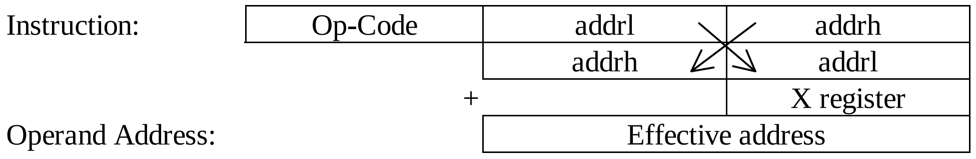
Zero Page Indexed with X-zpg,x—The second byte of the instruction is the zero page address and is added to the X index register to form the 16-bit effective address.



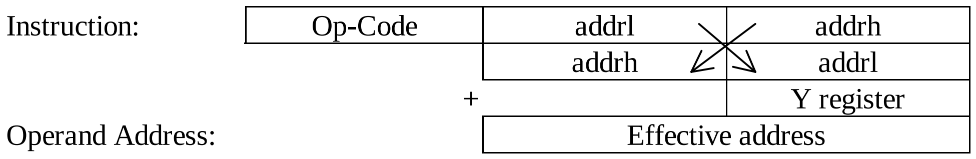
Zero Page Indexed with Y-zpg,y—The second byte of the instruction is the zero page address and is added to the Y index register to form the 16-bit effective address.



Absolute Indexed with X-a,x—The 2nd and 3rd bytes of the instruction are added to the X register to form the 16-bit effective address.



Absolute Indexed with Y-a,y—The 2nd and 3rd bytes of the instruction are added to the Y register to form the 16-bit effective address.



Program Counter Relative-r—This addressing mode, referred to as relative addressing, is used only with the Branch instructions. If the condition being checked is met, the second byte (displacement) of the instruction is added to the Program Counter, which has been updated to point to the Op Code of the next instruction. The offset is a signed 8-bit quantity in the range from –128 to 127.

Indirect Absolute-(a)—It's used only by Jump instruction. The 2nd and 3rd bytes of the instruction form an address pointer. The Program Counter is loaded with the first and second bytes at this pointer.

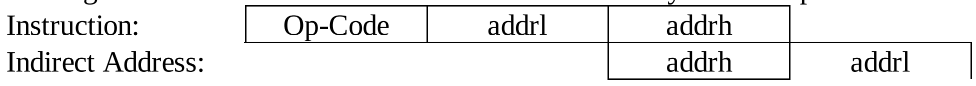


Table. 1-2-9 Addressing Mode Summary

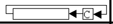
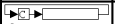
Addressing Mode	Instruction Cycles	Occupies Memory Bytes
1. Immediate	2	2
2. Absolute	4 (3)	3
3. Zero Page	3 (3)	2
4. Accumulator	2	1
5. Implied	2	1
6. Zero Page Indirect Indexed (d),y	5 (1)	2
7. Zero Page Indexed Indirect (d,x)	6	2
8. Zero Page, X	4 (3)	2
9. Zero Page, Y	4	2
10. Absolute, X	4 (1, 3)	3
11. Absolute, Y	4 (1)	3
12. Relative	2 (2)	2
13. Indirect Absolute (Jump)	5	3

NOTES: (1) Page boundary, add 1 cycle if page boundary is crossed when forming address.
(2) Branch taken; add 1 cycle if branch is taken.
(3) Read-Modify-Write, add 2 cycles.

1-2-14 Summary of Instruction Set

Table 1-2-10 is a summary table to which you will be often referring.

Table 1-2-10 6502 Instructions set summary

Instructions		Immediate	Absolute	Zero-page	AC	Implied	(Indirect), X	(Indirect), Y	Zero-page, X	Absolute, X	Absolute, Y	Relative	Indirect	Zero-page, Y		
Mne.	Operation	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	N V 1 B D I Z C	Mne.
ADC	A+M+C→ A	69 2 2	6D 4 3	65 3 2			61 6 2	71 5 2	75 4 2	7D 4 3	79 4 3				N V . . . . Z C	ADC
AND	A^M→A	29 2 2	2D 4 3	25 3 2			21 6 2	31 5 2	35 4 2	3D 4 3	39 4 3				N . . . . . Z .	AND
ASL	C←b_7 b_0← 0		0E 6 3	06 5 2	0A 2 1				16 6 2	1E 7 3					N . . . . . Z C	ASL
BCC	Jump if C=0											9D 2 2			. . . .	BCC
BCS	Jump if C=1											B0 2 2			. . . .	BCS
BEQ	Jump if Z=1											F0 2 2			. . . .	BEQ
BIT	A^M		2C 4 3	24 3 2											M_7M_6 . . . Z . .	BIT
BMI	Jump if Z=1											30 2 2			. . . .	BMI
BNE	Jump if Z=0											DO 2 2			. . . .	BNE
BPL	Jump if N=0											10 2 2			. . . .	BPL
BRK	Interrupt					00 7 1									. . . 1 0 1 . .	BRK
BVC	Jump if V=0														. . . .	BVC
BVS	Jump if V=1											50 2 2			. . . .	BVS
CLC	0 → C					18 2 1						70 2 2			 0	CLC
CLD	0 → D					D8 2 1									. . . . 0 . . .	CLD
CLI	0 → I					58 2 1									. . . . . 0 . .	CLI
CLV	0 → V					B8 2 1									. 0	CLV
CMP	A - M	C9 2 2	CD 4 3	C5 3 2			C1 6 2	D1 5 2	D5 4 2	DD 4 3	D9 4 3				N . . . . . Z C	CMP
CPX	X - M	EO 2 2	EC 4 3	E4 3 2											N . . . . . Z C	CPX
CPY	Y - M	CO 2 2	CC 4 3	C4 3 2											N . . . . . Z C	CPY
DEC	M - 1 → M		CE 6 3	C6 5 2					D6 6 2	DE 7 3					N . . . . . Z .	DEC
DEX	X - 1 → X					CA 2 1									N . . . . . Z .	DEX
DEY	Y - 1 → Y					88 2 1									N . . . . . Z .	DEY
EOR	A (+) M → A	49 2 2	4D 4 3	45 3 2			41 6 2	51 5 2	55 4 2	5D 4 3	59 4 3				N . . . . . Z .	EOR
INC	M + 1 → A1		EE 6 3	E6 5 2					F6 6 2	FE 7 3					N . . . . . Z .	INC
INX	X + 1 → X					E8 2 1									N . . . . . Z .	INX
INY	Y + 1 → Y					C8 2 1									N . . . . . Z .	INY
JMP	Jump to location		4C 3 3										6C 5 3		. . . .	JMP
JSR	Jump to program		20 6 3												N . . . . . Z .	JSR
LDA	M → A	A9 2 2	AD 4 3	A5 3 2			A1 6 2	B1 5 2	B5 4 2	BD 4 3	B9 4 3				N . . . . . Z .	LDA
LDX	M → X	A2 2 2	AE 4 3	A6 3 2							BE 4 3			B6 4 2	N . . . . . Z .	LDX
LDY	M → Y	A0 2 2	AC 4 3	A4 3 2					B4 4 2	BC 4 3					N . . . . . Z .	LDY
LSR	0 → b_7 b_0 → C		4E 6 3	A6 5 2	4A 2 1				56 6 2	5E 7 3					0 . . . . . Z C	LSR
NOP	No operation					EA 2 1									. . . .	NOP
ORA	A v M → A	09 2 2	0D 4 3	05 3 2			01 6 2	11 5 3	15 4 2	1D 4 3	19 4 3				N . . . . . Z .	ORA
PHA	A → Ms S -1→S					48 3 1									. . . .	PHA
PHP	P → Ms S- 1→S					08 3 1									. . . .	PHP
PLA	S+1→S Ms → A					68 4 1									N . . . . . Z .	PLA
PLP	S+1→S Ms→ P					28 4 1									N V . 1 D I Z C	PLP
ROL			2E 6 3	26 5 2	2A 2 1				26 6 2	3E 7 3					N . . . . . Z C	ROL
ROR			6E 6 3	66 5 2	6A 2 1				76 6 2	7E 7 3					N . . . . . Z C	ROR
RTI	Interrupt return					40 6 1									N V . 1 D I Z C	RTI
RTS	Subroutine return					60 6 1									. . . .	RTS
SBC	A-M-$\overline{C}$→ A	E9 2 2	ED 4 3	E5 3 2			E1 6 2	F1 5 2	F5 4 2	FD 4 3	F9 4 3				N V . . . . Z C	SBC
SEC	1 → C					38 2 1									 1	SEC
SED	1 → D					F8 2 1									. . . . 1 . . .	SED
SEI	1 → I					78 2 1									. . . . . 1 . .	SEI
STA	A → M		8D 4 3	85 3 2			81 6 2	91 6 2	95 4 2	9D 5 3	99 5 3				. . . .	STA
STX	X → M		8E 4 3	86 3 2										96 4 2	. . . .	STX
STY	Y → M		8C 4 3	84 3 2					94 4 2						. . . .	STY
TAX	A → X					AA 2 1									N . . . . . Z .	TAX
TAY	A → Y					A8 2 1									N . . . . . Z .	TAY
TSX	S → X					BA 2 1									N . . . . . Z .	TSX
TXA	X → A					8A 2 1									N . . . . . Z .	TXA
TXS	X → S					9A 2 1									. . . .	TXS
TYA	Y → A					98 2 1									N . . . . . Z .	TYA

X	Index X	+	ADD	M_7	Bit7 of the contents of memory location
Y	Index Y	-	SUBTRACT	M_6	Bit6 of the contents of memory location
A	Accumulator (AC)	^	AND	n	Cycles No.
M	Memory of effective location	v	OR	#	Bytes No.
Ms	Memory for Stack Register	(+)	EOR		

Chapter 1-3 Interrupts and Reset

Interrupts: There are two types of interrupts, maskable (interrupts that can be temporarily ignored) and nonmaskable (interrupts that require immediate attention).

Reset: A reset is the operation by which a system is initialized to some known state. All microprocessors will be reset when the system power is applied. Further, most microprocessors are designed to allow the internal registers to be initialized at other times by an external signal — $\overline{\text{RES}}$ (Reset). We have just defined two separate conditions, reset at power-up time and restart at all other times.

Interrupts and reset are functionally similar because they use vector pointers to determine the memory address from which the next instruction will be fetched. The 6502 microprocessor uses locations $FFFA through $FFFF to hold the whole vector pointers, and, they contain three sections—

① Locations $FFFA and $FFFB hold the vector pointers for the nonmaskable interrupt;
② Locations $FFFC and $FFFD hold the vector pointers for the reset, and
③ Locations $FFFE and $FFFF hold the vector pointers for the maskable interrupt request.

1-3-1 Interrupts

Maskable interrupts ($\overline{\text{IRQ}}$) are external signals which attempt to temporarily suspend the program that is being executed by the microprocessor, and cause program control to be transferred to a subroutine that is designed to serve that particular interrupt. Peripheral devices use interrupts to "inform" the microprocessor that they have data to be input, or that they need data from the microprocessor. This technique eliminates the need for the microprocessor to waste valuable execution time polling the states of the peripheral devices of the system when none of the devices require servicing.

Non-maskable interrupts ($\overline{\text{NMI}}$) are used to signal some condition that requires the immediate attention of the microprocessor, such as a power failure. Peripheral devices request interrupt service from the 6502 by activating the Interrupt Request ($\overline{\text{IRQ}}$), and the critical situations (such as power failure) force immediate service from the 6502 by activating the Non-maskable interrupts ($\overline{\text{NMI}}$). The "bar" over the IRQ and NMI signal names indicates that they are active low.

1-3-2 Interrupt Request (IRQ)

How can the 6502 "know" which peripheral devices in the system need service? Instinctively, you can design the main program that can stop processing data and poll every device in the system in every short period to check whether any device requires servicing. Obviously, this is very inefficient, and processor will waste much time in polling. There is another way you can use; the 6502 has an $\overline{\text{IRQ}}$ input that permits external devices to request servicing from the microprocessor. The $\overline{\text{IRQ}}$ functions as the telephone bell and indicates to the 6502 microprocessor that some device in the system is "calling". Also, like a telephone bell, the $\overline{\text{IRQ}}$ can be ignored until the 6502 is prepared to respond to it.

Instructions	Codes	Cycles / Bytes	Status flags affected
SEI	78	2/1	I=1
CLI	58	2/1	I=0

Interrupt Control Instructions

The IRQ Disable bit (I) in the processor status register determines whether or not the 6502 will respond to an interrupt request on the $\overline{\text{IRQ}}$ line. The two instructions:

Instruction	**Description**
SEI	Set Interrupt Disable Bit
CLI	Clear Interrupt Disable Bit

The SEI instruction sets the Interrupt Disable bit (I), which will cause the 6502 to ignore all subsequent $\overline{\text{IRQ}}$ interrupt requests. The CLI instruction clears the Interrupt Disable bit (I), which will cause an external interrupt request to be serviced as soon as it is sensed by the 6502. The 6502 initially set the Interrupt request Disable bit (I) when power is applied.

Responding to an IRQ

If the IRQ Disable bit (I) is cleared and some external device activates the $\overline{\text{IRQ}}$ signal (pulls it low), after the 6502 finishes executing the current instruction completely, then it will automatically initiate an eight-cycle interrupt sequence. During this sequence, the 6502 pushes three bytes of “return” information onto the stack (the high and low bytes of the program counter and the contents of the processor status register). It then loads the contents of the dedicated IRQ vector low ($FFFE) and high ($FFFF) into the program counter. In the meantime, the 6502 also sets the IRQ Disable bit (I) to temporarily “lock out” subsequent interrupt requests.

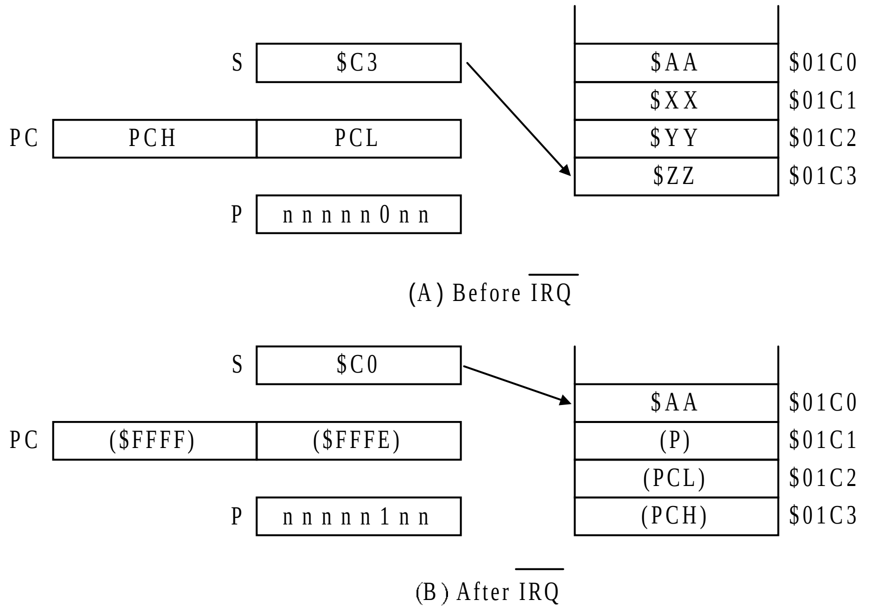


Fig. 1-3-1 6502 responds to an IRQ

Fig 1-3-1 summarizes how the 6502 responds to an IRQ, by showing both “before” and ‘after” diagrams of the Stack Pointer (S), program counter (PC), the processor status register (P), and four locations in the stack (here, assumed $01C0 to $01C3).

In the diagram of Fig. 1-3-1A, the Stack Pointer is pointing to the next free stack location (assumed to be $01C3 for this example). The program counter contains the address of the next instruction in the main program (high-address is PCH, low-address is PCL), and the “I” bit in the processor status register is cleared to zero. In Fig. 1-3-1B, it is shown that following $\overline{\text{IRQ}}$, the program counter and processor status register are now on the stack, and the Stack Pointer is pointing to location $01C0. Furthermore, the low-order and high-order bytes (PCL and PCH) of the program counter now contain the contents of memory locations $FFFE and $FFFF, respectively, and the I bit has been set to a 1 in the processor status register. You can see that, the (I) bit of the processor status register is the only one bit that was altered by this operation.

After responding to the $\overline{\text{IRQ}}$, the program counter contains the starting address of a program designed to serve IRQ-generated interrupts. This program is called an **interrupt service routine** (ISR). The interrupt service routine must perform two functions:

① It must identify the device that generated the interrupt request (if there is more than one device in system).
② It must perform the operation required by that device.

Generally, the ISR identifies that device by polling the status register of each device in the system, to find out which device has its interrupt request bit set. Once the interrupting device is identified the interrupt service routine (ISR) must fetch a new address - the starting address of a specified interrupt routine for that interrupting device.

The 6502-compatible devices maintain their interrupt request bits in either bit 6 or 7 of a register, so that the polling sequence can be performed with a series of BIT and branch instructions. Example 1-3-1 shows a polling sequence for a system that contains four devices. Devices 1,2, and 4 can generate only one interrupt request, and will indicate this request in bit7 of their respective status registers (locations SDEV1, SDEV2, and SDEV4). Device 3 can generate two separate interrupt requests, and indicate these requests in bits6 and 7 of its status register (location SDEV3). The interrupt status of Devices 1, 2, and 3 in the system is queried with a BIT instruction, which loads the state of bit6 and 7 into the **processor status register’s** Overflow (V) and Negative (N) flag, respectively. For Devices 1 and 2, a BMI instruction determines whether or not the device has an active interrupt request. Device 3 requires two branch instructions (BMI and BVS) because this device is capable of generating either of two separate interrupt requests. Device 4 requires no query, since it must have generated the interrupt request if Device 1, 2, or 3 did not.

You should have noted that the polling sequence in Example 1-3-1 assigns priorities to the devices in the system; Device 1 has the highest priority, Device 4 has the lowest priority. Although it seems that the 6502 microprocessor needs a large amount of time to get to the Device 4 interrupt service routine, due to a number of instructions that are executed before a Device 4 interrupt request can be serviced. However, the polling takes only six cycles for Devices 1 and 2, and eight cycles for Device 3. Therefore, the 6502 needs only 20 cycles (~ 5 *u*sec) before it can service Device 4.

Example 1-3-1 Interrupt Polling Sequence

```
            :
            BIT     SDEV1             ; Interrupt request from Device 1?
            BMI     JISR1             ; If so, branch to JISR1.
            BIT     SDEV2             ; Interrupt request from Device 2?
            BMI     JISR2             ; If so, branch to JISR2.
            BIT     SDEV3             ; Interrupt request from Device 3?
            BMI     JISR3A            ; If so, branch to JISR3A
            BVS     JISR3B            ;    or JISR3B.
            JMP     ISR4              ; Go service Device 4 interrupt.
JISR1               JMP     ISR1      ; Go service Device 1 interrupt.
JISR2               JMP     ISR2      ; Go service Device 2 interrupt.
JISR3A      JMP     ISR3A             ; Go service Device 3 interrupt "A".
JISR3B      JMP     ISR3B             ; Go service Device 3 interrupt "B".
            :
            :
```

Besides the instructions that actually transfer information between the interrupting peripheral device and the 6502, the most interrupt service routines (ISR) begin with instructions that save, on the stack, the current values of registers that will be altered by the service routine. Certainly, the end of the subroutine must have complementary instructions that pull those register values off the stack.

Furthermore, most interrupt service routines also include a CLI (Clear Interrupt Disable Bit) instruction, to allow other higher-priority interrupt requests to be served. The location of CLI in the interrupt service routine will vary with the priority of the device being serviced. In the lowest-priority device ISR, CLI may follow the instructions that save register values on the stack. Conversely, the interrupt service routine for the highest-priority device may not even include a CLI instruction, and will allow interrupt requests to be enabled only in return from the interrupt service routine. The instruction that performs the return from an interrupt service routine is a special instruction called RTI. The RTI instruction must be the final instruction to be executed in every interrupt service routine.

Instruction	**Description**
RTI	Return from Interrupt

This RTI instruction gets the processor status register and the program counter to be reinitialized with their values before interrupt from the stack. Execution of the RTI instruction will automatically (re-) enable $\overline{\text{IRQ}}$ interrupt requests, since the Interrupt Disable bit (I) of the processor status register was clear when this register was pushed onto the stack.

The RTI instruction is quite similar to the RTS (ReTurn from Subroutine). Both instructions return from a subroutine to the program from where the subroutine was called. For RTI, the call was made automatically, by an externally generated interrupt requests.

For RTS, the call was made under software control by a JSR (Jump to SubRoutine) instruction. The RTS and RTI instructions are implied-address instructions that occupy only one byte in memory and require six cycles to be executed.

The only difference between RTS and RTI is that RTS pulls two bytes of information from the stack, whereas, RTI pulls three bytes of information from the stack (besides the low and high bytes of the program counter, like RTS, also including the processor status register). In fact, if you had a subroutine that needed to save the processor status register value before subroutine, you could use a PHP (Push Processor Status on Stack) as the first instruction in the subroutine, and use an RTI, rather than an RTS, as the return instruction for the subroutine.

1-3-3 Nonmaskable Interrupt (NMI)

The other interrupt of the 6502, Nonmaskable Interrupt ($\overline{\text{NMI}}$), provides interrupts for high-priority devices and events (such as a power failure) that cannot afford to wait during the time that the interrupt requests are disabled. Unlike $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$-generated interrupts cannot be masked out or disabled; the 6502 will begin processing a $\overline{\text{NMI}}$ interrupt on completion of the currently executing instruction.

$\overline{\text{NMI}}$ interrupt always has priority over $\overline{\text{IRQ}}$ interrupt that means if an interrupt request and a nonmaskable interrupt occur simultaneously, the 6502 will process the nonmaskable interrupt. The 6502 performs an eight-cycle sequence for either type but, for the $\overline{\text{NMI}}$, the vector pointer is fetched from locations $FFFA (low-address byte) and $FFFB (high-address byte), rather than from locations $FFFE and $FFFF (those for $\overline{\text{IRQ}}$). In fact, Fig. 1-3-1 also illustrates how the 6502 responds to $\overline{\text{NMI}}$ as well, except that for the $\overline{\text{NMI}}$, the post-interrupt value of the program counter would be ($FFFB) and ($FFFA).

Since the $\overline{\text{NMI}}$ line is dedicated to serve high-priority events, it is important to know just how fast the 6502 can be expected to respond to the interrupt. Let us find out by taking a "worst case". The worst case occurs if the $\overline{\text{NMI}}$ is activated just as the 6502 has fetched the op code of one of its longest instructions – a seven-cycle instruction such as INC (Increment Memory) or DEC (Decrement Memory) with absolute indexed addressing. The $\overline{\text{NMI}}$ interrupt will not be accepted until completion of this instruction six cycles (**6**) later, at which time the 6502 will initiate an eight-cycle interrupt sequence. In the eighth cycle of this sequence (**8 – 1 = 7**), the 6502 will fetch the op-code byte of the first instruction in the interrupt service routine (ISR). This instruction can be either the input (LDA) or output (STA) instruction that transfers data from or to the peripheral device, in which case, it will be executed in no less than four cycles (**4**). Adding these execution cycle values (**6 + 7 + 4 = 17**), we see that with an $\overline{\text{NMI}}$ interrupt, data can be transferred to or from a peripheral device in no more than 17 cycles (~ 4.25 *u*sec).

1-3-4 Break

The Break (BRK) instruction is usually inserted at one or more points in a developed program wherever you would like to halt the 6502 and check what your program has done to that point. When BRK is executed, the 6502 sets the BRK Command flag (B) in the processor status register (P), increments the program counter (PC) by one, and then pushes three bytes of information onto the stack (the high and low bytes of PC and the contents of P register).

At this point, the 6502 loads the contents of the IRQ vector low ($FFFE) and high ($FFFF) into the program counter.

Instruction	Codes	Cycles / Bytes	Status flags affected
BRK	00	7/1	B=1， I=1

Since the BRK instruction causes a vector to the same location as an $\overline{\text{IRQ}}$ -generated interrupt, the IRQ interrupt service routine must include instructions that determine whether the routine is being executed due to an external $\overline{\text{IRQ}}$ or due to a BRK instruction. The simplest way to make this determination is to check the state of the BRK Command Bit (B) in the processor status register value on the stack. If the B bit is a 1, the routine is being executed due to a BRK instruction; otherwise, the routine is being executed due to $\overline{\text{IRQ}}$.

Since BRK instructions are temporary instructions during software development, they must overlay an existing instruction. BRK is a one-byte instruction, so it will just overlay the op code byte; if a multi-byte instruction is overlaid with BRK, its operand byte(s) will remain intact. It will be no problem for two-byte instruction because the program counter values on the stack (the return address) addresses the second byte after the BRK instruction. If you want to overlay the op code of a three-byte instruction with a BRK instruction, the interrupt service routine (ISR) will return to the third byte of the instruction (the high-order byte of the operand), and the 6502 will attempt to execute the byte as an instruction op-code. In this situation, you must code two temporary instructions into the program— BRK to overlay the op code and NOP instruction to overlay the third byte to prevent the 6502 from executing the third byte as an instruction op-code.

1-3-5 Reset

The contents of the program counter determine the memory location from where the 6502 will fetch the next instruction to be executed. How does the program counter receive its initial contents when power is applied to the system? The signal that initializes the program counter and, thereby, initiates the 6502 operation, is an externally generated signal called $\overline{\text{RES}}$ (Reset). This signal is used to reset or start the microprocessor from a power-down condition. Information can be neither written to nor read from it, and the contents of the internal registers will be undefined when $\overline{\text{RES}}$ is held low (to ground), the 6502 is in a disabled state.

After $\overline{\text{RES}}$ has reached to the high level, the 6502 will immediately initiate a six-cycle start sequence. During this sequence, the 6502 sets the IRQ Disable bit (I) of the processor status register to “lock out” external interrupts while the microprocessor is being initialized, and loads the contents of memory locations $FFFC and $FFFD into the low-order and high-order bytes, respectively, of the program counter. These two locations, $FFFC and $FFFD, must contain the address of the first instruction to be executed by the 6502. This is the starting address of an initialization program. (Since the contents of these locations must be preserved while the power is off, they must reside in ROM. In fact, the initialization programs that these locations address must also reside in ROM.)

Example 1-3-2 A 6502 Reset Program

```
 ；This program initializes a 6502-based microcomputer system
 ；following power-on RESET
 ；
RESET        LDX    #$FF              ；Initialize Stack Point to $01FF。
             TXS
             :
             :               ;Configure I/O devices in the system
             :

             LDA    #00               ；Initialize registers to zero
             TAX                      ；
             TAY
             CLD                      ；Clear decimal mode
             CLC                      ；Clear Carry
             CLI                      ；Enable interrupt
             JMP    USERPG            ；Execute user program
```

Example 1-3-2 shows a typical sequence of instructions in a reset program:

① This program begins with initializing the Stack Pointer. The Stack Pointer is normally initialized to point to location $01FF, as it is here, because that is the starting location of the stack, but there is no reason why a different reset program could not initialize the Stack Pointer to some other page one location.

(1-1) Why is the Stack Pointer initialization assigned the highest priority in this program? It is assigned this priority so that an immediate non-maskable interrupt ($\overline{\text{NMI}}$), such as a power failure, can be properly serviced.

(1-2) After initializing the Stack Pointer, the program should initialize the registers of the various peripheral I/O devices in the system. The I/O initialization instructions are not shown because they depend on the configuration of the system.

② The next three instructions (LDA #00, TAX, and TAY) load zeroes into the accumulator, the X register, and the Y register, respectively.

③ The next two instructions, CLD and CLC, are arbitrary; some reset programs may set one or both of these bits, other reset programs may clear one and set the other.

④ With the stack and all registers now established, the next instruction, CLI, enables $\overline{\text{IRQ}}$ interrupt requests. (These interrupts were disabled as part of the start sequence of the 6502 microprocessor mentioned previously.) The final instruction in the reset program executes a jump to the program of the user, which is assigned the label USERPG in this example.

Sometimes, you will like to be able to initialize the system to a known state rather than power-up. These cases include instances in which a program has somehow entered an endless loop, or the malfunction of some device causes the system to "hang up." As an alternative to turning the power off, which will cause information in R/W memory to be lost, most microcomputer systems allow the $\overline{\text{RES}}$ signal to be activated by a pushbutton or by some other means.

Chapter 1-4 Subroutines

If a specific series of instructions must be performed at more than one place in the program, the entire sequence of instructions must be repeated in the program at each one of these places. Obviously, repeating a sequence of instructions in a program at many places would be frustrating and time-consuming for the programmer, for eliminating this needless repetition you can define the repeated code as a **subroutine**. A subroutine is a sequence of instructions that is written just once, but which can be executed as needed, at any point in the program.

The process of transferring control from a program to a subroutine is defined as **calling**. Once a subroutine called, the 6502 microprocessor executes the sequence of instructions contained in the subroutine and returns control to the calling program.

1-4-1 SUBROUTINE INSTRUCTIONS

There are three functions must be performed during a subroutine called:

① They must include some provision for saving the contents of the program counter. Once the subroutine has been executed, this address will be used to return to the program. This address is often called a **return address**.

② They must cause the microprocessor to begin executing the subroutine.

③ They must return to the calling program and continue executing the program at this point, using the return address.

Two 6502 instructions, Jump to Subroutine (JSR) and Return from Subroutine (RTS), can perform these three functions.

Instruction	Codes	Cycles / Bytes	Status flags affected
JSR abs	20aabb	6/3	
RTS	60	6/1	
RTI	40	6/1	N, V, B=1, D, I, Z, C

Jump to Subroutine (JSR)

The JSR instruction performs the “storing-return-address” and “begin-executing” functions (those are requirements ① and ②, above). The stored return-address is the address of the third byte of the JSR instruction. (Clearly, this is not the address to which the return will ultimately be made, but this address will be incremented upon return to provide the proper destination. This will be discussed in more detail when we examine the RTS instruction.) Where is the return address stored? It is stored on the stack, which means that the JSR instruction operates like the PHA and PHP push instruction. However, the PHA and PHP instructions just push one byte of data onto the stack (the accumulator contents and the processor status register contents, respectively), and the JSR instruction pushes two bytes onto the stack — the two-byte address in the program counter. After storing the program counter on the stack, the JSR instruction loads the program counter with the absolute address contained in its second and third bytes, which transfers control to the starting address of the subroutine.

The JSR instruction occupies three bytes in memory— one op-code byte and two subroutine address bytes (the low-address byte and high-address byte, respectively). The subroutine address is an absolute address in memory or the label of an absolute address, for assembler source code. The 6502 needs six cycles to execute the JSR instruction because of the stack operations involved.

Now, consider a typical jump to Subroutine instruction, JSR $0503, which is located in memory at locations $0201, and $0203. Thus:

Memory location	**Instruction**	
$0201	JSR	(e.g. the op code— 20H)
$0202	$03	
$0203	$05	
:	:	
$0503	(First subroutine instruction)	

Here, using a figure to describe the Stack Pointer (S), Program Counter (PC), and the Stack in memory both before and after the instruction "JSR $0503":

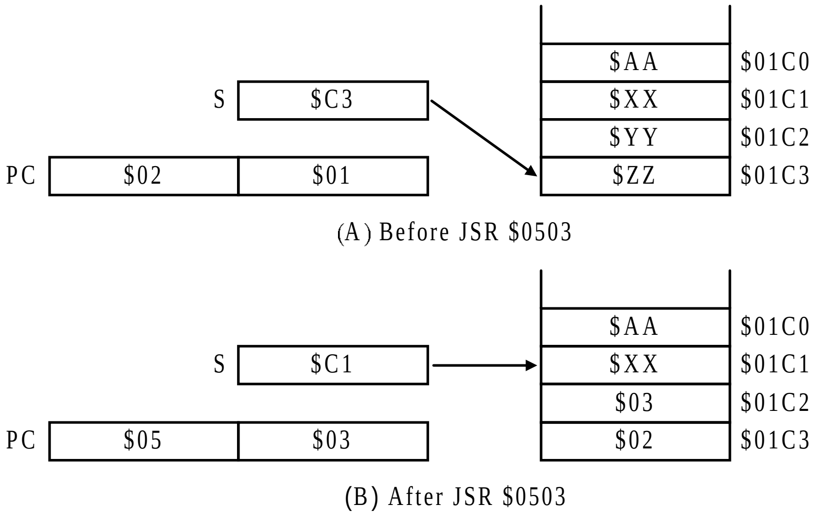


Fig. 1-4-1 The circumstance when a JSR instruction executed

In the diagram of Fig.1-4-1A, the program counter is pointing to the first byte of the JSR instruction ($0201) and the Stack Pointer is pointing to the next free location on the stack (here, assumed $01C3). After the JSR $0503 instruction (Fig. 1-4-1B,) has been executed: ①The program counter contain the starting address of the subroutine— $0503. ②The Stack Pointer is pointing to a new "next stack location" ($01C1). ③And the two top bytes on the stack are the MSBY and LSBY addresses of the third byte of the JSR instruction. As previously mentioned, the 6502 microprocessor must return to the instruction that immediately follows the JSR instruction. For our example, the 6502 must return to the instruction that starts in location $0204.

Return from Subroutine (RTS)

The Return from Subroutine (RTS) instruction causes the 6502 to return from the subroutine to the calling program (the program that contains the JSR instruction).

A RTS instruction is always the last instruction that is executed in a subroutine. The RTS instruction occupies only one byte in memory, takes six cycles to execute, and affects no bits in the processor status register.

The RTS instruction causes the 6502 to continue program execution at an absolute address that is one greater than the address on the last two bytes of the stack. Recall that before loading the subroutine address into the program counter, the JSR instruction pushed two addresses bytes onto the stack. These were the least-significant address byte (LSBY) and the most-significant address byte (MSBY) of the memory location that contains the high-address byte (the third byte) of the JSR instruction — it was address $0203, in our example. To retrieve that address, the RTS instruction increments the Stack Pointer ($C2, in the example), loads the LSBY address into the lower half of the program counter and, then, increments the Stack Pointer again and, then, loads the MSBY address into the upper half of the program counter. The program counter is then incremented so that it addresses the next instruction after the JSR instruction.

There is an implied requirement that when a RTS instruction is executed, the Stack Pointer is addressing the stack location used or established by the JSR instruction. Therefore, if any push operations are performed in the subroutine, there must be an equal number of pull operations before the RTS is executed.

How JSR and RTS are Used Together

Here, taking an example to look over how the JSR and RTS are used together. Example 1-4-1 shows the Jump to Subroutine instruction that was diagrammed in Fig. 1-4-1. The instruction JSR $0503 is being used to call a subroutine that simply doubles the value in the Y register. The JSR instruction starts in location $0201. The subroutine starts in location $0503 and ends in location $0508, with the RTS instruction.

The 6502 microprocessor must return to the location that follows the JSR $0503 instruction. Since the JSR instruction occupies three bytes in memory, the return will be to location $0204. Fig. 1-4-2 shows the configuration of the Stack Pointer (S), the Program Counter (PC), and the memory stack before and after the RTS instruction is executed.

Example 1-4-1 The subroutine call and return sequence

Memory location	Instruction	Operand	Comment
$0201	JSR	$0503	; Subroutine call instruction
$0204	:	:	; Subroutine returns here
:	:	:	
(subroutine)			
$0503	PHA		; Save accumulator on Stack
$0504	TYA		; Transfer Y into accumulator,
$0505	ASL	A	; double it,
$0506	TAY		; transfer result back to Y
$0507	PLA		; Retrieve accumulator
$0508	RTS		; Return

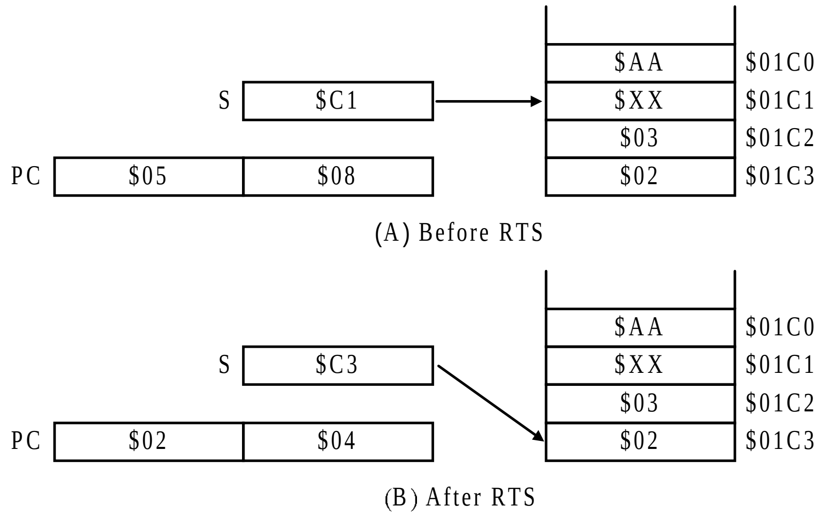


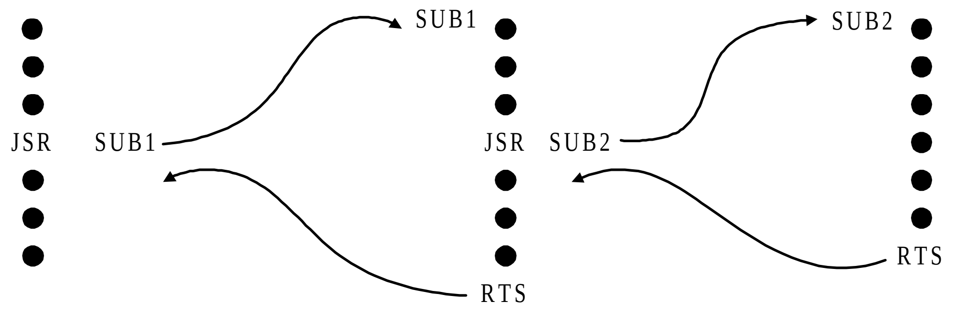
Fig. 1-4-2 RTS pulls an address from Stack

1-4-2 SUBROUTINE NESTING

Of course, a subroutine may include one or more JSR instructions that call other subroutines. The process of calling a subroutine from within a subroutine is called **subroutine nesting**. Example 1-4-2 shows the JSR and RTS instructions for a program in which subroutine SUB2 is called from within subroutine SUB1 (i.e., SUB2 is nested within SUB1).

Nesting is usually described in terms of levels. An application like the one shown in Example 1-4-2, in which the nesting extended only to the JSR to SUB2 (SUB2 did not call another subroutine) is said to have one level of nesting. There is no reason, though, why SUB2 could not have called another subroutine (SUB3), with SUB3 calling SUB4, and so on. Considering that each JSR instruction pushes two address bytes onto the stack, only the capacity of the stack limits the amount of nesting. Since the stack can utilize all 256 bytes of page one in memory, a 6502 program can have up to 127 levels of subroutine nesting. However, very few applications will require nesting even approaching this limit !

Example 1-4-2 Subroutine Nesting



Chapter 1-5 2500AD 6502 Assembler/Linker Quick Reference

This chapter is written for the beginner to learn "2500AD 6502 assembler/linker" rapidly. Here are practical examples to guide readers to use the 6502 Assembler and Linker. Certainly, if you like to study it in detail, you should refer to the user guides published by 2500AD Software Inc.

1-5-1 Assembler

How to use x6502.exe?

2500AD 6502 assembler provides two operation modes, Prompt Mode and Command Line Mode. You will learn how to use the two operation modes from examples presented in each section.

1.1 Prompt Mode

Under DOS prompt (pure DOS or DOS mode of Windows 95), typing "x6502" followed with the entire path to run Assembler. Such as,

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\X6502 (enter)
```

You will see the message as below…

```
6502 Macro Assembler Copyright (C) 1990 by 2500AD Software Inc. Version 5.02a    ①
Listing Destination  (N, T, D, E, L, P, <CR> = N): (enter)                       ②
Generate Cross Reference? (Y/N  <CR> = No): (enter)                              ③

Input Filename:  (enter)                                                         ④
Output Filename: (enter)                                                     ⑤
```

Description:

① Show the version and copyright of the Assembler.
② Require the listing destination to be assigned.
③ Do you want to generate a cross-reference table? If you answer "Yes", then the cross-reference table will be generated and it will be put at the last of the list.
④ Require a source file that wants to be assembled.
⑤ Assign output filename. It'll use the same name of input file if you don't assign (.OBJ).

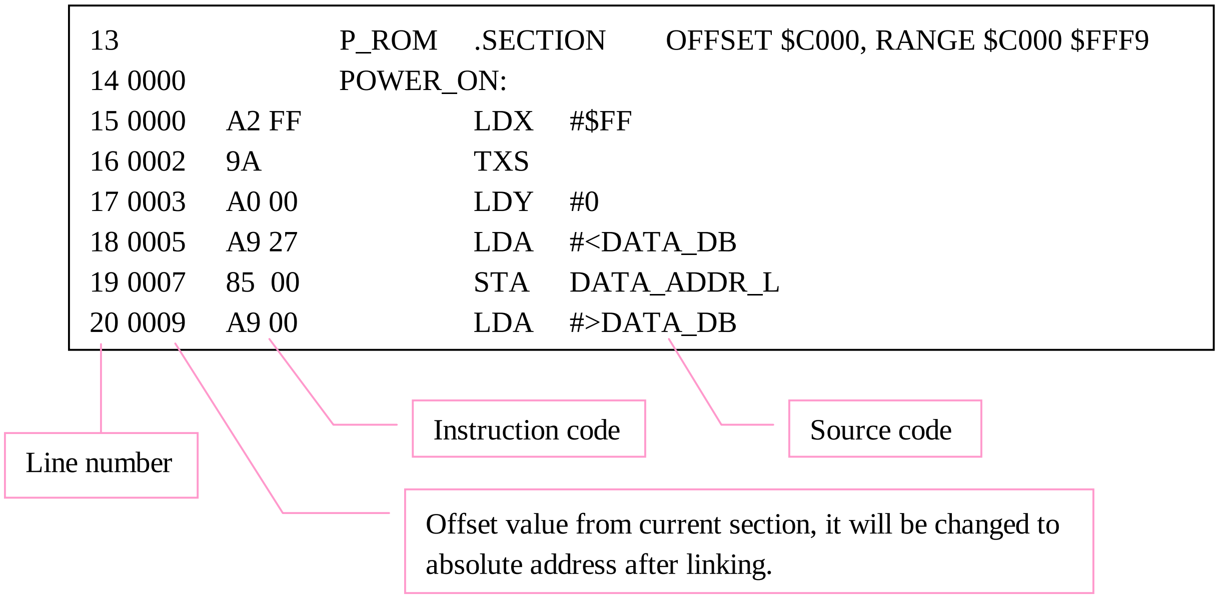
[Example] (generating list file)

```
6502 Macro Assembler Copyright (C) 1990 by 2500AD Software Inc. Version 5.02a
Listing Destination  (N, T, D, E, L, P, <CR> = N): D (enter)
Generate Cross Reference? (Y/N  <CR> = No): (enter)

Input Filename: DEMO1 (enter)
Output Filename: (enter)
```

The result message is shown as below…

```
              2500 A.D. 6502 Macro Assembler  - Version 5.02a
          ----------------------------------------------------------------------
                    Input Filename:           DEMO1.ASM
                     Output Filename:         DEMO1.OBJ

                   Lines Assembled: 51              Errors: 0
```

And, you will get two files: DEMO1.OBJ、DEMO1.LST. Here we list some portion of the contents of the DEMO1.LST file…



```
13                P_ROM    .SECTION     OFFSET $C000, RANGE $C000 $FFF9
14 0000           POWER_ON:
15 0000   A2 FF            LDX    #$FF
16 0002   9A               TXS
17 0003   A0 00            LDY    #0
18 0005   A9 27            LDA    #<DATA_DB
19 0007   85  00           STA    DATA_ADDR_L
20 0009   A9 00            LDA    #>DATA_DB
```

Instruction code
Source code
Line number
Offset value from current section, it will be changed to absolute address after linking.

If there is any syntax error in the source code, the error messages will be shown in the list file. For an instance, <u>LDY #$FFF</u> is an instruction with a too large immediate value, so the error will be shown out as below…

```
13                P_ROM    .SECTION     OFFSET $C000, RANGE $C000 $FFF9
14 0000           POWER_ON:
15 0000   A2 FF            LDX    #$FF
16 0002   9A               TXS
17 0003   A0 FF            LDY    #$FFF
  ***** demo1.asm : Line 17 *****
  ***** # TOO LARGE *****
18 0005   A9 27            LDA    #<DATA_DB
19 0007   85  00           STA    DATA_ADDR_L
```

1.2 Command Line Mode

Using the example above, you can assemble a source code in “Command Line Mode” by typing…

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\X6502 DEMO1 -D
```

1.3 Filename Extension

The file types listed in table are frequently encountered when you use 2500AD 6502 assembler / link. Those file types are…

Assembler	
.asm	Source code, input to the assembler
.obj	Object file, output from the assembler
.lst	Listing file
Linker	
.obj	Object file, input to the linker
.tsk	Executable binary machine code file
.hex	Intel Hex and Extended Intel Hex format file, output from linker
.dcf	2500AD high level debug control file
.sym	Symbol table file, which includes symbols, labels and corresponding address
.map	Load map file, which includes section range and the corresponding address for loading each section

1.4 Assembly Language Syntax

1.4.a Number Base Designations

The 2500AD assembler accept number operands in several forms. The form is specified by applying an appropriate prefixes or suffixes to numbers, as follows:

Prefix	**Operand Form**	**Suffix**
None	Base 10 (Decimal)	D
$	Base 16 (Hexadecimal)	H
@	Base 8 (Octal)	O or Q
%	Base 2 (Binary)	B
“ or ‘	ASCII	” or ’

[Example] 16_{10} can be represented as:

Format	Base
16 or 16D	Decimal
$10 or 10H	Hexadecimal
@20 or 20O or 20Q	Octal
%00010110 or 00010110B	Binary

1.4.b Program Comments

The comment field is always optional, and is used to add an explanatory note to a statement. The text comment should be proceeded by a semicolon (;). For instance,

```
          LDX     #$FF          ; STACK POINTER = $01FF
          TXS                   ; HERE IS PROGRAM COMMENTS
```

1.4.c Program Counter

You can use the special dollar sign ($) in an expression to specify the program counter. The dollar sign is assigned the program-counter value at the start of an instruction. The instruction below causes an endless loop of jumping to itself.

```
STORE_END:
          JMP     $
```

1.4.d Global Labels

All labels, which are case-sensitive (a letter in upper case is different from the same letter in lower case), must start with an alphabetic character or underbars (_). Global labels can have any number of characters, but only 32 are significant. A label can start in any column if its name ends with a colon (:). If there is no colon, the label must start in column 1. For example:

```
POWER_ON:                                              ①
          LDX     #$FF          ; STACK POINTER = $01FF
SET_SP                                                 ②
          TXS                   ; HERE IS PROGRAM COMMENTS
  MOVE_DATA:                                           ③
          LDY     #0
          LDA     #<DATA_DB
          STA     DATA_ADDR_L
          LDA     #>DATA_DB
          STA     DATA  ADDR  H
```

1.4.e Local Labels

Local labels are used like global labels, but the definition of a local label is valid only within its own "local area", one bounded by labels which keep their definition throughout the entire program. When a program passes from one local area to the next, you can reuse local label names, since these are referenced only within their own local areas. The local labels are with three characteristics:

- ➢ The assembler identifies a local label by the (?) prefix or suffix.
- ➢ You can use any character, and up to 32 of them.
- ➢ The effective range for any local label is between two global labels.

[Example]:

```
DELAY1:
            LDX    #0
?LOOP1:                                         ❶
            DEX
            BNE    ?LOOP1                        ①
?LOOP2:
            LDY    #10
            DEY
            BNE    ?LOOP2
DELAY2:
            LDY    #0
?LOOP1:                                         ❷
            DEY
            BNE    ?LOOP1                        ②
            JMP    ?LOOP2                        ③
```

Description:

① Program control will branch to "?LOOP1" marked with ❶.

② Program control will branch to "?LOOP1" marked with ❷.

③ The JMP instruction will cause a syntax error because "?LOOP2" local label is not at local area. (Actually, the "?LOOP2" local label effective area is between DELAY1 and DELAY2 global labels)

1.4.f High / Low byte

```
18 C005   A9 27                 LDA    #<DATA_DB        ①
19 C007   85  00                STA    DATA_ADDR_L
20 C009   A9 C0                 LDA    #>DATA_DB        ②
21 C00B   85  01                      STA    DATA_ADDR_H
 :                              :                       :
35 C027          DATA_DB:                               ③
36 C027   00 01 02 03 04   DB
$00,$01,$02,$03,$04,$05,$06,$07,$08,$09,$FF
          05 06 07 08 09
```

To load the high byte of a 16-bit value, use ">"(greater than mark). This allows bits 8 through 15 to be used as re-locatable byte value. To load the low byte of a 16-bit value, use "<"(less than sign). This allows bits 0 through 7 to be used as re-locatable byte value. In the above example, the address of label "DATA_DB" is $C027(③), you can load low byte of the $C027 by using "<" (see ①), and you can load high byte of the $C027 by using ">"(see ②).

1.5 Assembler Directives

If a directive starting in column one must be prefixed by a decimal point (.), otherwise the assembler treats it as a label. You can precede it with a decimal point no matter where it is in the line to distinguish it as a directive clearly. Make an explanation in the following example:

DEMO1.ASM

```
                        .CHIP           6502                                    ①
                        .SYNTAX         6502                                    ②
                        .SYMBOLS                                                ③
                        .OPTIONS        DCH                                     ④
                        .LINKLIST                                               ⑤

W_RAM                   .SECTION        PAGE0, RANGE $0 $7F, REF_ONLY          ⑥
DATA_ADDR_L:            DS      1                                               ⑦
DATA_ADDR_H:            DS      1
ALABEL_DS:  DS      10

P_ROM                   .SECTION        OFFSET $C000, RANGE $C000 $FFF9
        ⑧
POWER_ON:
                        LDX     #$FF
                        TXS
                        LDY     #0
                        LDA     #<DATA_DB
                        STA     DATA_ADDR_L
                        LDA     #>DATA_DB
                        STA     DATA_ADDR_H
NEXT_BYTE:
                        LDA     (DATA_ADDR_L),Y
                        CMP     #$FF
                        BEQ     STORE_END
                        STA     ALABEL_DS,Y
                        INY
                        JMP     NEXT_BYTE
STORE_END:
                        JMP     $

LABEL_BLKB:
        BLKB            10,$A5                                                  ⑨

DATA_DB:
        DB      $00,$01,$02,$03,$04,$05,$06,$07,$08,$09,$FF                     ⑩

DATA_DW:
        DW      $0000,$0001,$0002,$0003,$0004,$FFFF                             ❶

IRQ_ISR:
                RTI
NMI_ISR:
                RTI

VECTAB          .SECTION        OFFSET $FFFA, RANGE $FFFA $FFFF                 ❷
                DW      IRQ_ISR
                DW      POWER_ON
                DW      NMI_ISR
```

Description:

① Specify 6502 CPU instruction codes to be assembled.

② Declare the assembly language syntax is standard 6502 syntax.

③ Specify a symbol file to be generated for debugging.

④ Specify a map file, high-level debug control file to be generated, and the output file format is in Intel Hex format.

⑤ Make the linker produces re-locatable list file. After link operation completed, the addresses in list file will be changed to appropriate absolute addresses. Please refer to the next two examples:

If no LinkList declared, the addresses in list file are still the offset values in that section. Like,

```
13 0000            P_ROM    .SECTION      OFFSET $C000, RANGE $C000 $FFF9
14 0000            POWER_ON:
15 0000    A2 FF            LDX    #$FF
16 0002    9A               TXS
```

With LinkList declaration, the addresses in list file are changed to the absolute addresses:

```
13                 P_ROM    .SECTION      OFFSET $C000, RANGE $C000 $FFF9
14 C000            POWER_ON:
15 C000    A2 FF            LDX    #$FF
16 C002    9A               TXS
```

⑥ Declare a section named W_RAM, which is located in zero-page. The range of the section is $00~$7F, if you use the space over the range then the error message will appear in link. The last "REF_ONLY" option is a reference only declaration that the section does not generate real instruction codes to occupy spaces. Normally, just those sections that belonged to RAM will declare "REF_ONLY". Please note that the section name is not suffixed with colon (:).

⑦ DS 1 (Define Storage) directive reserves one byte space for variables, but does not store a value in the reserved area. This directive is used in RAM memory range. In the example below, due to "DS 1" reserves 1 byte, so the address of "DATA_ADDR_H" will be $0001.

```
 8         W_RAM            .SECTION      PAGE0, RANGE $0 $7F, REF_ONLY
 9  0000   DATA_ADDR_L:     DS     1
10 0001    DATA  ADDR  H:   DS     1
```

⑧ Declare a section named as "P_ROM" that its start address is $C000 (offset $C000), and its range is from $C000 to $FFF9 (RANGE $C000 $FFF9).

⑨ Define 10 bytes with identical value - $A5, see the following example:

```
30 C01A   4C 1A C0                        JMP    $
31
32 C01D              LABEL_BLKB:
33 C01D                                   BLKB  10,$A5
34
35 C027              DATA_DB:
```

It will generate the Intel Hex format output file like:

```
:10 C010 00 FF F0 07 99 02 00 C8 4C 0D C0 4C 1A C0 A5 A5 A5 99
:10 C020 00 A5 A5 A5 A5 A5 A5 A5 00 01 02 03 04 05 06 07 08 69
```

$C01A stores JMP $ machine code

Consecutive 10 bytes for $A5 from $C01D

⑩ DB (Define Byte) directive declares the values followed are stored in byte size. Please refer to the list file below:

```
 35 C027                     DATA_DB:
 36 C027     00 01 02 03 04        DB     $00,$01,$02,$03,$04,$05,$06,$07,$08,$09,$FF
             05 06 07 08 09
             FF
```

❶ DW (Define Word) directive declares the values followed are stored in word size. Please refer to the list file below:

```
 38 C032                     DATA_DW:
 39 C032     0000 0100 0200        DW     $0000,$0001,$0002,$0003,$0004,$FFFF
             0300 0400 FFFF
```

❷ Start a new section named "VECTAB", and specify its start address as $FFFA. This section is declared for composing the interrupt vector table of 6502 CPU.

❸ Ending the assembly process to here.

DEMO2.ASM

```
FIVE_TIME
             .CHIP          6502
             .SYNTAX        6502
             .SYMBOLS
             .OPTIONS       DCH
             .LINKLIST
             .INCLUDE       MAC.INC                                  ①

ADD_COUNT    EQU      6                                              ②
.IFDEF FIVE_TIME                                                     ③
SUB_COUNT    =        5                                              ④
.ELSE
SUB_COUNT    =        4
.ENDIF

W_RAM        .SECTION       PAGE0, RANGE $0 $7F, REF_ONLY
TEMP:        DS       1
```

```
P_ROM           .SECTION        OFFSET $C000, RANGE $C000 $FFF9
POWER_ON:
                LDX     #$FF
                TXS
                LDA     #0
                STA     TEMP
                LDX     #ADD_COUNT
        ?MORE_ADD:
                ADD     TEMP,TEMP,#3                                        ⑤
                DEX
                BNE     ?MORE_ADD
                LDX     #SUB_COUNT
        ?MORE_SUB:
                SUB     TEMP,TEMP,#2
                DEX
                BNE     ?MORE_SUB
                JMP     $
IRQ_ISR:
                RTI
NMI_ISR:
                RTI

VECTAB          .SECTION        OFFSET $FFFA, RANGE $FFFA $FFFF
                DW      IRQ_ISR
                DW      POWER_ON
                DW      NMI_ISR
                END
```

MAC.INC

```
ADD:            .MACRO          RESULT,OP1,OP2                              ⑥
                CLC
                LDA             OP1
                ADC             OP2
                STA             RESULT
                .ENDM                                                       ⑦

SUB:            .MACRO          RESULT,OP1,OP2
                SEC
                LDA             OP1
                SBC             OP2
                STA             RESULT
                .ENDM
```

Description:

① Include “MAC.INC” file into the DEMO2 file as a portion of source code by using “INCLUDE directive”.

② Equate label “ADD_COUNT” to “6”.

③ Declare a conditional assembly. Assemble the source code between .IFDEF directive and .ELSE directive if “FIVE_TIME” symbol already defined; otherwise, assemble the source code between .ELSE and .ENDIF directives.

④ Declare EQU directive is equivalent to “=”.

⑤ Using macro definition (here, “ADD” macro) to improve the readability of source code.

⑥ Define a macro named “ADD”. Please note that the macro name suffixed with colon (:) and use “.MACRO” to specify it as a macro.

⑦ End of macro.

DEMO3.ASM

```
            .CHIP       6502
            .SYNTAX     6502
            .SYMBOLS
            .OPTIONS    DCH
            .LINKLIST

BANK0       .SECTION    OFFSET $8000, RANGE $8000 $A000, INDIRECT    ①
            BLKB        $A000-$8000,$00

BANK1       .SECTION    OFFSET $8000, RANGE $8000 $A000, INDIRECT    ②
            BLKB        $A000-$8000,$11

BANK2       .SECTION    OFFSET $8000, RANGE $8000 $A000, INDIRECT
            BLKB        $A000-$8000,$22

BANK3       .SECTION    OFFSET $8000, RANGE $8000 $A000, INDIRECT
            BLKB        $A000-$8000,$33

W_RAM       .SECTION    OFFSET 0, RANGE $0 $7F, PAGE0, REF_ONLY      ③
TEMP:       DS      1

P_ROM       .SECTION    OFFSET $C000, RANGE $C000 $FFF9
POWER_ON:
            LDX     #$FF
            TXS
            LDA     #$55
            STA     TEMP
            JMP     $
```

```
IRQ_ISR:
                RTI

NMI_ISR:
                RTI

VECTAB          .SECTION     OFFSET $FFFA, RANGE $FFFA $FFFF
                DW     IRQ_ISR
                DW     POWER_ON
                DW     NMI_ISR
                END
```

Description:

① Start a new section named "BANK0", and specify it as an indirect section. We declared it as an indirect section to serve the purpose of multi-ROM-bank. Since the addressing range of 6502 CPU is limited to 64K bytes, we need build some bank-switching mechanism in the single chip we developed. For the different banks, their addresses mapped to 6502 CPU are same. The indirect section allows overlapped ROM data, so we can use this feature to serve as bank data definition.

② Start a new section named "BANK1", and specify it as an indirect section. You can see that its addresses overlap on BANK0.

③ After an indirect section declared, the zero-page section declaration should be more complete in form. You can see the declaration of zero-page section is different from that appeared in the prior example.

1-5-2 Linker

How to use link.exe?

2500AD 6502 linker provides four operation modes: prompt mode、command line mode、data file mode and enhanced data file mode. We just like to describe the first three modes here, and we also take some practical examples to aid description.

2.1 Prompt Mode

Under DOS prompt (pure DOS or DOS mode of Windows 95), typing "link" followed with the entire path to run Linker. Such as,

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\LINK
```

You will see the message as below…

```
2500 A.D. Linker Copyright (C) 1990 by 2500AD Software Inc. Version 5.03d    ①
Input Filename: DEMO1 (enter)                                                ②
Input Filename: (enter)                                                      ③

Output Filename: (enter)                                              ④

Library Filename: (enter)                                                    ⑤

Options (D,G,P  A,R,S[U] C,M,N,SM,Z  E,H,T,X,1,2,3  <CR> = Default) : (enter)   ⑥
```

Description:

① Show the version and copyright of the Assembler.

② Require .OBJ file to be linked (here, we use DEMO1 file for instance).

③ Assign another .OBJ file to be linked if necessary or press (enter) to bypass.

④ Assign the output filename. It'll use the same name of input file if you don't assign, and the extension name depends on the definition in options block.

⑤ Please give the library name if you have used, or press (enter) to bypass.

⑥ Link option. It has same effect as OPTIONS directive in source code. Press (enter) to use the options specified in source code as default, or assign new options to replace it. If you want to generate a binary executable file, for example, input the "X" option here. However, if you have used the indirect section, the "X" parameter will cause errors in link.

2.2 Command Line Mode

Under DOS prompt (pure DOS or DOS mode of Windows 95), typing "link with –C parameter" in entire path, to run Link in "Command Line Mode". Such as,

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\LINK -C DEMO1
```

2.3 Data File Mode

Firstly, make a text file like DEMO1.LNK below:

DEMO1.LNK

```
DEMO1                                                                        ①
                                                                             ②
                                                                             ③
                                                                             ④
DCH                                                                          ⑤
```

Description:

① The string in this row is corresponding to the first input string in Prompt mode. (Input Filename)

② The string in this row is corresponding to the second input string in Prompt mode. (Input Filename)

③ The string in this row is corresponding to the third input string in Prompt mode. (Output Filename)

④ The string in this row is corresponding to the fourth input string in Prompt mode. (Library Filename)

⑤ The string in this row is corresponding to the fifth input string in Prompt mode. (Options)

After a text file edition completed, typing “link with that filename” in entire path under DOS prompt (pure DOS or DOS mode of Windows 95) to run Link. Such as,

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\LINK DEMO1
```

Part II RICE65 Emulator

Chapter 2-1 Introduction

Welcome to RICE65, a real-time in-circuit emulator developed by Advanced Transdata Corporation for 6502 or WDC65C02 compatible microprocessor based products of NOVATEK Microelectronics Corp.

The RICE65 contains the emulator base unit and the PB-65C02 (or PB-6502) probe card. It also comes with 64K program-memory, 8K real-time trace memory, and a 12-clip external probe. The emulator baseboard contains all emulation and control logic and the real-time trace circuitry. The PB-65 is connected to the baseboard via header pins.

The RICE65 system works with any IBM PC or compatible system running DOS 5.0 or higher. You can also run the DOS version under Windows3.x or Windows95 DOS shell. Furthermore, we will also offer a version that you can run under Windows95 environment. The accompanied software provides a windowed development environment supplying separate windows for examining source code, program memory, special register, watch variables, processor status, program counter and stacks. In summary, the RICE65 emulator provides a flexible, portable and complete environment for developing all your 6502 or 65C02 applications.

2-1-1 System Features

- Hosted by any PC-386/486/pentium or compatibles.
- Parallel port interface.
- Real-time and transparent emulator to 8MHz.
- 64K*8 program memory.
- 8K by 24-bit wide real-time trace buffer, with 8 logic trace probes.
- Source level debugging on 2500AD and other assembler / C compiler.
- Unlimited number of internal breakpoints within program memory.
- Trigger output on any range of address within program memory.
- 12 logic probes including 8 trace inputs, a break input, a break output, a trigger output and a GND pin.
- 32-bit instruction cycle counter.
- Data capture logic.
- Emulator controls include Go from Reset, Go, From, To cursor, Single Step, Step over Call, Return to Caller, and animate.
- Search capability in source, program memory, and trace buffer.
- Support external clock and user-selectable internal clock frequencies.
- Separate windows for source code, special registers, program memory, trace, watch variables, disassembled codes, and stacks.
- All special registers and program memory can be modified directly inside the windows.

- Easy to use windowed interface with pull-down menu, dialog box, functions / hot keys, mouse support, and context sensitive help.
- Each window can be sized, moved, added or removed.
- Save software setup including window layout, set breakpoints, watch variables, trigger…etc. to editable text file.

2-1-2 Package Checklist

- RICE65 emulator unit with PB-65C02 or PB-6502 probe installed.
- Proper evaluation board (EV board or called “Target board”) for respective IC application.
- 12 clip external probe.
- Parallel adapter cable.
- 9V DC 500mA wall plug-in DC power adapter.
- Technical Reference Manual for Assembler.
- RICE65 program disk.

Please contact us if any of the above items is missing from your package.

2-1-3 System Requirements

- IBM PC 386/486/Pentium or compatibles.
- Parallel port.
- MS-DOS version 5.0 or higher or Windows95.
- At least 2MB of RAM.
- Hard Disk is highly recommended.
- Microsoft compatible mouse (necessary).

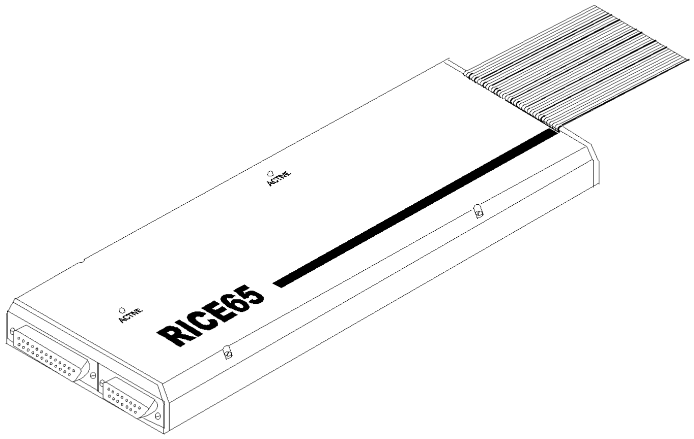
Warning:

Running RICE65 without an installed mouse will cause the program to crash.

Chapter 2-2 Getting Started

This chapter shows you how to install the RICE65 and optional hardware and software and how to get the system up and running.

2-2-1 The RICE65 Emulator

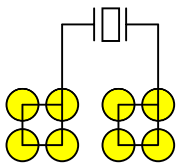


Power Jack	Found on the right side of the emulator, the power jack provides connection to the enclosed 9V-power adapter.
DB-25 Connector	The DB-25 connector on the left of the emulator is used to connect to the PC host via the enclosed parallel extension cable. This cable has connection on all 25 pins.
DB-15 Connector	The DB15 connector is used for connecting the external probe cable for logic trace, break or trigger functions.
POWER LED	The POWER LED is on whatever power is applied to RICE65. It blinks when power is initially supplied to the system during the power up self-test.
RDY STATUS LED	This LED shows the status of the RDY pin (pin2) of the 65(C)02 processor. The LED will be on when the pin is low which is the normal state for the microprocessor. When this LED is off, the 65(C)02 will be disabled and this indicates something is wrong with the target.
Emulation Cable	RICE65 can be operated with or without being connected to a target. If a target is present, the gray emulation cable connects the RICE65 emulator to the target application under test. The emulation cable ends in a 40-pin dual-in-line plug with pin1 indicated by the line red on the cable.

2-2-2 RICE65 Onboard Sockets and Jumpers

External Crystal Socket

The 8-pin socket marked “EXT XTAL” allows users to plug in a crystal and provide an exact oscillator frequency not available internally to run the application. The crystal can be plugged into sockets as shown below. If an oscillator input signal is used, please connect that to any of the four sockets to the right.



Jumpers

There are 3 sets of jumpers on the RICE65 emulator base:

① JP1 and JP2 are specifying the external oscillator clock range.
JP1: select external oscillator from 500KHz to 12MHz (default)
JP2: select external oscillator from 32KHz to 500KHz.

② JP3 and JP4 are for specifying the source of the external oscillator.
JP3: select external source from the crystal installed in the "EXT XTAL" socket inside RICE65.
JP4: select external source from the PHI2 (pin 37) of the target (default).

③ Found on the top right hand corner of the RICE65 base, these headers allow users to specify the operating voltage for the emulator. To select, shunt the corresponding jumpers next to the voltage value. Default is set at +5V. Please note that the emulation speed might be lower when operating under a lower voltage.

JP10	+3.2V	JP9	+3.5V
JP8	+4.2V	JP7	+5V (default)

2-2-3 PB-65(C)02 Probe Card Jumpers

The PB-65(C)02 Probe Card has been installed inside RICE65 to emulate the 6502 or 65C02 processor core. The probe is connected to the RICE65 base via socket and header pins. The probe card carries the 6502 or 65C02 processor for emulation. There are two jumpers on the probe card:

①JP1 FRZCTL
This jumper is just used for 65C02 processor-based products. The "Freeze Control" jumper is used to set pin35 to a low logic level when the emulator is in halt mode. It can then be used to freeze external peripherals, such as timer, during emulation.

②JP2 EXTMEM
The "External Memory" jumper is used to set pin35 to a low logic level when the processor is accessing external memory.

③JP3 +5V
The "+5V" jumper, when enabled, will provide a 5V (100mA) output to the VDD pin of the emulation plug. You can use this output to drive your target if it consumes less than 100mA.

2-2-4 External Probe Cable

The External probe cable has twelve logic probes and is connected to the emulator via a 15-pin connector. Clips on the first eight cables, using standard electronic color coding (brown to gray as for 1 to 8), are for external trace inputs. When connected to the target application, these 8 clips register the real-time logic status of those points during program execution.

Captured data are automatically stored in the real-time trace buffer. The functions of the other four logic clips are assigned as follows:

BRK:	Break Input
GND:	Common Ground
TRG:	Trigger Output
HALT:	Break Output

Pin	Name	Cable	Clip	Description	
1		Brown	Black	Bit 1 of the external trace input	
2		Red	Black	Bit 2 of the external trace input	
3		Orange	Black	Bit 3 of the external trace input	
4		Yellow	Black	Bit 4 of the external trace input	
5		Green	Black	Bit 5 of the external trace input	
6		Blue	Black	Bit 6 of the external trace input	
7		Purple	Black	Bit 7 of the external trace input	
8		Gray	Black	Bit 8 of the external trace input	
9	BRK	White	Black	Break Input–	will halt the processor when the qualified external break condition is satisfied. The external break condition is set in the Break | More Break Menu.
10	GND	Black	Black	Common Ground	
11	TRG	Brown	Red	Trigger Output-	generates a positive pulse at the trigger address, used to trigger a scope or to time measurement.
12	HALT	Red	Red	Break Output-	halts the processor at break condition and generates a positive pulse from high to low with a duration >50us. If connected to the External Break Input of another RICE65 hardware, the HALT line can be used to provide simultaneous break for multiple RICE65 emulation.

2-2-5 Hardware Installation

Connecting the Hardware

① Power down the PC (recommended but not necessary).
② Plug the parallel extension cable to an available parallel port at the back of the computer.
③ Plug the DC power adapter into a wall socket.
④ Plug the emulation cable plug into the micro-controller socket on the target system.
⑤ The system is now ready for power up.

Applying Power to System Components

Unless the +5V jumper (JP3) on PB-65(C)02 is enable (see 2-2-3), the +5V from the emulator has been isolated and a separate power source is needed for the target system. Please follow these sequences when powering up the hardware and reverse them when powering down to prevent damage to the emulator.

① Plug emulation cable to target.
② Apply power to host PC.
③ Apply power to RICE65 base unit.
④ Apply power to target system.
⑤ Plug the parallel extension cable into the emulator.

2-2-6 Software Installation

Backing Up the Diskette

Before you begin to use the RICE65 software, it is recommended you make a backup copy. The disk provided is not copy protected and a backup copy can be made using the DISKCOPY command.

Prepare a blank (either formatted or unformatted) disk, place Program Disk in drive A and type the following at the DOS prompt.

```
C:\>DISKCOPY A: A:     [Enter]       on single floppy system
C:\>DISKCOPY A: B:     [Enter]       on dual floppy system
```

Follow the instructions on screen and copy all files from the Program Disk (source) to the blank disk (target). Place the original disk in a safe place and proceed with the installation procedure.

Creating a New Subdirectory for RICE65

Make a new directory RICE65 and copy all files from the diskette (placed in drive A) to the new directory.

```
C:\>MD RICE65          [Enter]
C:\>CD\RICE65          [Enter]
C:\RICE65>COPY A:*.*   [Enter]
```

Setting a PATH statement

It is recommended to add the RICE65 directory to the PATH statement in the **Autoexec.bat** file and work in the directory where your working files (like .ASM or C source files, .INC or .H header files), reside.

```
PATH C:\;C:\DOS;C:\RICE65;
```

This way, you can call up RICE65 anywhere. Remember to restart the computer or run the Autoexec.bat file again for the new path to take effect.

2-2-7 What RICE65 Can Do for You

RICE65 helps you to develop and test 65(C)02 based applications by locating error and finding the cause of the error. It does these by slowing down program execution so you can examine the state of the program at any given spot. You can even test new values in different variables to see how they affect the program. Specific functions that aid in this debugging process are given below:

Stepping	Execute the program one instruction or multiple instructions at a time and watches if the program is behaving the way it should.
Stepping over Call	Execute the program one instruction at a time, but stepping over any subroutine calls. This function speeds up debugging by stepping over subroutine that you are sure to be error-free.
Viewing	Open a special window to show the state of the program from various perspective: source file, program memory, variables and special registers.
Setting Breakpoints	Control where the program stops running to examine the program’s status.
Real-time Tracing	Capture external data from the circuit and aids in debugging hardware. When working with the breakpoint system, it records all executed instructions prior to the break.
Changing	Replace the current value of a variable with a new value and see its effect on the program.
Watching	Isolate program variables and keep track of their changing values as the program runs.

You can use these functions to dissect your programs into discrete blocks, confirming that each block works before moving to the next. In this way, you can burrow through the program, no matter how large or complicated, until you find where the bug is.

Chapter 2-3 The RICE65 Software Environment

This chapter introduces the RICE65 user interface and describes how to control windows, choose commands from pull-down menus and move around dialog box.

2-3-1 Using a Mouse

It is necessary to have a mouse installed on the system, otherwise the program will not operate properly. To use a mouse, make sure it is properly connected and the mouse driver loaded before you start the program. Only the left button of your mouse is used for most of the time except for during the Run mode, pressing the right button will stop the program running.

To make selections with the mouse, point to the object and click the left button. When sizing or moving windows, hold down the left button and drag the mouse to a new location, release the left button to confirm the selection.

2-3-2 Getting in RICE65

If you have added the RICE65 directory in the PATH statement in the **Autoexec.bat** file, you can run RICE65 from anywhere by typing one of the followings:

```
[1]  C:\RICE65>RICE65                [Enter]
[2]  C:\RICE65>RICE65 filename       [Enter]
```

here, filename is an object file in hex (with an HEX extension).

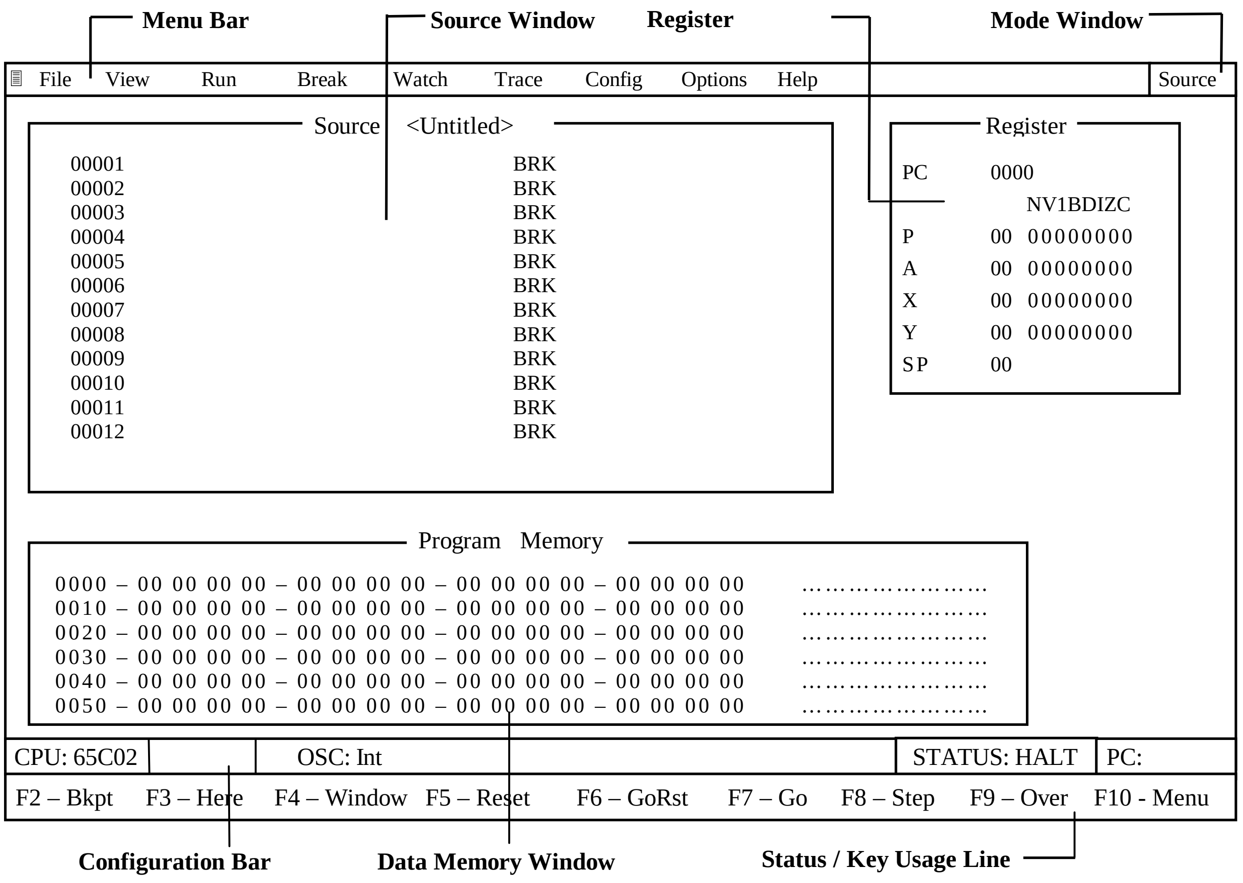
If the RICE65 command is followed by an object file which resides in the current directory as in case [2], the object file specified will automatically be loaded into the source window when the RICE65 program starts. The setup file, with a .SET extension, will also be loaded if it has the same root name. If no filename is supplied as in case [1] or if RICE65 can’t find the specified file in the current directory, it opens a blank disassembled object file in the source window.

2-3-3 The RICE65 Main Screen

RICE65’s main screen contains a global menu bar at the top of the screen, a Configuration Bar and a Status / Key Usage Line at the bottom. Three default windows are opened at start, the Source Window, Register Window and Program Memory Window.

Familiarize yourself with the menu functions by clicking on each pull down menu and look at each function. The following elements of the RICE65 screen will be described next.

- Menus
- Dialog Boxes
- Windows
- Configuration Bar
- Status / Key Usage Line



```
 File   View    Run    Break   Watch   Trace   Config   Options   Help                 Source

            Source  <Untitled>                                        Register
  00001                          BRK                            PC    0000
  00002                          BRK                                    NV1BDIZC
  00003                          BRK                            P     00  00000000
  00004                          BRK                            A     00  00000000
  00005                          BRK                            X     00  00000000
  00006                          BRK                            Y     00  00000000
  00007                          BRK                            SP    00
  00008                          BRK
  00009                          BRK
  00010                          BRK
  00011                          BRK
  00012                          BRK

                     Program   Memory
  0000 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00    ……………………
  0010 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00    ……………………
  0020 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00    ……………………
  0030 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00    ……………………
  0040 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00    ……………………
  0050 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00    ……………………
CPU: 65C02 |      | OSC: Int                              | STATUS: HALT | PC:
F2 – Bkpt   F3 – Here   F4 – Window  F5 – Reset   F6 – GoRst   F7 – Go   F8 – Step   F9 – Over  F10 - Menu
```

2-3-4 Menu

The RICE65 has a convenient menu system accessible from a menu bar running along the top of the screen. The main menu is always available except when a dialog box is active. A pull down functional menu is available for each item on the menu bar. Through these menus, you can:

- Execute a command.
- Access a submenu indicated by the menu item followed by a submenu icon (>>).
- Open a dialog box indicated by a menu item followed by a dialog box icon (…).

Using the Menu

You can access the functional menus:

- Anytime by using the [Alt]-[Key] combination:

[Alt]-[S]	System Menu	[Alt]-[F]	File Menu
[Alt]-[V]	View Menu	[Alt]-[R]	Run Menu
[Alt]-[B]	Break Menu	[Alt]-[W]	Watch Menu
[Alt]-[T]	Trace Menu	[Alt]-[C]	Configuration Menu
[Alt]-[O]	Options Menu	[Alt]-[H]	Help Menu

- By pressing [F10] to activate the menu bar, then
  ① uses the [↑] or [↓] arrow keys to highlight the desired functional menu and pressing [Enter].
  ② presses the highlighted letter of the functional menu.
- By clicking the selection of the menu bar with the left mouse button.

At the functional menu level:

- Use the [↑] and [↓] to move from one pull-down menu to the other.
- Press the highlighted letter to execute a command.
- Use [↑] and [↓] to scroll through the available commands and presses [Enter] to execute it.
- Click the left mouse button on a command.
- Press [Esc] to exit functional menu and returns to the main menu bar.

At the submenu level:

- Use the [↑] and [↓] to scroll through the selections and presses [Enter] to execute the command.
- Press the highlighted letter to execute a command.

To get out of the menu system:

- Press [Esc] to return to the menu bar and then [F4] or [Tab] back to the active window.
- Click a window with the left mouse button to leave the menu system and go to that window.

2-3-5 Dialog Boxes

If a command is followed by an ellipsis (…), it means more information must be provided before the command is executed. You provide this information in a dialog box, which appears when you issue the command. Dialog boxes can contain one or more of the following items:

Item	**Description**
(*) Radio button	Radio buttons offer a set of toggles; only one option can be chosen. Use the arrow keys to move between the choices and press [Spacebar] to select.
Push button	Push buttons are "shadowed" text, which pass commands to the dialog box. The $\boxed{\text{OK}}$ button exits the dialog box without changing the current settings.
Edit input	An edit input prompts you to type in a string.
List box	A list box contains a list of items from which you can choose. A typical list box is one to select the file to be debugged under File | Open File…command.

Moving Around Dialog Boxes

You navigate around a dialog box by pressing [Tab] or [Shift]-[Tab] key that moves the cursor from one item to the next. Within sets of radio buttons, use the up / down arrow keys to select the setting and press [Spacebar] to confirm the selection. To choose a push button, tab to it and press [Enter]. If you have a mouse, just click the item you want to choose. To cancel the dialog box, click the Cancel button.

2-3-6 Windows

RICE65 displays all information and data in windows. Some windows are for display only while others allow you to modify their contents. You can open most windows using commands in the View Menu.

When RICE65 starts, the Source Window, Register Window and Program Memory Windows will be opened by default. No matter how many windows are opened, only one window is active at a time. The active window has an intense white border around it and a title of a different color. If your windows are overlapping, the active window is the topmost one. The Mode Window at the upper right hand corner also displays the name of active object. Followed are the different types of windows you can open from the View Menu.

Program Memory Window

Displays the instruction codes in program memory in hex and ASCII formats. You can change the value of the registers by just typing over it. To do so, first press [F4] or [Tab] until Program Memory is the current window; then use the arrow keys to highlight the field to be edited and type in the new value. The new values will be updated in the Source Window, ASM Window and in the Emulator.

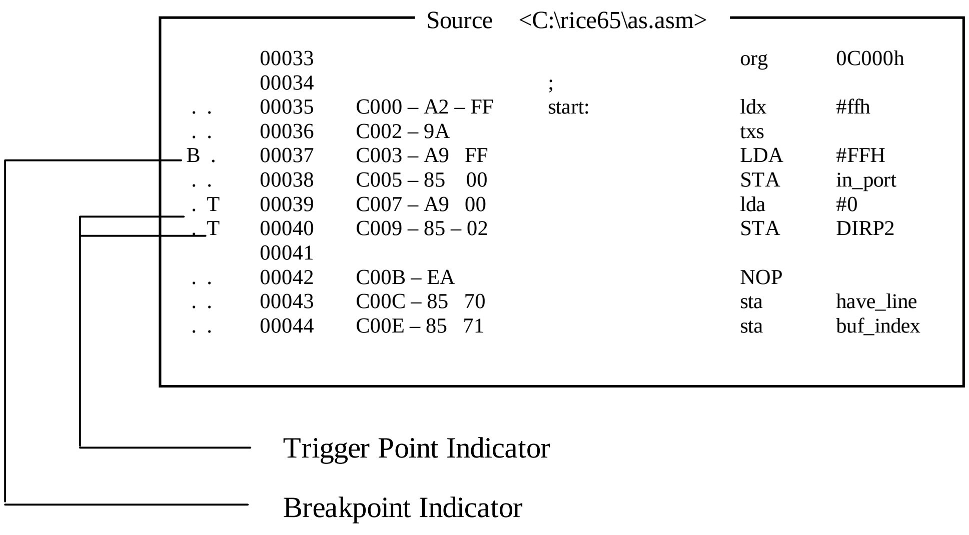
```
                          Program   Memory
0000 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00     ……………………
0010 – 48 65 6C 6C– 6F 20 57 6F – 72 6C 64 21 – 20 20 20 00     Hello World!
0020 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00     ……………………
0030 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00     ……………………
0040 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00     ……………………
0050 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00     ……………………
```

The Program Memory Window can be enlarged to display more registers. Before doing so, it is recommended to change the display to 43 / 50 line under Options | Display options. During stepping, only data in range 0 – 1FF and the visible range in the Program Memory Window will be updated. This ensures stepping to be more responsive.

Source Window

The Source Window consists of breakpoint and trigger point indicators, line number, program counter, op code and the source codes or disassembled codes of the program you are debugging.

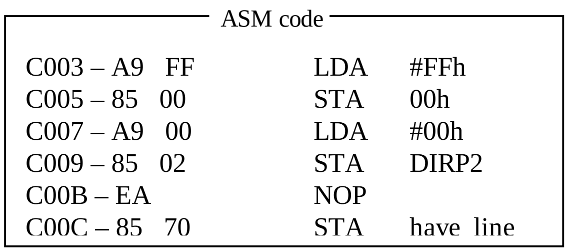
You can move around the window using arrow keys or clicking the scroll bar with a mouse. A triangular cursor marks the next executable line. If the Source Window does not display the original source lines, this means RICE65 cannot locate the ASM file in the current directory.



```
Source    <C:\rice65\as.asm>

      00033                                          org    0C000h
      00034                          ;
 . .  00035   C000 – A2 – FF         start:          ldx    #ffh
 . .  00036   C002 – 9A                              txs
 B .  00037   C003 – A9   FF                         LDA    #FFH
 . .  00038   C005 – 85   00                         STA    in_port
 . T  00039   C007 – A9   00                         lda    #0
 . T  00040   C009 – 85 – 02                         STA    DIRP2
      00041
 . .  00042   C00B – EA                              NOP
 . .  00043   C00C – 85   70                         sta    have_line
 . .  00044   C00E – 85   71                         sta    buf_index
```

ASM Window

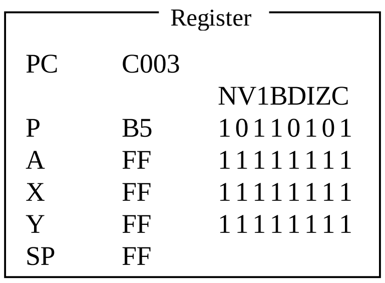
The ASM Window displays the disassembled codes for the object codes. When debugging in C, users can open this ASM Window to display the corresponding disassembly codes. The highlighted line in the ASM Window corresponds to the C line in the Source Window.



```
ASM code

C003 – A9   FF          LDA    #FFh
C005 – 85   00          STA    00h
C007 – A9   00          LDA    #00h
C009 – 85   02          STA    DIRP2
C00B – EA               NOP
C00C – 85   70          STA    have_line
```

Register Window

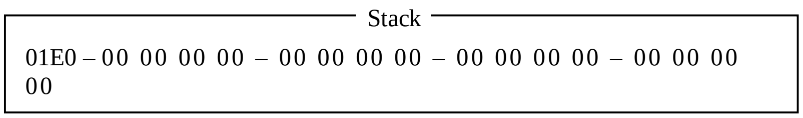
Displays the current contents of the program counter, the Processor Status (P or PS) register, the Accumulator (A), the X and Y Index registers and the Stack Pointer (SP). The values are displayed in both hex and binary formats. You can change the hex value of the registers by typing over them. To edit the codes, use the arrow keys to highlight the field and type in the new value.



```
Register

PC     C003
                NV1BDIZC
P      B5       1 0 1 1 0 1 0 1
A      FF       1 1 1 1 1 1 1 1
X      FF       1 1 1 1 1 1 1 1
Y      FF       1 1 1 1 1 1 1 1
SP     FF
```

Stack Window

Displays the stack memory from 0 to 100x1FF in program memory of the 65©02. This window can be sized to display the desired stack locations.



```
Stack

01E0 – 00 00 00 00 – 00 00 00 00 – 00 00 00 00 – 00 00 00
00
```

Watch Window

Displays the name, address and Hex value of the watch variables you define for specific viewing. You can add variables to the Watch Window using the [Ctrl]-[W] hot key. Watch variables can also be found in the Program Memory Window in registers as defined by the user. Followed is a sample Watch Window.

```
Watch

Buf_index <0071> = 0000
Fail <C0AE> = 46
Have_line <0060> = 00
```

Trace Buffer Window

Displays the contents of the 8K*24-bit circular trace buffer board which records the real-time executed instructions. The most recent instruction is found at the bottom of the buffer. The buffer contains:

- Line number.
- Data bus content or captured data on the eight external probes.
- Instruction cycle.
- Address.
- Op code.
- The source or disassembled code.

```
                                   Trace Buffer
Line   Bus    Cycle   Address              Instruction-code                        comments
0001   FF     3       C02C – A5  70                   lda      have_line           ; check
0002   00     2       C02E – EA                       NOP
0003   F0     2       C02F – F0  E4                   beq      wait                ; not do
0004   78     6       C015 – EE  F0  01    wait:      INC      TTTA
0005   DC     4       C018 – AD  F0  01               LDA      TTTA
0006   DC     2       C01B – D0  0A                   BNE      WAIT1
0007   EE     6       C027 –  20   90  C0  WAIT1:     JSR      TTLP
0008   C0     3       C090 –  48           TTLP:      PHA
0009   DC     2       C091 – 70   0F                  BVS      OVERFLOW
```

Trace buffer showing data bus contents

```
                                   Trace Buffer
Line   8 7 6 5 4 3 2 1   Cycle   Address                  Instruction-code
0001   1 1 1 1 1 1 1 1   3       C02C – A5  70                       lda      have_line
0002   1 1 1 1 1 1 1 1   2       C02E – EA                           NOP
0003   1 1 1 1 1 1 1 1   2       C02F – F0   E4                      beq      wait
0004   1 1 1 1 1 1 1 1   6       C015 – EE   F0  01       wait:      INC      TTTA
0005   1 1 1 1 1 1 1 1   4       C018 – AD  F0  01                   LDA      TTTA
0006   1 1 1 1 1 1 1 1   2       C01B – D0  0A                       BNE      WAIT1
0007   1 1 1 1 1 1 1 1   6       C027 –  20   90  C0      WAIT1:     JSR      TTLP
0008   1 1 1 1 1 1 1 1   3       C090 –  48               TTLP:      PHA
0009   1 1 1 1 1 1 1 1   2       C091 – 70   0F                      BVS      OVERFLOW
```

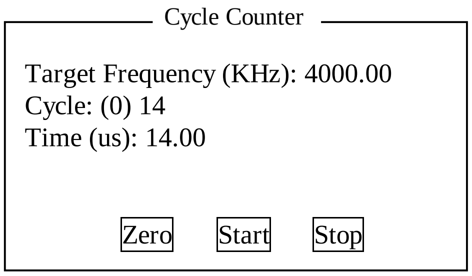
Trace buffer showing external captured data

You can scroll through the buffer using the arrow keys or by clicking the vertical bar with a mouse. The contents can also be saved to a text file using the File | Save >> Trace Buffer…command.

Users can stop capturing data but keep the processor running once the buffer is full. This is achieved through the Break | Set Breakpoints…| More Break >> Stop on Trace Buffer Full function.

Cycle Counter Window

The Cycle Counter Window displays instruction cycle and elapse time information. The execution time is based on the clock source specified under the OSC Frequency dialog box. You can reset the clock to zero, start or stop the clock anytime by clicking the corresponding button.

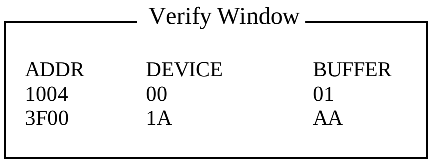


Cycle Counter

Target Frequency (KHz): 4000.00
Cycle: (0) 14
Time (us): 14.00

Verify Window

The Verify Window displays the verification errors when comparing the contents of the program memory inside the emulator against those in the memory buffer. If there is no error, the message “No verify Error” will appear.



Verify Window

ADDR	DEVICE	BUFFER
1004	00	01
3F00	1A	AA

2-3-7 Working with Windows

Once you have opened these windows, you can move, re-size or close them with commands from the System Menu or via their respective hot keys.

Moving Windows

You can easily move the onscreen windows around using the [Shift]-[F3] key. First, make the window you want to move active by pressing the [F4] key and then press [Shift]-[F3]. When the window changes to have a thick border, use the arrow keys to move the window around. Press [Enter] to confirm the change. If you have a mouse, click on the title bar and drag the mouse to define a new location. Release the mouse button to confirm the change.

Moving between Windows

You can cycle through windows on screen by pressing [F4] and [Shift]-[F4] keys. If you have a mouse, you can activate a window by clicking inside it.

Sizing Windows

To size a window, use the [Alt]-[F3] key. After making the window active, press [Alt]-[F3]. Once the border of the window changes, use the arrow keys to resize the active window. Press [Enter] to confirm the change. If you have a mouse, click and drag the lower right corner to resize the window. Release the button when done.

NOTE: Some windows cannot be sized while some can only be sized in one direction.

Closing Windows

When you are through working with a window, e.g. Trace Buffer or Program memory Window, you can close it by selecting System | Close Window command or press the [Alt]-[F4] hot key combination. If you have a mouse, click the inverted triangle at the top left corner of the window to close it.

Re-opening Windows

You can re-open a closed window by specifying it under the View Menu.

Restoring Window Layout

You can use the System | Restore command to repaint the screen the way the program was initially started with only four default windows: Source, Register, Data Memory and Stack Windows.

2-3-8 The Configuration Bar

The Configuration bar is located towards the bottom of the screen. It displays the source of the oscillator (OSC) to run the target application, status (HALT/RUN/WAIT) of the emulator processor and the current program counter.

CPU: 65C02 OSC: Int		STATUS: Halt	PC: C005

2-3-9 Status/Key Usage Line

The status line is located at the bottom of the RICE65 screen. When the menu system or a dialog box is active, the status line provides information on the highlighted item.

When one of the windows is the active screen object, the status line provides at-a-glance keystroke summary on the different emulation controls.

2-3-10 Getting Help

Getting context-sensitive help anywhere within the RICE65 program is simple. Select the desired menu option and then pressing the [F1] key. You will see a brief description of the feature in the Help dialog box. To exit, simply press the [Esc] key or click the Exit button.

You can learn a lot by working your way through the menu system and pressing [F1] at each command to get a brief description of what it does. Select Index button in the dialog box to see and access a list of items with help information available.

Chapter 2-4 RICE65 Basics

This chapter describes how to use RICE65 for debugging and introduces related files and basic functions of the system.

- Files provided on the Program Disk
- Oscillator source
- Program Execution
- Breakpoints
- Real-time Trace
- RICE65 debugging environment

2-4-1 The RICE65 Program Files

RICE65.EXE	The main executable program for the RICE65 system. Type **RICE65** or **RICE65 <filename>** to start the program. The second option runs the RICE65 program and loads the specified hex file to the Source Window. It also loads the setup file by the same root name with a .SET extension if found in the same directory.
RICE65.MSG/NDX	Files containing help information and help index to the RICE65 software.

2-4-2 The Application

Use an editor of your choice to create the application program. Currently, RICE65 only support source-level debugging for 2500AD assembler and compiler and WDC / Zardoz Assembler. Once you get a successful compilation, the 2500AD will generate the HEX (Intel Hex), DCF and SYM files used by RICE65 for source level debugging.

To start RICE65, simply type RICE65 at the DOS prompt followed by the optional name of the program, and press [Enter].

RICE65 will load the source program to the Source Window if the .HEX, .ASM, .DCF and .SYM files are found in the current directory. When only the .HEX file is found, the file will be disassembled and loaded into the Source Window. When no file by such name is found, RICE65 will load a blank disassembled file.

2-4-3 Oscillator Source

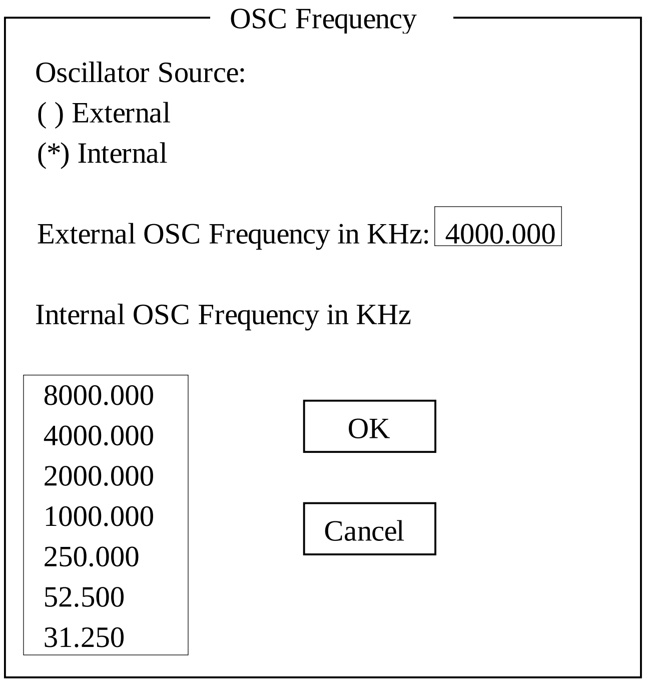
RICE65 supports both internal and external oscillator for the 65C02 processor from one of the following sources:

- an external crystal plugged inside the “EXT XTAL” socket of the RICE65 board. Make sure to jumper JP3 and one of JP1 or JP2 for the speed range. See page 6 for more information.
- an external oscillator input connected to the PHI2 (pin-37) of the target processor.
- from the internal clock frequencies provided by the RICE65 system, at 8MHz, 4MHz, 2MHz, …, 31.25KHz.

You can always select the internal clock and specify an appropriate frequency to walk through the codes, even without the presence of the target application.

To select oscillator source, press [Ctrl]-[F] or go to Config | OSC Frequency dialog box as shown below where you can specify the oscillator source for the clock and frequency for the internal source. Use the [Tab] key to cycle among the different fields listed below:

clock source	Use the up/down arrow key to select and then press [Spacebar] to confirm.
External Frequency	This is used to calculate elapse time in the Cycle Counter Window. Enter the value in KHz in the edit box.
Internal Frequency	When the list box is active, use the up/down arrow key to scroll through the list and select the desired frequency. Press [Tab] to exit to another field.


OSC Frequency

Oscillator Source:
( ) External
(*) Internal

External OSC Frequency in KHz: 4000.000

Internal OSC Frequency in KHz

8000.000
4000.000
2000.000
1000.000
250.000
52.500
31.250

OK

Cancel

If you have a mouse, just click the item you want to choose. Select OK to accept the settings and Cancel to cancel the changes.

2-4-4 Program Execution

Real-time Execution

Real-time emulation occurs when you issue a command with

- [F6] Go from Reset,
- [F7] Go, or
- [Alt]-[F7] From…
- [Shift]-[F7] Go from Cursor
- [F3] To Cursor.

These commands run the processor until encountering breakpoint or user interruptions. During real-time execution, the contents of the registers will not be retrieved or updated until the processor is halted.

Non-real-time Execution

Non-real-time execution occurs with the following commands:

- [F8] Single Step
- [Shift]-[F8] Asm Single step
- [F9] Step over Call
- [Shift]-[F9] Return to Caller
- [Alt]-[F8] Animate

  Unlike real-time execution, these commands force the emulator to execute single opcodes and thus allow users to step through codes, watch program flow and see all register contents.

2-4-5 Breakpoints

A breakpoint is a condition in which the processor halts program execution when a certain criterion is met. RICE65 can be programmed to halt in the following ways:

Break on address match

Allows users to halt the processor when the program counter equals certain values. Note that the processor breaks only after the valid instruction has been executed.
There is also a Multiple Break function which halts program execution after a break address has been executed up to 2047 times.

Break upon an external break trigger

One of the external probes on RICE65 is Break Input. It is used to halt the processor when the line sees a rising or falling edge.

Break on trace buffer full

Users can program the emulator to halt whenever the 8K real-time trace buffer is full. This break will automatically be in force when forward trace option is enabled.

Multiple Break

Found under the Break | Set Breakpoints>>More Break menu, this allows users to halt the processor after an instruction at a set breakpoint has been executed a particular number of times.

Data Capture Break

Found under the Break | Data Capture Break… menu, this function allows the processor to halt when the value in the captured address matches the specified value. A mask value can also be entered for special bit comparison. **NOTE:** When the oscillator speed is above 1MHz, the Multiple Break and Data Capture Break may execute several more instructions prior to breaking.

Break when Writing to Read – Only Memory

Once the user has defined an address range for read only memory, the processor will halt whenever the referenced address is being written to.

2-4-6 Real-time Tracing

The 8K by 24-bit trace buffer is a circular buffer which continuously captures the following data, with the most recent information stored at the bottom of the buffer.

- line number
- data bus content or captured data on the eight external probes (via the Options | Trace Options menu),
- instruction cycle
- address
- opcode, and
- the source or disassembled code (via the Options | Trace Options menu).

  The trace-buffer will automatically records the above data as program codes are executed. You can also specify special tracing under the Trace | Set Trace Range… Menu.

Backward Trace

This is the **default** trace mode, saving all executed instructions in the 8K buffer and throwing away old ones. The last executed instruction will be found at the bottom of the buffer. It is not necessary to input an address range for this trace.

Enable Halt Trace

You are required to enter the Start and End address for this option. The emulator will clear the buffer upon reaching the Start, start-capturing data until the End address at which a breakpoint has been set internally. To view the data, use the Upload Trace command.

Real – Time Trace

In this option, the processor will keep running while data within the address range will be repeatedly captured to the buffer. You can also enter multiple address ranges to be traced.

To do so, go the Break | Set Trigger Points Menu and select the addresses as trigger address and then enable “Real-time Trace” in the Trace | Set Trace Range Menu.

Forward Trace

The Forward Trace will start capturing data from the Start address and the processor will break if the trace buffer is full. This prevents the Start Address from being overwritten by the circular buffer. Note that if the Start Address is within a loop and keeps repeating, only one instance will be saved to the trace buffer. The processor will keep running. In such case as the buffer is not full to cause a break.

Working together with the breakpoint system, the trace buffer contains valuable information for debugging.

2-4-7 RICE65 Debugging Environment

When RICE65 initially starts, only the Source Window, Register Window and Program Memory Window are opened. You can open additional windows via the View Menu to aid in debugging.

To customize RICE65, open the windows you need, size them and place them in any location you feel helpful with your debugging. Set desired breakpoints, trigger points, watch variables, etc. To save the setup, select Options | Save Setup.., and press [Enter] to use the proposed *filename*.SET filename.

Since the setup file shares the same root name as the object file, the layout will be restored whenever the object file is opened. This allows you to quickly resume the debugging session the way you left it.

The Setup .SET File

The setup of the debugging environment is saved in an editable text file in the current working directory. You can edit this file using any editor.

```
[Port]
LPT=0x378
Delay=1

[Processor]
MCU=65C02
Osc=1
Freq=2
KHZ=4000.000

[Code]
File=demo65.hex
Trace=1
```

```
[Window]
Program=0 5 0 22 40
Source=1 1 0 14 34
Register=1 1 55 18 66
Stack=0 1 72 4 79
Watch=0 3 62 10 79
Trace=0 1 17 21 77
Asm=0 29 0 36 29

[Display]
Line=0
Tabsize=8
Speaker=0

[Break]
Num=0
Trigger=0
Buffer Full=0
BufferStop=0
MultipleBreakFlag=0
MultipleBreakCount=0

[Data_Capture]
Flag=0
Address=0xD400
Mask=0xFF
Compare=0x45

[Trig]
Num=0

[Watch]
Num=0

[Trace]
Trace_Range=0
Begin=0x0000
End=0x0000

[Update_Memory]
Num=0
```

```
[External_Memory]
Num=0

[ReadOnly_Memory]
Num=0
```

Chapter 2-5 File Menu

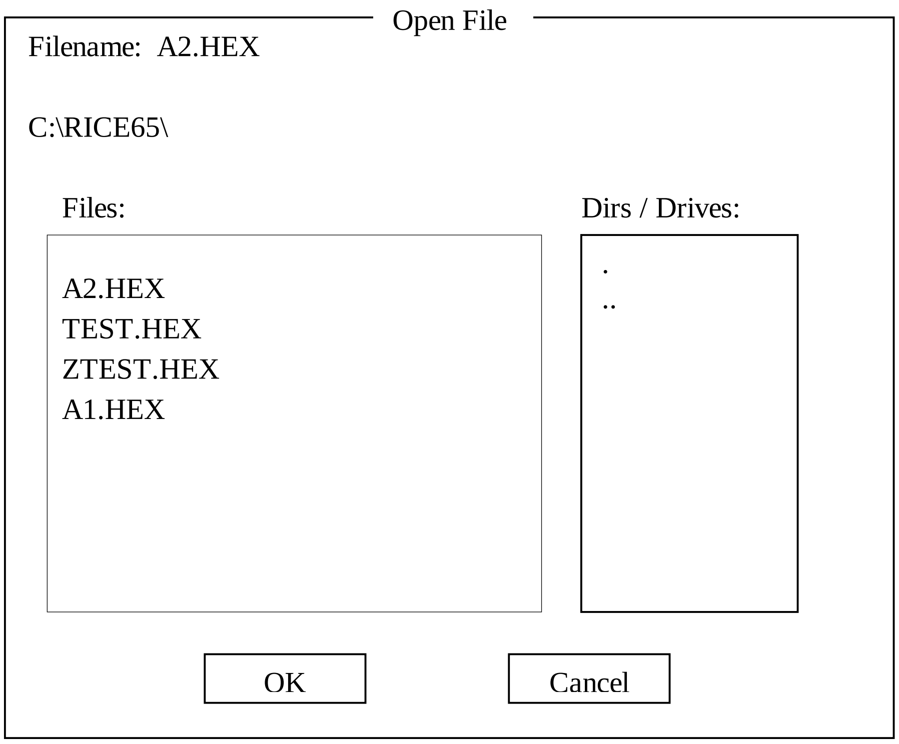
The File menu has a number of options that handle operations external to RICE65 program, such as loading a program to debug; saving the disassembled code, trace buffer, file registers or program memory; uploading and downloading program memory; logging to a different directory, executing DOS commands and returning to DOS.

2-5-1 Open File…

The Open File… command loads a program from disk to debug and downloads it to the emulation memory of the RICE65 hardware. Select from the pop-up list box (with an automatic *.Hex filter) the program to debug. The program must be an Intel Hex file with an .HEX extension.

RICE65 supports source level debugging on different assembler compilers. Please refer to Appendix A for more details.

To load a file, go to File | Open File… or press [Ctrl]-[O] to access the Open File dialog box. The list box will display files with HEX extension in the current directory. The first file in the list box will be highlighted. Use the up/down arrow key to highlight the file to be opened and press [Enter]. The selected file will be loaded in the Source Window and program memory downloaded to the emulation memory of the RICE65 hardware.


Open File

Filename: A2.HEX

C:\RICE65\

Files:

A2.HEX
TEST.HEX
ZTEST.HEX
A1.HEX

Dirs / Drives:

.
..

To change directory, press [Tab] to highlight the list box for Dirs/Drives and select the new directory desired.

2-5-2 Save>>

The Save>> command leads to a submenu through which you can save the contents of the followings to disk in the current directory:

- trace buffer
- disassembled code
- program memory

  Press the highlight character or move the highlight bar to select the option and press [Enter]. You are prompted for the name of the file in the upcoming dialog box. Enter the complete filename including the file extension, if desired.

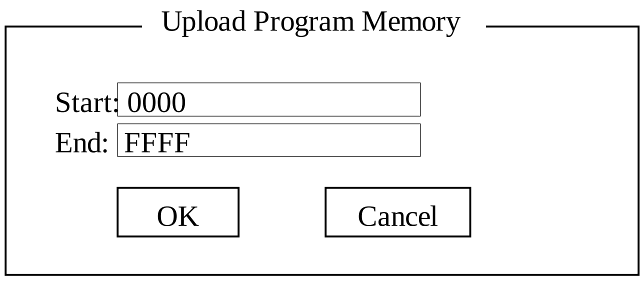
2-5-3 Download Program

The Download Program command lets you download the hex codes from the program memory buffer into the emulation memory inside the RICE65 hardware for debugging.

2-5-4 Upload Program

The Upload Program command allows you to upload a specified range of program memory from the emulator memory or external memory into the program memory buffer which can be viewed in the Program Memory Window.

To read from External Memory, first set the memory range under the “Option | Set Memory Range..” menu.



Upload Program Memory

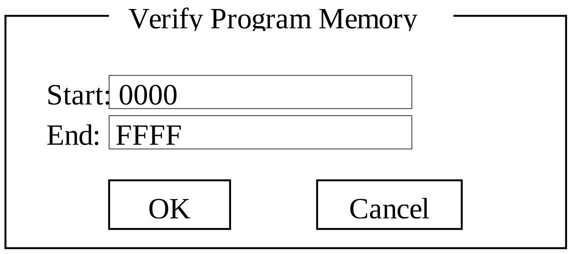
Start: 0000

End: FFFF

At the dialog box, enter the Start and End address, press [Tab] to or click the $\boxed{\text{OK}}$ button to exit. If the specified address range is within the external memory range defined as previously discussed, the data will be read from external memory in the target.

2-5-5 Verify Program

This command compares the data in the specified address range from the emulator or external memory to the program memory buffer. If there is error, the Verify Window will be opened, else, the message “Verify Program Pass” will appear. You can then press [Enter] to exit.


Verify Program Memory

Start: 0000

End: FFFF

2-5-6 External Memory

The command leads to a submenu with different file operations regarding the external memory:

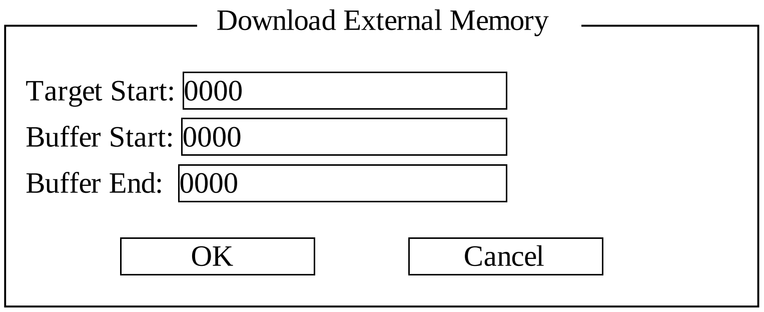
- Open File
- Download External Memory
- Upload External Memory

Open File

This command leads to the Open File dialog box where user can specify a hex file to be loaded into the external memory buffer in the PC.

Download External Memory

This command downloads a specified range of data, which is in the external memory buffer of the PC, to the memory in the target application. Use the [Tab] key to go to the different fields.


Download External Memory

Target Start: 0000

Buffer Start: 0000

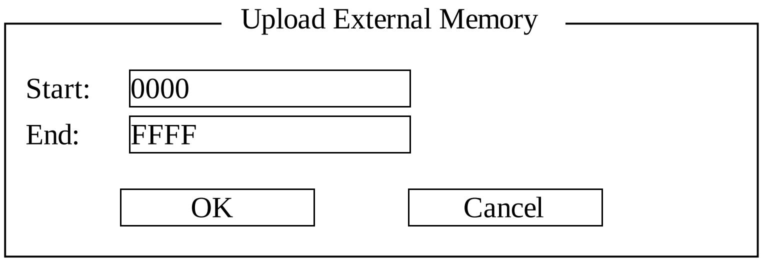
Buffer End: 0000

Enter the Target Start address where the data will reside in the target application. This address does not have to match the Buffer Start address, which is the beginning address for the data to be downloaded to the target. The Buffer End address provides the lower bound of the data to be downloaded to the target.
To accept the address range for download, press [Tab] to $\boxed{\text{OK}}$ and exit.

Upload External Memory

This command transfers a specified range of external memory from the target to the internal memory inside the emulator.

This upload is necessary in order to debug the program via stepping or setting breakpoints. Enter the Start and End addresses at the dialog box and click OK to exit.


Upload External Memory

Start: 0000

End: FFFF

NOTE:
When debugging codes uploaded from external memory, make sure to open the ASM window as RICE65 only supports disassembled code debugging for external memory.

2-5-7 New Directory

The New Directory command lets you set a new current drive and/or directory by prompting you for the new drive and/or directory name. The prompt is initialized to the current drive and directory. Enter any logical drive and directory path to overwrite the existing one. You can use this command to examine the current drive and directory settings.

2-5-8 Shell to DOS

The DOS shell command starts a DOS command processor so that you can type commands exactly as you were at the normal DOS prompt. When you have finished entering DOS commands, type "**exit** " to return to the ICE program.

The memory for your program image is still resident in the RAM. Therefore, there may not be enough memory to run another large program.

2-5-9 Quit to DOS

The Quit to DOS command returns control to DOS. The memory for your program image is freed, and any open file handles are closed.

Chapter 2-6 View Menu

The View Menu has a number of options for opening windows to display different information of the program you are debugging. You can also find the Search function where you can specify a string and its format and the window to be searched.

2-6-1 Program Memory Window

Displays the instruction codes in program memory in hex and ASCII (display only) formats. You can change the hex value of the registers by just typing over it. To do so, first press [F4] or [Tab] until Program Memory is the current window; then use the arrow keys to highlight the field to be edited and type in the new value. The new values will be updated in the Source Window, ASM Window and in the Emulator.

The Program Memory Window can be enlarged to display more registers. Before doing so, it is recommended to change the display to 43/50 line under Options | Display options.

```
                    Program Memory

0000-00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00 ................
0010-48 65 6C 6C-6F 20 57 6F-72 6C 64 21-20 20 20 00 Hello World!   .
0020-00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00 ................
0030-00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00 ................
0040-00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00 ................
```

The data in the Program Memory Window can be either be from the internal program memory buffer or external memory, depending on the definition under the Options | Set Memory Range... menu. In order to be more responsive during stepping, only the following portions of the entire 64K memory will be updated

- data in range 0-1FF
- data in the visible range in the Program Memory Window, this depends on the size of the window
- data within the “Update Memory Range” defined under the Options | Set Memory Range menu.

2-6-2 Source Window

The Source Window command opens a window to show the source file of the debugged program. If there is no valid debug file (see Appendix A), this window shows disassembled codes rather than the source listing.

In the following source window, you will find breakpoint (B) and trigger (T) identifiers, line number, program counter and opcodes.

```
                  Source <C:\rice65\as.asm>

    00045                       int *iptr;
    00046
 .T 00047   D45F-08             main ( )
    00048                       {
    00049                       char error_flag;
    00050                       int int_flag;
    00051                       int i;
    00052
 .. 00053   D46F-A9 01              while  (1)
    00054                           {
 B. 00055   D476-A0 04                      error_flag=0x23;
 .. 00056   D47C-A0 02                      int_flag=0x45;
```

A triangular mark always points to the next program counter or executable statement as indicated by the PC field in the Register Window. Users can also scroll through the codes with the arrow keys and bring the cursor (highlight bar) to any desired location. To move between the pages, use the [PgUp], [PgDn], [Home] or [End] keys.

NOTE:
The Source Window will go blank when encountering an invalid program counter. When this happens, check if you have specified the correct device type and if the program flow is correct.

The window will also be blank if you are debugging codes uploaded from external memory. In such a case, open the ASM Window to see the referenced codes during debugging.

2-6-3 ASM Window

The ASM Window command opens a window, which displays disassembled codes when debugging in C. When first, opened, the window is blank and will be updated upon the next emulation step. The highlighted line in the ASM Window corresponds to the same instruction highlighted in the Source Window.

```
        ASM Code
C003-A9 FF      LDA   #FFh
C005-85 00      STA   00h
C007-A9 00      LDA   #00h
C009-85 02      STA   DIRP2
C00B-EA         NOP
C00C-85 70      STA   have_line
```

Open this window if you are debugging codes uploaded from external memory.

2-6-4 Register Window

Displays the current contents of the program counter, the Processor Status (PS) register, the Accumulator (A), the X and Y Index registers and the Stack Pointer (SP). You can change the hex value of the registers by typing over them. To edit the codes, use the arrow keys to highlight the field and type in the new value.

```
Register
PC C003
         NV1BDIZC
PS B5    10110101
A  FF    11111111
X  FF    11111111
Y  FF    11111111
SP FF
```

2-6-5 Stack Window

Displays the stack memory of the 65C02. This window can be sized to display the desired stack locations.

```
Stack
01E0-00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
01F0-00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

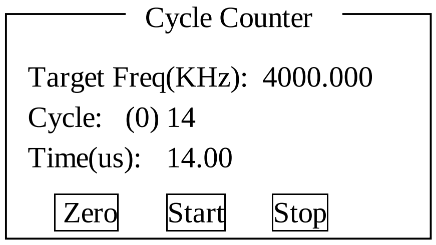
2-6-6 Watch Window

The Watch Window command opens a Watch window displaying the values of watch variables you defined using the Add Watch... Command. You can define watch variables only if a valid SYM file is available.

```
Watch
buf_index<0071>=0000
fail<C0AE>=46
have_line<0060>=00
```

2-6-7 Cycle Counter Window

The Cycle Counter Window displays instruction cycle and elapse time information. The execution time is based on the clock source specified under the OSC Frequency dialog box. You can reset the clock to zero, start or stop the clock anytime by clicking the corresponding button.

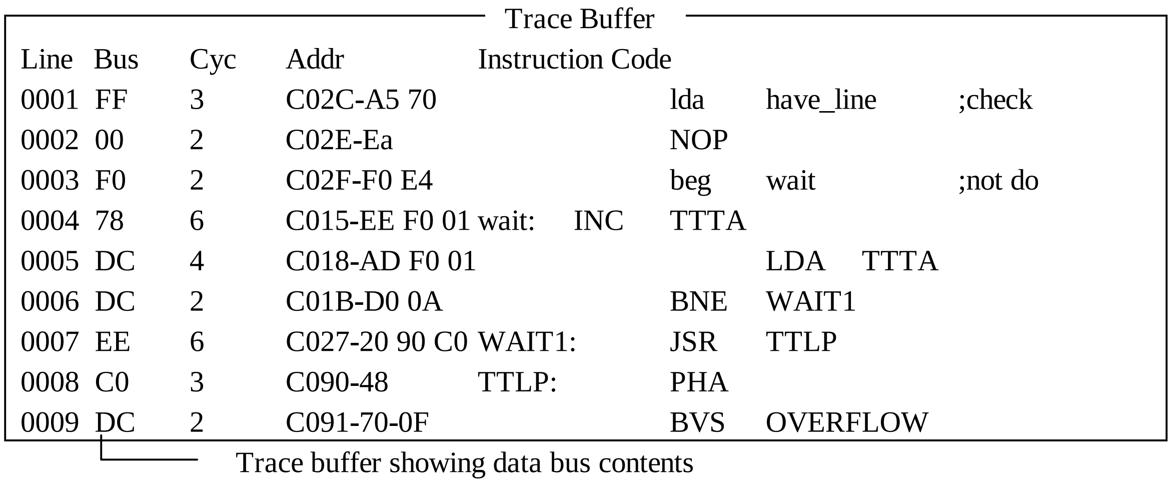


```
Cycle Counter

Target Freq(KHz):  4000.000
Cycle:   (0) 14
Time(us):   14.00

   Zero     Start     Stop
```

2-6-8 Trace Buffer

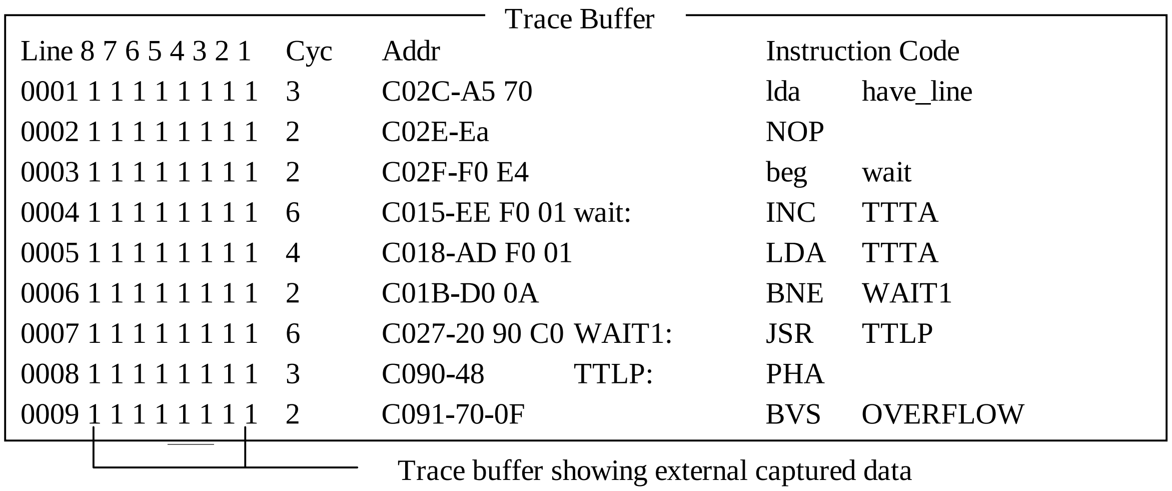
Displays the contents of the 8Kx24-bit circular trace buffer board which records the real-time executed instructions. The most recent instruction is found at the bottom of the buffer. The buffer contains:

- line number
- data bus content or captured data on the eight external probes,
- instruction cycle
- address
- opcode, and
- the source or disassembled code.



```
                         Trace Buffer
Line  Bus   Cyc   Addr              Instruction Code
0001  FF    3     C02C-A5 70                         lda     have_line       ;check
0002  00    2     C02E-Ea                            NOP
0003  F0    2     C02F-F0 E4                         beg     wait            ;not do
0004  78    6     C015-EE F0 01 wait:     INC        TTTA
0005  DC    4     C018-AD F0 01                              LDA     TTTA
0006  DC    2     C01B-D0 0A                         BNE     WAIT1
0007  EE    6     C027-20 90 C0 WAIT1:               JSR     TTLP
0008  C0    3     C090-48           TTLP:            PHA
0009  DC    2     C091-70-0F                         BVS     OVERFLOW
```

Trace buffer showing data bus contents



```
                         Trace Buffer
Line 8 7 6 5 4 3 2 1   Cyc   Addr                          Instruction Code
0001 1 1 1 1 1 1 1 1   3     C02C-A5 70                    lda      have_line
0002 1 1 1 1 1 1 1 1   2     C02E-Ea                       NOP
0003 1 1 1 1 1 1 1 1   2     C02F-F0 E4                    beg      wait
0004 1 1 1 1 1 1 1 1   6     C015-EE F0 01 wait:           INC      TTTA
0005 1 1 1 1 1 1 1 1   4     C018-AD F0 01                 LDA      TTTA
0006 1 1 1 1 1 1 1 1   2     C01B-D0 0A                    BNE      WAIT1
0007 1 1 1 1 1 1 1 1   6     C027-20 90 C0 WAIT1:          JSR      TTLP
0008 1 1 1 1 1 1 1 1   3     C090-48       TTLP:           PHA
0009 1 1 1 1 1 1 1 1   2     C091-70-0F                    BVS      OVERFLOW
```

Trace buffer showing external captured data

You can scroll through the buffer using the arrow keys or by clicking the vertical bar with a mouse. The contents can also be saved to a text file using the File | Save>>Trace Buffer... command.

Users can stop capturing data but keep the processor running once the buffer is full. This is achieved through the Break | Set Breakpoints... | More Break>>Stop on Trace Buffer Full function.

2-6-9 Verify Window

The Verify Window displays the verification errors when comparing the contents of the program memory inside the emulator against those in the memory buffer. If there is no error, the message “No Verify Error” will appear.

```
        Verify Window
 ADDR   DEVICE   BUFFER
 1004    00       01
 3F00    1A       AA
```

2-6-10 Search...

Unlike search functions from text editor, the Search function in RICE65 is based only on information available in the SYM file. So search pattern is limited to address labels, program counters, line numbers and instruction codes.

```
                             Search
 Search Pattern:                   (*) Hexadecimal integer
 [                   ]             ( ) Decimal integer
                                   ( ) ASCII string

 Search Window:          Search Key:
  (*) Source Window                (*) Program Counter address
  ( ) Trace Buffer Window          ( ) Line Number
  ( ) Program Memory Window        ( ) Address Label
                                   ( ) Instruction Code

          [  OK  ]                 [Cancel]
```

The Search dialog box, as shown here, allows users to specify a data string to be searched in the Source Window, Trace Buffer Window or Program Memory Window.

Input the data to be searched in the Search Pattern field and specify the format of the data: hex, decimal or ASCII. Then specify the window to be searched and key for the search. Tab to $\boxed{\text{OK}}$ to start the search.

Examples

Pattern	Format	Window	Key
103	Hex	Source	Program counter
0018	Decimal	Source	Line number
INIT	ASCII	Source	Label
0C96	Hex	Program memory	Instruction code

2-6-11 Next Match

The Next Match command searches for the next match of the search string you specified in the Search dialog box. This function applies to searching a repeated opcode inside the Trace Buffer and Program Memory Windows but not to labels.

Chapter 2-7 Run Menu: Emulator Functions

The RICE65 provides different emulation features for debugging your 65C02-based applications. This chapter describes these functions and goes through the different items under the Run Menu.

The Run Menu

The Run Menu contains different emulation control functions to execute your program. These frequently used options can all be accessed globally via hot function keys.

2-7-1 Reset Processor

The Reset Processor command resets the processor hardware. During the RESET conditions, the interrupt mask flag is set, the decimal mode is cleared and the program counter is loaded with the restart vector from location FFFC (low byte) and FFFD (high byte).

2-7-2 Go from Reset

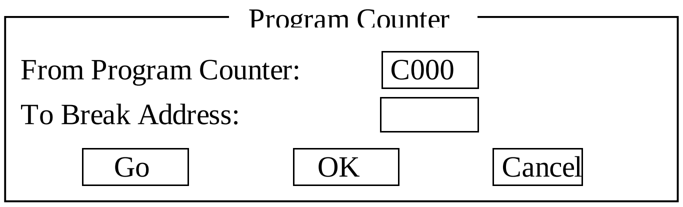
The Go from Reset command executes your program from the reset condition continuously until either a breakpoint is encountered, or the program is interrupted with a keyboard or mouse entry.

2-7-3 Go

The Go command executes your program from the current Program Counter continuously until either a breakpoint is encountered, or the program is interrupted with a keyboard or mouse entry.

2-7-4 From…

The From... command executes your program from the current Program Counter or new program counter until it reaches the temporary breakpoint you specified in the dialog box. Note that the address you specify may never be reached, depending on program logic.


Program Counter
From Program Counter: C000
To Break Address:

2-7-5 Go from Cursor

The Go from Cursor commands executes your program from the instruction line at the cursor position.

To select the new instruction line, press [F4] until the Source Window is active or click inside the window. Click on the selected line or use the Up/Down arrow key to bring the highlight bar to the new line to start the processor. To start running from this new program counter, press [Shift]-[F7].

2-7-6 To Cursor/Here

The To Cursor/Here command executes your program up to the currently highlighted source or instruction line. Note that the address you specify may never be reached, depending on program logic.

2-7-7 Single Step

The Single Step command executes a single instruction statement of your program. So if you press [F8] three times, RICE65 will execute three statement. Single stepping allows you to examine register data after each instruction.

If you are stepping codes uploaded from external memory, make sure to open the ASM Window to display the referenced codes.

2-7-8 ASM Single Step

The ASM Single Step command executes a single machine instruction of your program. This is the same as single step if your source code is in assembly language. When debugging in C, you can easily switch between single step (in C) and ASM step by using the [F8] and [Sift]-[F8] keys.

2-7-9 Step over Call

The Step over Call command executes a single statement of your program and skips over any function call. If the next executable statement is a function CALL instruction, pressing [F9] will step over the subroutine being called (but fully executing it) and return to the line of the function.

This command is useful if you have stepped through enough of a function to determine that it is working properly, and you wish to execute the rest of it without interruption.

2-7-10 Return to Caller

The Return to Caller command is the same as single stepping except when inside a procedure or function, this command will execute the rest of the procedure and return to the instruction of the function.

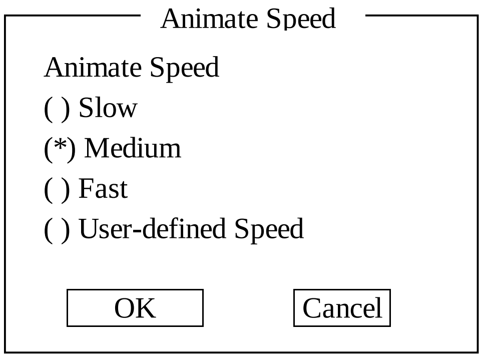
2-7-11 Animate

The Animate command is a self-repeating Single Step command. Instructions are executed continuously until a key is pressed. The RICE65 program displays changes to reflect the current program state between each trace, which allows you to watch the flow of control in your program.

2-7-12 Animate Speed...

The Animate Speed... command prompts you for the □ate of delay at which instructions step. Select the corresponding radio button for the speed desired or select user-defined speed. Press [Tab] to OK and exit.

If you select user-defined speed, RICE65 will bring up a dialog box. Input the number from 0.1 to 99.9. Again, press [Tab] to OK and exit.



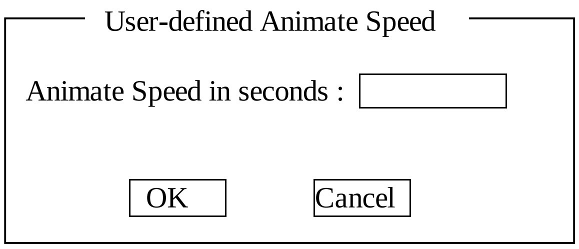
Animate Speed

Animate Speed
( ) Slow
(*) Medium
( ) Fast
( ) User-defined Speed

The relative speed on the different animate speed is given as follows:

Slow	~ 1.5 seconds
Medium	~ 0.5 seconds
Fast	~ 0.1 seconds
User-defined	~ 0.1-99.9 seconds

When user-defined speed is selected, enter the relative delay time from 0-99 in the dialog box as shown below. The actual speed depends on the speed of the PC host.



User-defined Animate Speed

Animate Speed in seconds :

Chapter 2-8 Break Menu: Breaks and Triggers

This chapter describes the breakpoints system within RICE65 and goes through the different functions inside the Break Menu, including breakpoints, cycle counter and trigger breaks.

Breakpoint defines a condition in which the processor executes the user's code and halts after a certain condition is met. RICE65 supports the following types of breakpoints:

2-8-1 Break on Address Match

Users can halt the processor at a specific or a range of program address within program memory.

2-8-2 External Break Input Signal

RICE65 has a 12-clip Logic Probe. One of the clips is an External Break Input. The RICE65 software can be programmed to break when the External Break Input line sees a rising or falling edge. If enabled, the processor will halt whenever the external break signal is latched. Users can specify to latch the External Break Input on one of the followings:

- Falling Edge
- Rising Edge

To specify the External Break Trigger, go to the Break | Set Breakpoints>More Break>> dialog box.

2-8-3 Break when the Trace Buffer is Full

RICE65 has an optional 8Kx24-bit ”circular” trace buffer, which logs real-time data. When the buffer is full, new data will continuously be logged to the buffer starting at the very beginning. Users can specify to halt the processor whenever the trace buffer is full, that is, when the trace buffer has captured 8K data. This break will automatically be in force when forward trace option is enabled.

2-8-4 Multiple Break

Found under Set Breakpoints>More Break>> menu, this allows users to halt the processor after an instruction at a set breakpoint has been executed a particular number of times.

2-8-5 Data Capture Break

Found under the Break | Data Capture Break... menu, this function allows the processor to halt when the value in the captured address matches the specified value. A mask value can also be entered for special bit comparison.

NOTE: When the oscillator speed is above 1MHz, the Multiple Break and Data Capture Break may execute several more instructions prior to breaking.

2-8-6 Break when Writing to Read-Only Memory

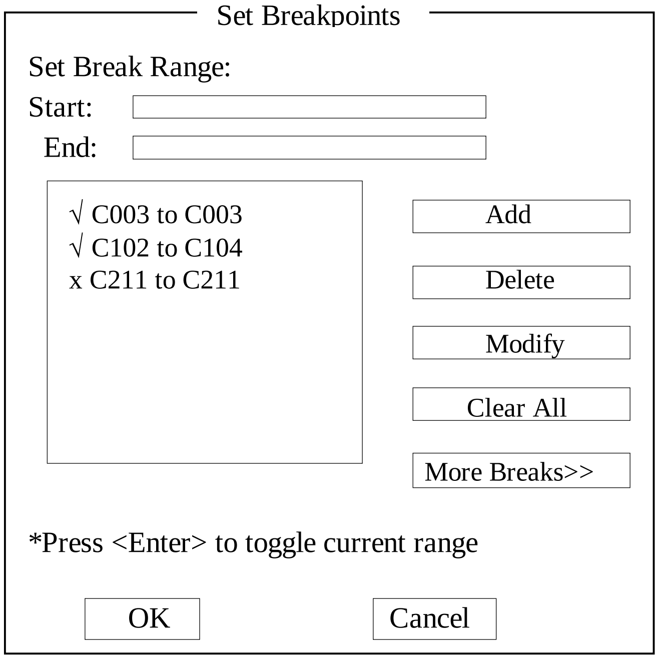
Once the user has defined an address range for read only memory, the processor will halt whenever the referenced address is being written to.

The Break Menu

Access the Break Menu at any time by pressing the [Alt]-[B] hot key. The menu has several options to let you add, delete, modify, toggle and clear breakpoints. Set breakpoints are marked with a "B" rather than a "." in the first column of the Source Window. You can also enable External Break Input via the Break Menu.

2-8-7 Set Breakpoints...

This command leads to the Set Breakpoint" dialog box which allows you to add, delete, modify or clear all breakpoints. Each breakpoint can be a specific address or an address range. Enter the address range in the Start and End fields, tab to the $\boxed{\text{Add}}$ button and press [Enter] to add. The addresses will appear in the list box.


Set Breakpoints

Set Break Range:

Start:

End:

√ C003 to C003
√ C102 to C104
x C211 to C211

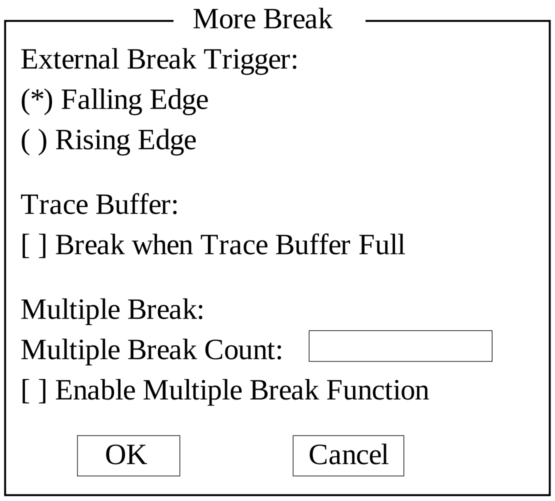
*Press <Enter> to toggle current range

To disable a set break address, press [Tab] to highlight the list box. Use the up/down arrow key to highlight the address range and press [Enter]. The "√" mark will turn into "x", indicating the disabled status. To delete a set break address, tab to the list box, use the up/down arrow key to highlight the range and press [Tab] to $\boxed{\text{Delete}}$ and then press [Enter]. The address will be removed from the list box.

To modify an address range, highlight the range in the list box. Press [Tab] to Start and End fields and input the new values. Press [Tab] again to $\boxed{\text{Modify}}$, press [Enter] and the specified range will be updated.

2-8-8 Setting Other Breaks

Clicking the More Break >> button in the "Set Breakpoint" dialog box will bring up the "More Break" dialog box as shown below.



More Break

External Break Trigger:
(*) Falling Edge
( ) Rising Edge

Trace Buffer:
[ ] Break when Trace Buffer Full

Multiple Break:
Multiple Break Count:
[ ] Enable Multiple Break Function

OK Cancel

For Multiple Break Count, you can enter any integer from 1- 2047.

The More Break dialog box allows you to set up the External Break condition, either on rising or falling edge, as well as to enable break upon trace buffer is full. It also allows you to specify the multiple break count. Use the [Tab] key to select groups and within groups, use the up/down arrow keys to select option. To confirm the selection, press the [Spacebar].

NOTE: For multiple breaks, the program may overrun some extra instructions and not break at the exact location when the oscillator frequency is over 1Mhz.

2-8-9 Toggle Breakpoint

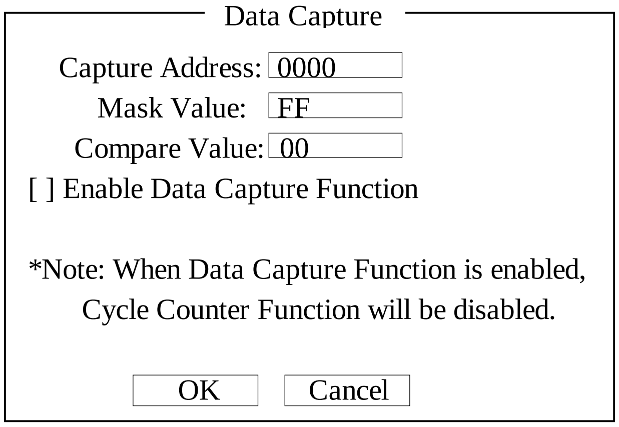
The Toggle Breakpoint command provides a quick and easy way to set or clear a breakpoint without going through a dialog box. By simply pressing the [F2] key, users can set the currently highlighted source line in Source Window to be a break address. To clear a set breakpoint, press [F2] again. A "B" in the first column of the Source Window indicates set breakpoints.

2-8-10 Clear All Breakpoints

The Clear All Breakpoints command removes all previously defined breakpoints. Use this command when you want to continue debugging your program but no longer want it to stop at any previous set breakpoint locations.

2-8-11 Data Capture Break...

The Data Capture Break... command allows you to setup the real-time conditional break. If the data inside the Data Capture Address equals to the compare value ANDed with the mask value, RICE65 will generate a break to halt the processor.



Data Capture

Capture Address: 0000
Mask Value: FF
Compare Value: 00
[ ] Enable Data Capture Function

*Note: When Data Capture Function is enabled,
Cycle Counter Function will be disabled.

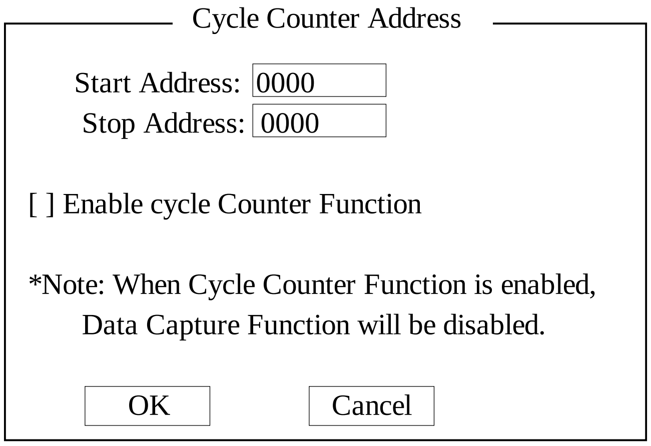
OK Cancel

Input the value in the Capture Address and press [Tab] to go to the next fields. Press [Spacebar] to enable the function and press [Tab] to $\boxed{\text{OK}}$ and exit.

$\boxed{\textbf{NOTE}}$**:** Since the Data Capture and Cycle Counter functions share the same hardware, the Cycle Counter Function will be disabled when the Data Capture Function is enabled and vice versa.

2-8-12 Cycle Counter Address

This sets the Start and Stop Address for the Cycle Counter and provides the execution time when running codes between the addresses. When the Start Address is reached, the cycle counter will be cleared and started until the Stop Address is encountered. the counter will then stop and display the elapse time in the Cycle Counter Window.



Cycle Counter Address

Start Address: 0000
Stop Address: 0000

[ ] Enable cycle Counter Function

*Note: When Cycle Counter Function is enabled,
Data Capture Function will be disabled.

OK Cancel

2-8-13 Set Trigger Points...

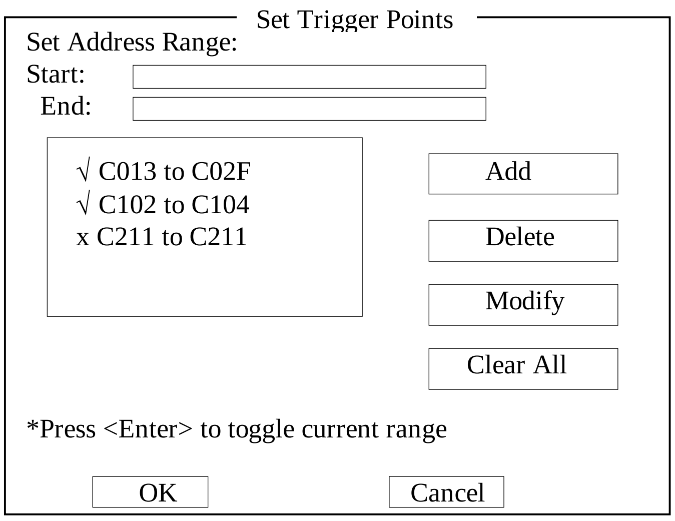
The Set Trigger Points... command allows you to setup the trigger output address ranges. When an instruction is qualified for trigger output, the trigger output line on the external logic probes generates a positive pulse. This line can be used to:

- trigger an oscilloscope when the processor executes an instruction at a particular address
- trigger other logic or instruments

- connect to the External Break Input (BRK) of another RICE65 to provide cross triggering on multiple emulators. In this case, the processor in the other emulator will halt.

NOTE: To halt the processors in both emulators, connect the Break Output line of Emulator1 to the External Break Input line of Emulator2. Emulator1 will halt and send a break signal to Emulator2, halting its processor.

This command leads to the "Set Trigger Points" dialog box. It allows users to input a specific address or an address range for trigger outputs. Enter the address range in the "Start" and "End" fields, tab to the Add button and press [Enter] to add. The address will appear in the list box. To disable a trigger address, press [Tab] to highlight the list box. Use the up/down arrow key to highlight the address range and press [Enter]. The "√" mark will turn into "x", indicating the disabled status.


Set Trigger Points

Set Address Range:

Start:

End:

√ C013 to C02F
√ C102 to C104
x C211 to C211

*Press <Enter> to toggle current range

To delete a trigger address, tab to the list box, use the up/down arrow key to highlight the range and press [Tab] to Delete and then press [Enter]. The address will be removed from the list box.
To modify an address range, highlight the range in the list box. Press [Tab] to the "Start" and "End" fields and input the new values. Press [Tab] again to Modify press [Enter] and the specified range will be updated.

It is much easier to move around the dialog box with a mouse. Just click on any item with the left mouse button instead of pressing [Tab] to cycle through the different elements. Trigger points are indicated by a "T" rather than a "." in the second column of the Source Window.

2-8-14 Using Trigger for Real-time Trace

The Set Trigger Points can also be used with the Real-time Trace function to save interested address ranges to the trace buffer. Set the address in the Set Trigger Points dialog box as described and then go to Trace | Set Trace Range and enable "Real-time Trace".

2-8-15 Clear All Triggers

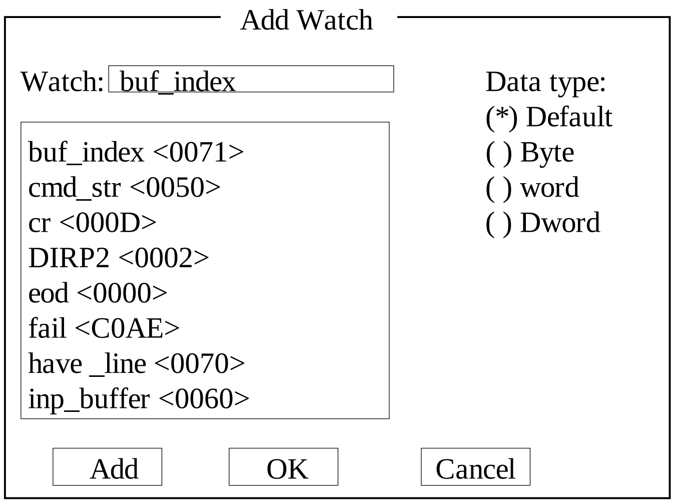
The Clear All Triggers command removes all previously set trigger points. All trigger output indicators "T" in the Source Window will turn to ".".

Chapter 2-9 Watch Menu

The Watch Menu allows you to examine specific symbol variables in your program. The same variables can also be viewed inside the Program Memory Window but the Watch function will single them out to be observed in a different window.

2-9-1 Add Watch

The Add Watch... command places the variable on the watch list displayed in the Watch Window and continuously updates its value. The command leads to an Add Watch dialog box where you can select available variables from a symbol list to watch.



Watch: buf_index

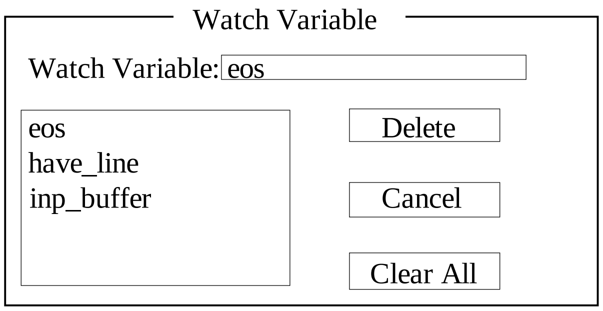
```
buf_index <0071>
cmd_str <0050>
cr <000D>
DIRP2 <0002>
eod <0000>
fail <C0AE>
have _line <0070>
inp_buffer <0060>
```

Data type:
(*) Default
( ) Byte
( ) word
( ) Dword

First, use the Up/Down arrow key to highlight the variable to be watched. Press [Tab] to specify data type and then press [Tab] to Add. Repeat the same procedure to specify more variables. When ready, press [Tab] to the OK button and exit.

2-9-2 Delete Watch

The Delete Watch.. command leads to the dialog box where you can remove watch variables from the Watch Window. Highlight in the list box the variable to be removed, press [Tab] to go to the Delete button and press [Enter]. To clear all the watch variables, press [Tab] to highlight the Clear All button and press [Enter].



Watch Variable: eos

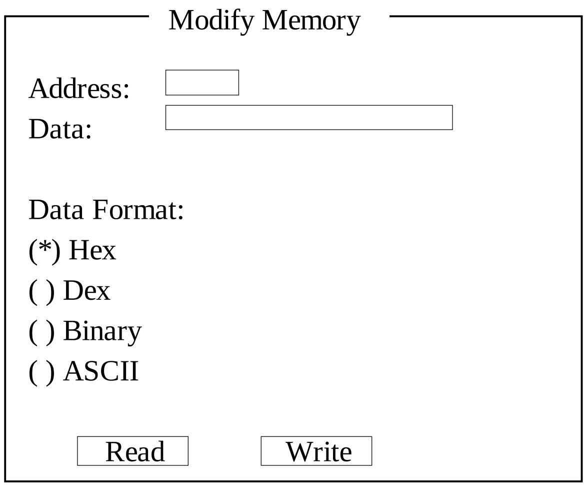
```
eos
have_line
inp_buffer
```

2-9-3 Clear All Watches

The Clear All Watches command removes all watch variables from your program. The contents of the Watch Window will be cleared once the command is issued.

2-9-4 Modify Memory

The Modify Memory command allows you to read (inspect) or write (modify) the contents of Program Memory. It leads to the dialog box as shown below.



Modify Memory

Address:

Data:

Data Format:

(*) Hex

( ) Dex

( ) Binary

( ) ASCII

To read (inspect) a data:

1. Enter the address in hex value at which data is to be inspected.
2. Select data format by highlighting the corresponding radio button.
3. Press [Tab] to the $\boxed{\text{Read}}$ button and press [Enter].
4. The value will be displayed in the “Data” field in the dialog box.

To write (modify) a data:

1. Enter the address in hex value at which data is to be modified.
2. Input the new data to be written.
3. Specify memory type by highlighting the respective radio button.
4. Select data format by highlighting the corresponding radio button.
5. Press [Tab] to the $\boxed{\text{Write}}$ button and press [Enter].
6. The new value will be updated in the corresponding window.

The Modify function can also be achieved by typing the new value directly in the appropriate window.

Chapter 2-10 Trace Menu

Commands under the Trace Menu allow you to read the contents, set begin trace address or clear the optional trace buffer.

The 8K by 24-bit trace buffer is a circular buffer which continuously captures the following data, with the most recent information stored at the bottom of the buffer.

- line number
- data bus content or captured data on the eight external probes (via the Options | Trace Options menu),
- instruction cycle
- address
- opcode, and
- the source or disassembled code (via the Options | Trace Options menu).

The trace buffer will automatically records the above data as program codes are executed. You can also specify special tracing under the Trace | Set Trace Range... Menu. Working together with the breakpoint system, the trace buffer contains valuable information for debugging.

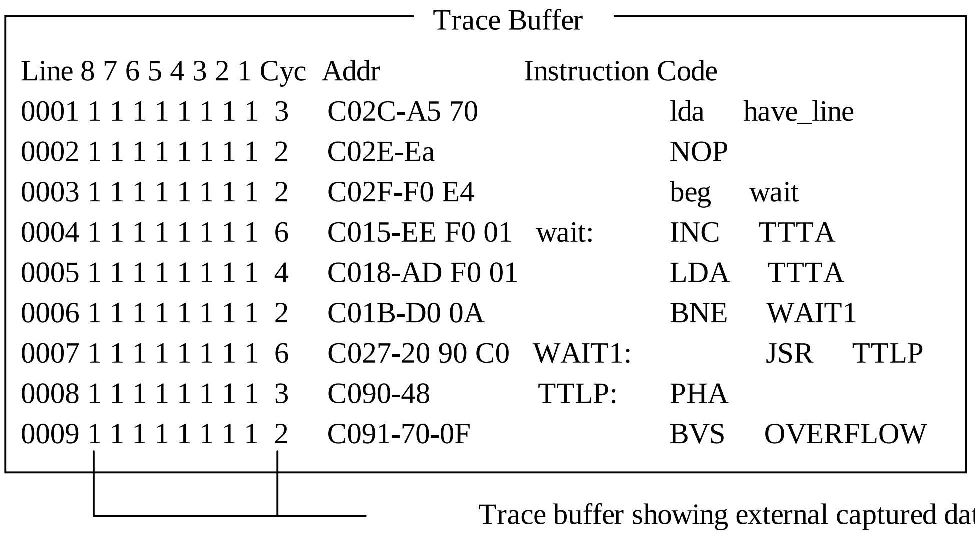
2-10-1 Read Trace Buffer

The Read Trace Buffer command uploads the trace buffer data from the RICE65 hardware and displays them in the Trace Buffer Window. The circular trace buffer can hold up to 8Kx24-bit of information and 8K instructions. The last executed line is highlighted with a “>“ mark for easy reference.

The Trace Buffer can be set to record either Data Bus Contents or logic states captured from the eight external probes as illustrated below:

```
                         Trace Buffer

Line  Bus   Cyc   Addr          Instruction Code
0001  FF    3     C02C-A5 70                   lda    have_line      ;check
0002  00    2     C02E-Ea                      NOP
0003  F0    2     C02F-F0 E4                   beg    wait           ;not do
0004  78    6     C015-EE F0 01        wait:          INC    TTTA
0005  DC    4     C018-AD F0 01                       LDA    TTTA
0006  DC    2     C01B-D0 0A                   BNE    WAIT1
0007  EE    6     C027-20 90 C0 WAIT1:         JSR    TTLP
0008  C0    3     C090-48       TTLP:          PHA
0009  DC    2     C091-70-0F                   BVS    OVERFLOW
```

Trace buffer showing data bus contents



```
                         Trace Buffer
Line 8 7 6 5 4 3 2 1 Cyc  Addr              Instruction Code
0001 1 1 1 1 1 1 1 1  3   C02C-A5 70                  lda    have_line
0002 1 1 1 1 1 1 1 1  2   C02E-Ea                     NOP
0003 1 1 1 1 1 1 1 1  2   C02F-F0 E4                  beg    wait
0004 1 1 1 1 1 1 1 1  6   C015-EE F0 01   wait:       INC    TTTA
0005 1 1 1 1 1 1 1 1  4   C018-AD F0 01               LDA    TTTA
0006 1 1 1 1 1 1 1 1  2   C01B-D0 0A                  BNE    WAIT1
0007 1 1 1 1 1 1 1 1  6   C027-20 90 C0   WAIT1:             JSR    TTLP
0008 1 1 1 1 1 1 1 1  3   C090-48          TTLP:      PHA
0009 1 1 1 1 1 1 1 1  2   C091-70-0F                  BVS    OVERFLOW
```

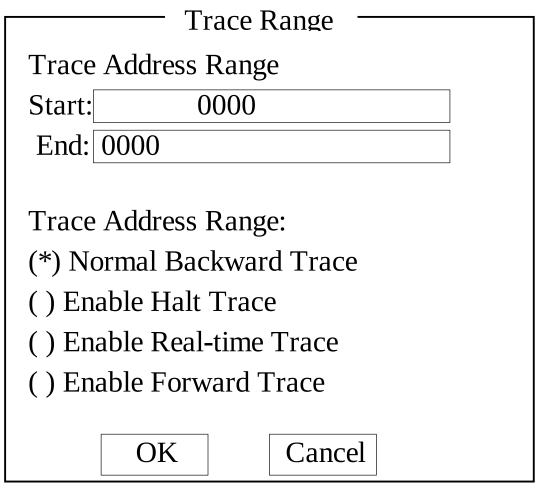
Trace buffer showing external captured data

2-10-2 Clear Trace Buffer

The Clear Trace Buffer command clears the contents of Trace Window and reset the trace pointer.

2-10-3 Set Trace Range...

The Set Trace Range... command lets you set the program counter address to begin capturing real-time data in the trace buffer. You are prompted for the Start and End trace address in the pop-up dialog box.



```
        Trace Range
Trace Address Range
Start:      0000
 End: 0000

Trace Address Range:
(*) Normal Backward Trace
( ) Enable Halt Trace
( ) Enable Real-time Trace
( ) Enable Forward Trace

      OK        Cancel
```

◆ **Normal Backward Trace**

This is the **default** trace mode, saving all executed instructions in the 8K buffer and throwing away old ones. The last executed instruction will be found at the bottom of the buffer. It is not necessary to input an address range for this trace.

- ### Enable Halt Trace

You are required to enter the Start and End address for this option. The emulator will clear the buffer upon reaching the Start, start-capturing data until the End Address at which a breakpoint has been set internally. To view the data, use the Upload Trace command.

- ### Enable Real-time Trace

In this option, the processor will keep running while data within the address range will be repeatedly captured to the buffer. You can also enter multiple address ranges to be traced. To do so, go the Break | Set Trigger Points Menu and select the addresses as trigger address and then enable "Real-time Trace" in the Trace | Set Trace Range Menu.

- **Enable Forward Trace**

The Forward Trace will start capturing data from the Start address and the processor will break if the trace buffer is full. This prevents the Start Address from being overwritten by the circular buffer.

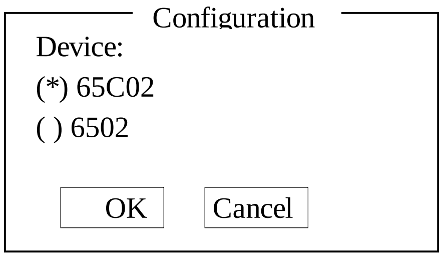
Note that if the Start Address is within a loop and keeps repeating, only one instance will be saved to the trace buffer. The processor will keep running. In such case as the buffer is not full to cause a break.

Chapter 2-11 Config Menu

The Config Menu provides functions that allows users to set device configuration, oscillator source, the frequency of the internal clock, as well as to override the auto-detect parallel port address for communication.

2-11-1 Configuration...

The Configuration... command leads to the “Configuration” dialog box where users can specify the device to be emulated.



Configuration

Device:

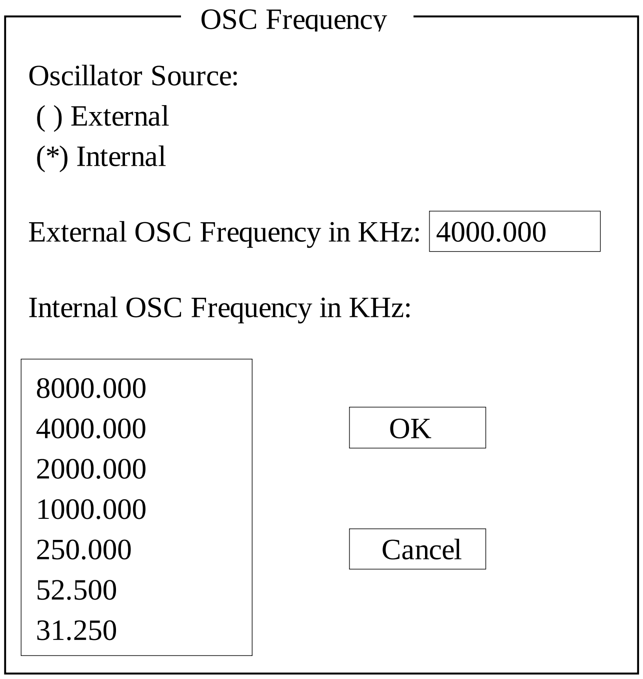
(*) 65C02

( ) 6502

If you have a mouse, just click the item you want to choose. Select OK to accept the settings and Cancel to cancel the changes.

2-11-2 OSC Frequency

The OSC Frequency... command allows users to specify the source of the oscillator for emulation.



OSC Frequency

Oscillator Source:

( ) External

(*) Internal

External OSC Frequency in KHz: 4000.000

Internal OSC Frequency in KHz:

8000.000
4000.000
2000.000
1000.000
250.000
52.500
31.250

When external source is selected, users need to connect a crystal or provide an oscillator input to the “EXT XTAL” socket on the RICE65 motherboard (please refer to page 6). The frequency input is for the cycle counter to calculate elapse time.

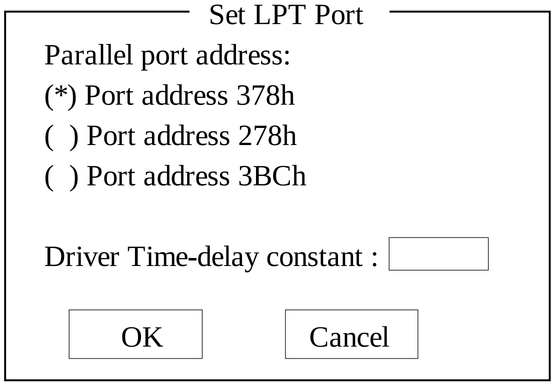
For internal source, users can select from a range of available frequencies, from 31.25KHz to 8MHz as show in the list box.

Within the dialog box, use the up/down arrow key to select oscillator source and press [Spacebar] to confirm. To specify the frequency for the internal source, press [Tab] to go to the list box and use the up/down arrow key to scroll through the list and then press [Enter] to select the frequency and exit.

If you have a mouse, just click the item you want to choose. Select $\boxed{\text{OK}}$ to accept the settings and $\boxed{\text{Cancel}}$ to cancel the changes

2-11-3 LPT Port...

The LPT Port... command lets you override the default setting on the parallel port address detected by the RICE65 program. You can specify 378h, 278h or 3BCh. Use the arrow key to pick an alternate port address and press [Spacebar] to confirm. Tab to $\boxed{\text{OK}}$ and exit.



Set LPT Port

Parallel port address:

(*) Port address 378h

( ) Port address 278h

( ) Port address 3BCh

Driver Time-delay constant :

An incorrect LPT port will cause communication error and incorrect emulation results. If you get a warning message, make sure:

- the parallel adapter cable is properly connected to the parallel port and to the RICE65 unit
- power is supplied to the RICE65 unit which undergoes the power-up self test properly (the LED blinks several times)

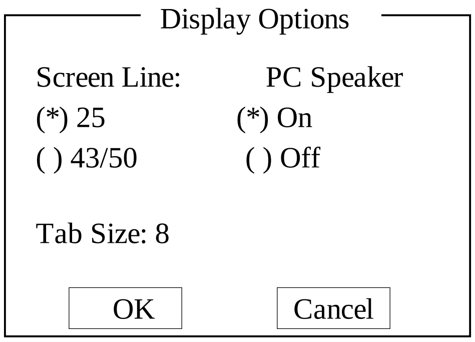
The Driver Time-delay constant enables RICE65 to work with fast PC if it fails communication. In such a case, enter a constant from 5 to 50, in increments of 5 and then 100 to 500, in increments of 50. If the RICE65 fails to communicate with your PC, please contact our support teams.

Chapter 2-12 Options Menu

The Options Menu allows you to change the screen display of the RICE65 software, specify update memory range, external memory range and read-only memory range, and to save and restore the RICE65 program setup.

2-12-1 Display Option

The Display Option... command allows you to select the number of lines to display on screen, the setting of the PC speaker and to specify the tab size in the Source Window. The tab size is a decimal field and you can enter a value up to 20. The default setting is 8.



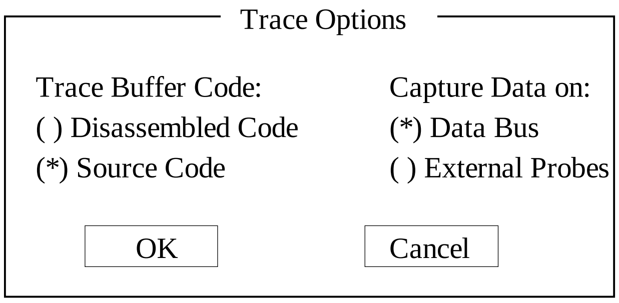
```
Display Options

Screen Line:        PC Speaker
(*) 25              (*) On
( ) 43/50            ( ) Off

Tab Size: 8
```

NOTE:

The System | Restore command does not reset the screen line to 25. It only restores the screen to the way the program starts, that is, with the Source, Program Memory and Register as the three open windows.

2-12-2 Trace Option...

The Trace Option... command allows you to specify if the trace buffer would contain disassembled or source codes and the capture data from the Data bus or from the 8 external probes.

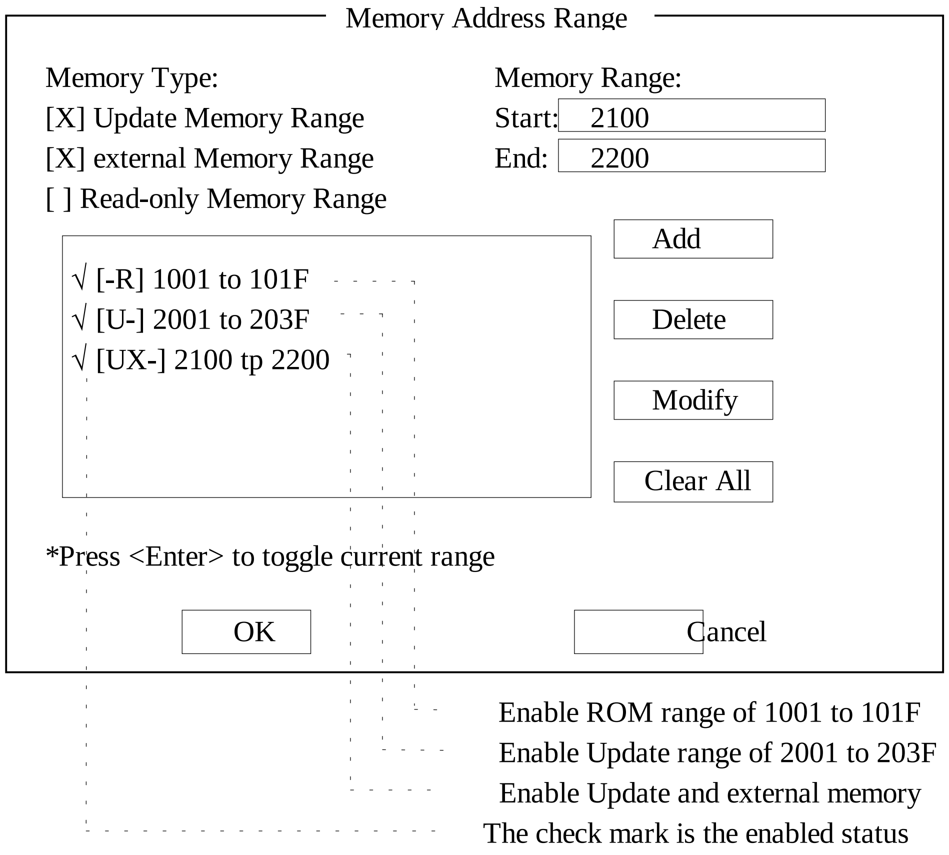


```
Trace Options

Trace Buffer Code:           Capture Data on:
( ) Disassembled Code        (*) Data Bus
(*) Source Code              ( ) External Probes
```

To select, press [Tab] to select among the two groups and the buttons. Within each group, use the Up/Down arrow keys to select and press [Spacebar] to confirm. Once selection is made, press [Tab] to $\boxed{\text{OK}}$ and press [Enter] to exit.

2-12-3 Set Memory Range

This command sets the different memory range for debugging. It leads to the dialog box as shown below.



Memory Address Range

Memory Type:
[X] Update Memory Range
[X] external Memory Range
[ ] Read-only Memory Range

Memory Range:
Start: 2100
End: 2200

√ [-R] 1001 to 101F
√ [U-] 2001 to 203F
√ [UX-] 2100 tp 2200

*Press <Enter> to toggle current range

Enable ROM range of 1001 to 101F
Enable Update range of 2001 to 203F
Enable Update and external memory
The check mark is the enabled status

To set Memory Range, enter the Start and End address in the Edit box, click the desired Memory Type and click the $\boxed{\text{Add}}$ button. The Memory Type is initialized as **U**, **X** and **R** in the list box.

In the dialog box above, address 1001 to 101F is set as Read-only memory while address 2100 and 2200 is set as both Updated Memory and External Memory. That is, RICE65 will fetch data from external memory for range 2100 and 2200 and also update their value in the Program memory window for each single step. To modify memory range, highlight the range in the list box, click the different Memory Type and/or change the Start/End address. When done, click the $\boxed{\text{Modify}}$ button.

To temporarily disable a range, highlight the range in the list box and press [Enter]. The check mark will change to a cross (x), indicating the disable status.

Update Memory Range

The Update Memory Range is the address range in program memory, which will be updated after each Run command. Besides the Update Memory range, RICE65 will also update the program memory from 0 to 0x1FF and the visible range in the Program Memory Window after each Stepping or Go command.

External Memory Rang

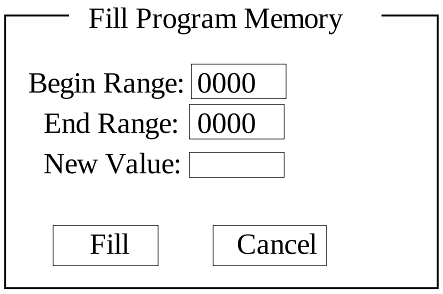
The External Memory Range is the address range of the external memory, such as I/O mapping memory. If the external memory range is set, RICE65 will fetch the data from external memory for those address location.

Read-Only Memory Range

The Read-Only Memory Range is the address range in program memory to be read-only so that the data inside those address locations will not be destroyed.
If this is defined, RICE65 will halt when the program tries to write to this address range.

2-12-4 Fill Memory Range

This command writes an assigned value to a block of memory in the emulator.


Fill Program Memory

Begin Range: 0000
End Range: 0000
New Value:

Enter the Begin and End address of the memory block and then the new value, which is used to fill the locations. Press [Tab] to Fill and [Enter] to exit. The new value will be updated in the Program Memory Window if the range is with the Update Memory, from 0-1FF or within the visible range.

2-12-5 Save Setup...

The Save Setup... command saves the RICE65 debug environment settings to an editable text file. You are prompted for the filename. You can enter any name but are recommended to use the same root name as the corresponding object file with a .SET extension. This enables the setup to be restored whenever you load the object file to debug.

Information in the configuration setup file include:

- LPT port
- Processor information (device type, oscillator source and frequency)
- Program name
- Window layout
- Display options (screen line and tab size)
- Break settings

- Data Capture settings
- Trigger settings
- Watch Variables
- Update Memory, External Memory and Read-Only Memory.

2-12-6 Restore Setup

The Restore Setup... command allows you to load a setup file previously saved using the Options | Save Setup... command.

Appendix A

Using different 65C02 Assemblers and Compilers with RICE65

RICE65 currently supports source level debugging on assembler and C compiler from 2500AD. More assemblers/compilers will be supported in the future. Please contact Advanced Transdata or NOVATEK Microelectronics Corp. for more details.

2500AD Assembler/Compiler

To support source level debugging, the following files are required:

HEX, SYM, DCF and the ASM/C source files.

Code File Format	Intel Hex	.hex
Symbol Table	2500AD High Level	.sym
Debug Control File	High Level Debug Control File	.dcf

To assemble and link an assembly language source file, e.g. ATUTOR.ASM:

```
X6502 atutor.asm -ds
link atutor.lnk
```

To compile and link a C program, e.g. CTUTOR.C

```
c6502 -C -d ctutor.c
link ctutor.lnk
```

Both .lnk files have specified the options: map, high level, Intel hex. Please refer to the 2500AD manuals.

WDC/Zardox Assembler

For the WDC/Zardox assembler, RICE65 uses the WDC/Zardox symbol format for source level debugging. Make sure to use the following command when compiling:

```
ZAS -GLS <<filename.asm>>

ZLN -G -HI -SZ <<filename.obj>>
```

If only a hex file is available, the file will be disassembled and displayed in the Source Window.