Silicon (CMOS) construction in a variety of package types. An enhanced version of the 8051, the 8052, also exists with its own family of variations and even includes one member that can be programmed in BASIC. An inspection of Appendix E shows that there are dozens of other variations on the "core" 8051 architecture. This galaxy of parts, the result of desires by the manufacturers to leave no market niche unfilled, would require many chapters to cover. In this chapter, we will study a "generic" 8051, housed in a 40-pin DIP, and direct the investigation of a particular type to the data books. The block diagram of the 8051 in Figure 3.1a shows all of the features unique to microcontrollers:

Internal ROM and RAM
I/O ports with programmable pins
Timers and counters
Serial data communication

The figure also shows the usual CPU components: program counter, ALU, working registers, and clock circuits.[1]

The 8051 architecture consists of these specific features:

- Eight-bit CPU with registers A (the accumulator) and B
- Sixteen-bit program counter (PC) and data pointer (DPTR)
- Eight-bit program status word (PSW)
- Eight-bit stack pointer (SP)
- Internal ROM or EPROM (8751) of 0 (8031) to 4K (8051)
- Internal RAM of 128 bytes:
  - Four register banks, each containing eight registers
  - Sixteen bytes, which may be addressed at the bit level
  - Eighty bytes of general-purpose data memory
- Thirty-two input/output pins arranged as four 8-bit ports: P0–P3
- Two 16-bit timer/counters: T0 and T1
- Full duplex serial data receiver/transmitter: SBUF
- Control registers: TCON, TMOD, SCON, PCON, IP, and IE
- Two external and three internal interrupt sources
- Oscillator and clock circuits

[1] Knowledge of the details of circuit operation that cannot be affected by any instruction or external data, although intellectually stimulating, tends to confuse the student new to the 8051. For this reason, this text concentrates on the essential features of the 8051; the more advanced student may wish to refer to manufacturers' data books for additional information.

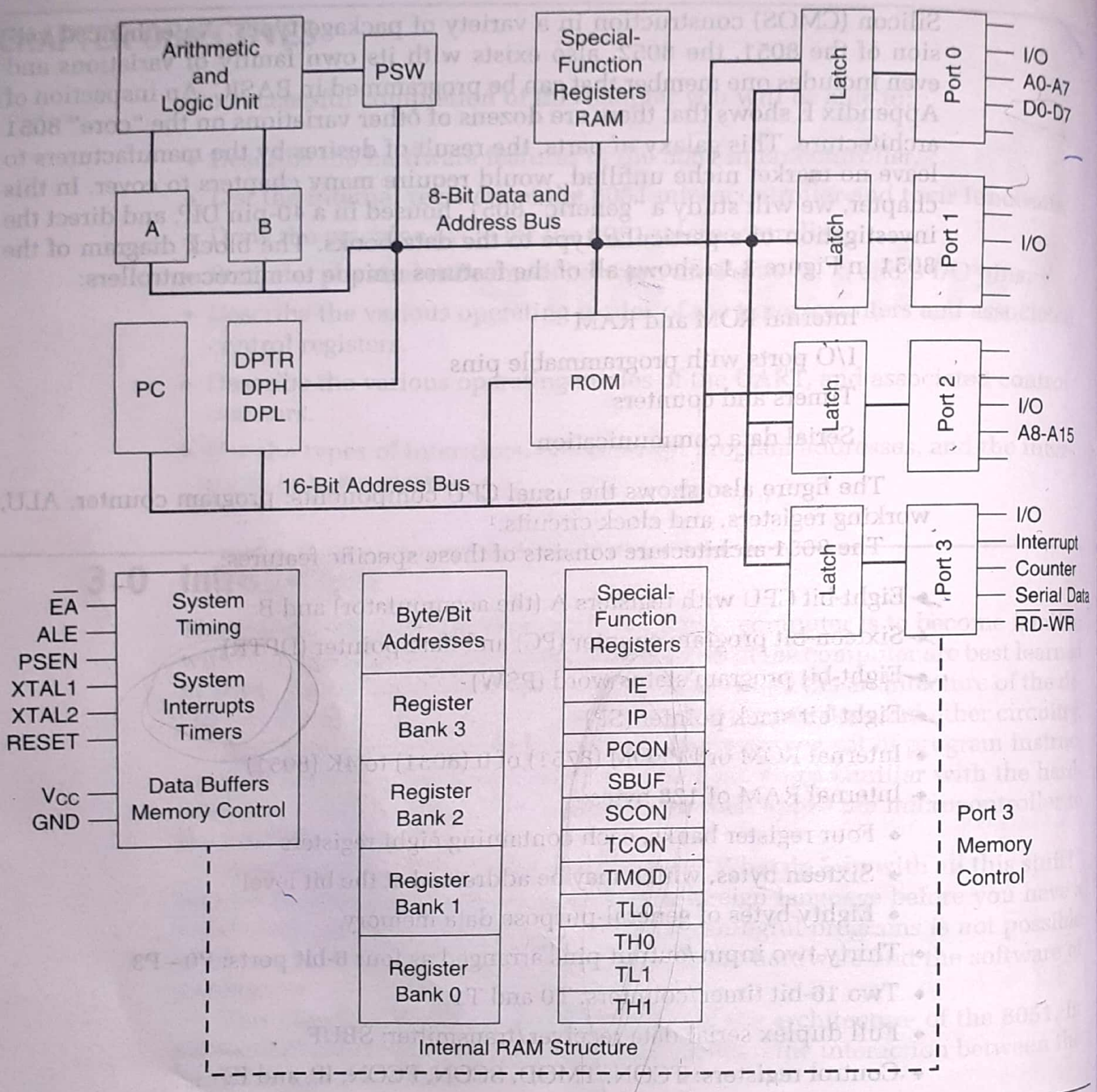
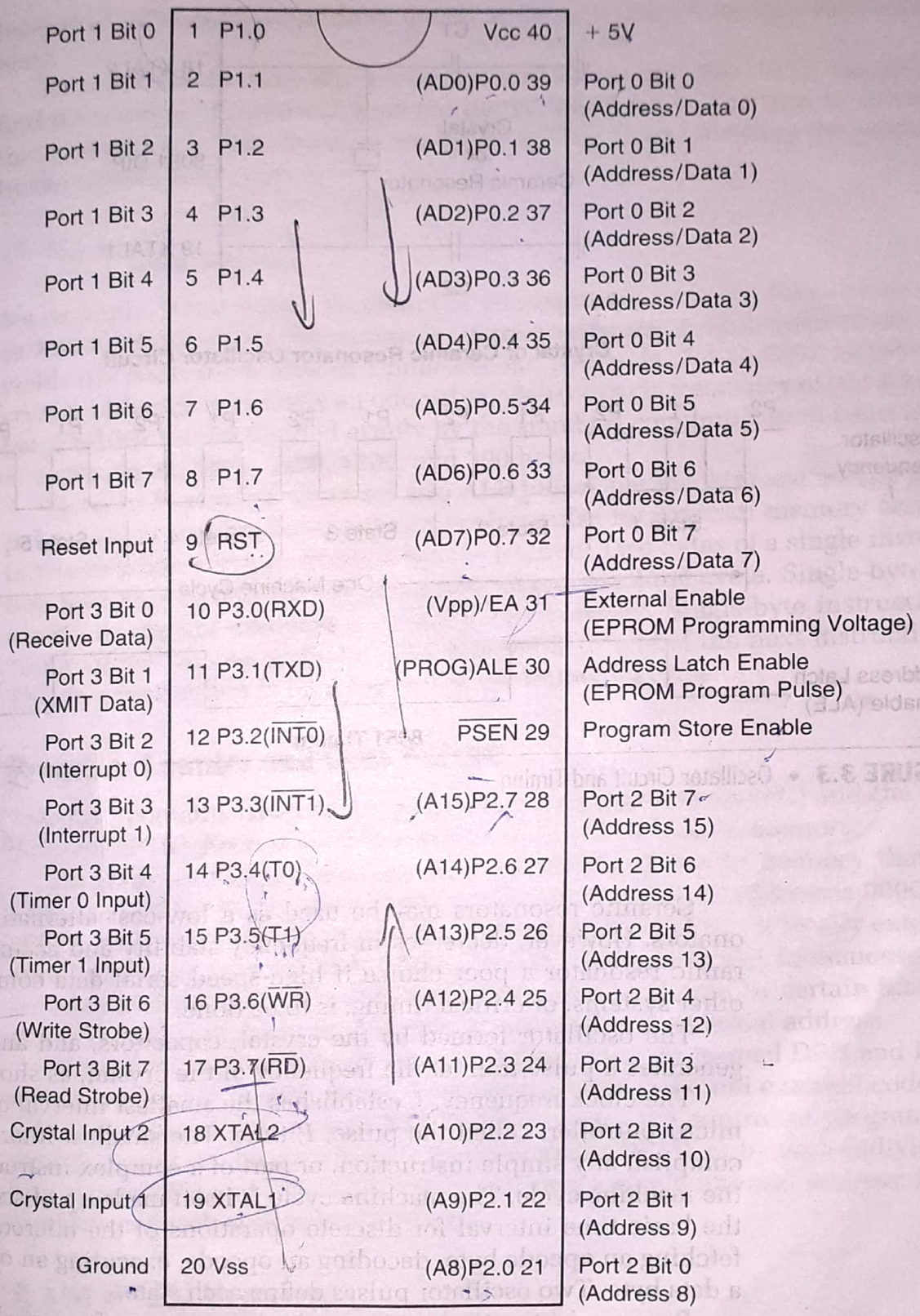



**FIGURE 3.1A** ◆ *8051 Block Diagram*

The programming model of the 8051 in Figure 3.1b shows the 8051 as a collection of 8- and 16-bit registers and 8-bit memory locations. These registers and memory locations can be made to operate using the software instructions that are incorporated as part of the design. The program instructions have to do with the control of the registers and digital data paths that are physically con-




Note: Alternate functions are shown below the port name (in parentheses). Pin numbers and pin names are shown inside the DIP package.

**FIGURE 3.2** ◆ 8051 DIP Pin Assignments

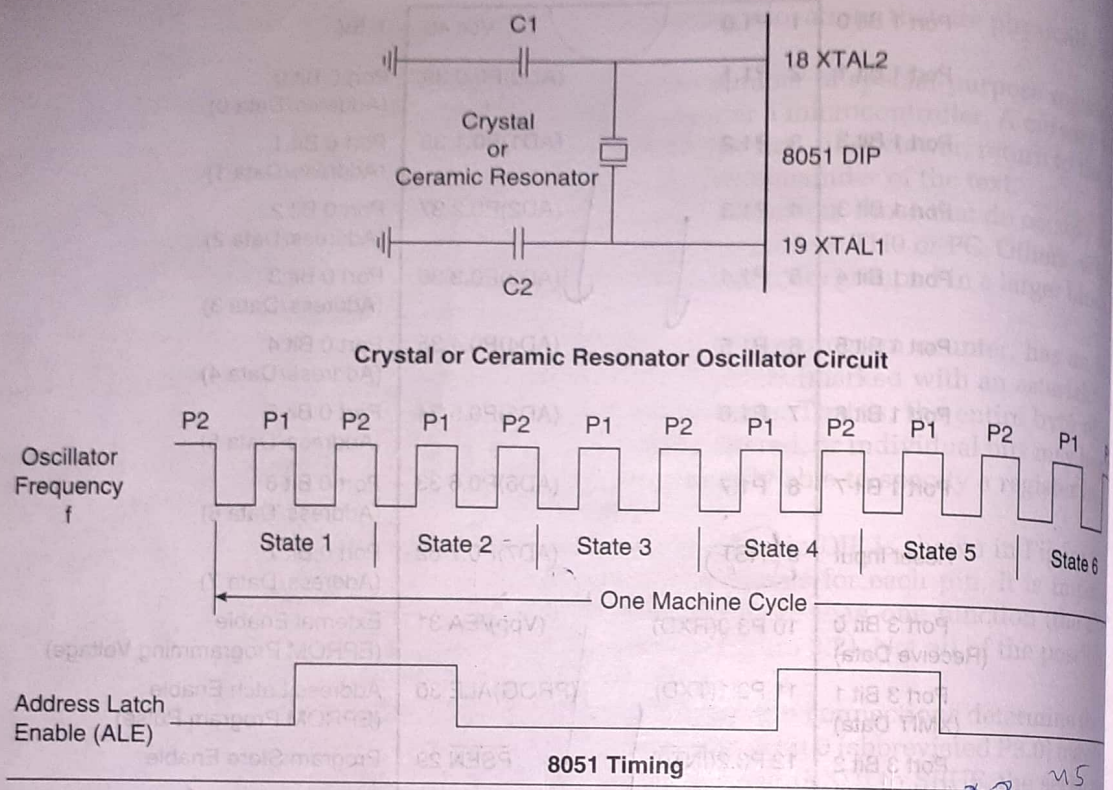



**FIGURE 3.3** ◆ Oscillator Circuit and Timing

Ceramic resonators may be used as a low-cost alternative to crystal onators. However, decreases in frequency stability and accuracy make th ramic resonator a poor choice if high-speed serial data communication other systems, or critical timing, is to be done.

The oscillator formed by the crystal, capacitors, and an on-chip inv generates a pulse train at the frequency of the crystal, as shown in Figur

The clock frequency, *f*, establishes the smallest interval of time within microcontroller, called the pulse, *P*, time. The smallest interval of time complish any simple instruction, or part of a complex instruction, howeve the machine cycle. The machine cycle is itself made up of six states. A st the basic time interval for discrete operations of the microcontroller su fetching an opcode byte, decoding an opcode, executing an opcode, or w a data byte. Two oscillator pulses define each state.

Program instructions may require one, two, or four machine cycles executed, depending on the type of instruction. Instructions are fetched executed by the microcontroller automatically, beginning with the instru

located at ROM memory address 0000h at the time the microcontroller is first reset.

To calculate the time any particular instruction will take to be executed, find the number of cycles, *C*, from the list in Appendix A. The time to execute that instruction is then found by multiplying *C* by 12 and dividing the product by the crystal frequency:

$$T_{inst} = \frac{C \times 12d}{\text{crystal frequency}}$$

For example, if the crystal frequency is 16 megahertz, then the time to execute an ADD A, R1 one-cycle instruction is .75 microseconds. A 12 megahertz crystal yields the convenient time of 1 microsecond per cycle. An 11.0592 megahertz crystal, although seemingly an odd value, yields a cycle frequency of 921.6 kilohertz, which can be divided evenly by the standard communication baud rates of 19200, 9600, 4800, 2400, 1200, and 300 hertz.

Note, in Figure 3.3, there are two ALE pulses per machine cycle. The ALE pulse, which is primarily used as a timing pulse for external memory access, indicates when every instruction byte is fetched. Two bytes of a single instruction may thus be fetched, and executed, in one machine cycle. Single-byte instructions are not executed in a half cycle, however. Single-byte instructions "throw-away" the second byte (which is the first byte of the next instruction). The next instruction is then fetched in the following cycle.

## Program Counter and Data Pointer

The 8051 contains two 16-bit registers: the program counter (PC) and the data pointer (DPTR). Each is used to hold the address of a byte in memory.

Program instruction bytes are fetched from locations in memory that are addressed by the PC. Program ROM may be on the chip at addresses 0000h to 0FFFh, external to the chip for addresses that exceed 0FFFh, or totally external for all addresses from 0000h to FFFFh. The PC is automatically incremented after every instruction byte is fetched and may also be altered by certain instructions. The PC is the only register that does not have an internal address.

The DPTR register is made up of two 8-bit registers, named DPH and DPL, which are used to furnish memory addresses for internal and external code access and external data access. The DPTR is under the control of program instructions and can be specified by its 16-bit name, DPTR, or by each individual byte name, DPH and DPL. DPTR does not have a single internal address; DPH and DPL are each assigned an address.

## A and B CPU Registers

The 8051 contains 34 general-purpose, or working, registers. Two of these, registers A and B, hold results of many instructions, particularly math and logical

operations, of the 8051 central processing unit (CPU). The other 32 are arranged as part of internal RAM in four banks, B0–B3, of eight registers and comprise the mathematical core.

The A (accumulator) register is the most versatile of the two CPU registers and is used for many operations, including addition, subtraction, integer multiplication and division, and Boolean bit manipulations. The A register is also used for all data transfers between the 8051 and any external memory. The B register is used with the A register for multiplication and division operations and has no other function other than as a location where data may be stored.

## Flags and the Program Status Word (PSW)

Flags are 1-bit registers provided to store the results of certain program instructions. Other instructions can test the condition of the flags and make decisions based on the flag states. In order that the flags may be conveniently addressed, they are grouped inside the program status word (PSW) and the power control (PCON) registers.

The 8051 has four math flags that respond automatically to the outcomes of math operations and three general-purpose user flags that can be set to 1 or cleared to 0 by the programmer as desired. The math flags include Carry (C), Auxiliary Carry (AC), Overflow (OV), and Parity (P). User flags are named F0, GF0, and GF1; they are general-purpose flags that may be used by the programmer to record some event in the program. Note that all of the flags can be set and cleared by the programmer at will. The math flags, however, are also affected by math operations.

The program status word is shown in Figure 3.4. The PSW contains the math flags, user program flag F0, and the register select bits that identify which of the four general-purpose register banks is currently in use by the program. The remaining two user flags, GF0 and GF1, are stored in PCON, which is shown in Figure 3.13.

Detailed descriptions of the math flag operations will be discussed in chapters that cover the opcodes that affect the flags. The user flags can be set or cleared using data move instructions covered in Chapter 5.

## Internal Memory

A functioning computer must have memory for program code bytes, commonly in ROM, and RAM memory for variable data that can be altered as the program runs. The 8051 has internal RAM and ROM memory for these functions. Additional memory can be added externally using suitable circuits.

Unlike microcontrollers with Von Neumann architectures, which can use a *single* memory address for either program code or data, *but not for both*, the 8051 has a Harvard architecture, which uses the *same address*, in *different* memo-

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| CY | AC | F0 | RS1 | RS0 | OV | — | P |

**The Program Status Word (PSW) Special Function Register**

| Bit | Symbol | Function |
|---|---|---|
| 7 | CY | Carry flag; used in arithmetic, jump, rotate, and Boolean instructions |
| 6 | AC | Auxiliary Carry flag; used for BCD arithmetic |
| 5 | F0 | User flag 0 |
| 4 | RS1 | Register bank select bit 1 |
| 3 | RS0 | Register bank select bit 0 |
| | | RS1 RS0 |
| | | 0 0 Select register bank 0 |
| | | 0 1 Select register bank 1 |
| | | 1 0 Select register bank 2 |
| | | 1 1 Select register bank 3 |
| 2 | OV | Overflow flag; used in arithmetic instructions |
| 1 | — | Reserved for future use |
| 0 | P | Parity flag; shows parity of register A: 1 = Odd Parity |
| | | Bit addressable as PSW.0 to PSW.7 |



**FIGURE 3.4** ◆ PSW Program Status Word Register

ries, for code and data. Internal circuitry accesses the correct memory based on the nature of the operation in progress.

## Internal RAM

The 128-byte internal RAM, which is shown generally in Figure 3.1 and in detail in Figure 3.5, is organized into three distinct areas:

1. Thirty-two bytes from address 00h to 1Fh that make up 32 working registers organized as four banks of eight registers each. The four register banks are numbered 0 to 3 and are made up of eight registers named R0 to R7. Each register can be addressed by name (when its bank is selected) or by its RAM address. Thus R0 of bank 3 is R0 (if bank 3 is currently selected) or address 18h (whether bank 3 is selected or not). Bits RS0 and RS1 in the PSW determine which bank of registers is currently in use at any time when the program is running. Register banks not selected can be used as general-purpose RAM. Bank 0 is selected on reset.
2. A *bit*-addressable area of 16 bytes occupies RAM *byte* addresses 20h to 2Fh, forming a total of 128 addressable bits. An addressable bit may be

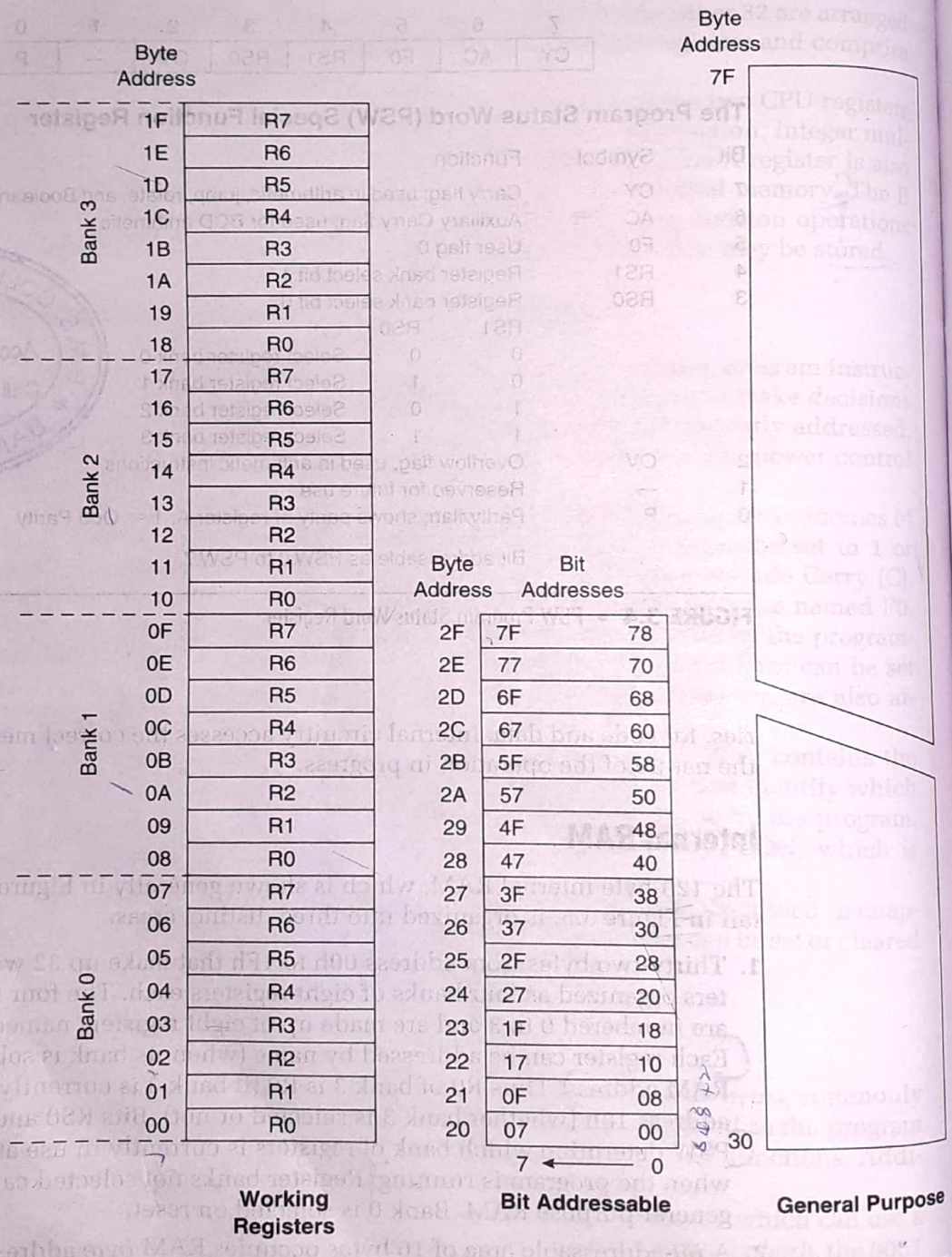



**FIGURE 3.5** ◆ Internal RAM Organization

specified by its *bit* address of 00h to 7Fh, or 8 bits may form any *byte* address from 20h to 2Fh. Thus, for example, bit address 4Fh is also bit 7 of byte address 29h. Addressable bits are useful when the program need only remember a binary event (switch on, light off, etc.). Internal RAM is in short supply as it is, so why use a byte when a bit will do?

3. A general-purpose RAM area above the bit area, from 30h to 7Fh, addressable as bytes.

## The Stack and the Stack Pointer

The stack refers to an area of internal RAM that is used in conjunction with certain opcodes to store and retrieve data quickly. The 8-bit Stack Pointer (SP) register is used by the 8051 to hold an internal RAM address that is called the *top of the stack*. The address held in the SP register is the location in internal RAM where the last byte of data was stored by a stack operation.

When data is to be placed on the stack, the SP increments *before* storing data on the stack so that the stack grows *up* as data is stored. As data is retrieved from the stack, the byte is read from the stack, and then the SP decrements to point to the next available byte of stored data.

Operation of the stack and the SP is shown in Figure 3.6. The SP is set to 07h when the 8051 is reset and can be changed to any internal RAM address by the programmer, using a data move command from Chapter 5.

The stack is limited in height to the size of the internal RAM. The stack has the potential (if the programmer is not careful to limit its growth) to overwrite valuable data in the register banks, bit-addressable RAM, and scratch-pad RAM areas. The programmer is responsible for making sure the stack does not grow beyond predefined bounds!

The stack is normally placed high in internal RAM, by an appropriate choice of the number placed in the SP register, to avoid conflict with the register, bit, and scratch-pad internal RAM areas.

## Special Function Registers

The 8051 operations that do not use the internal 128-byte RAM addresses from 00h to 7Fh are done by a group of specific internal registers, each called a Special-Function register (SFR), which may be addressed much like internal RAM, using addresses from 80h to FFh.

Some SFRs (marked with an asterisk* in Figure 3.1b) are also bit addressable, as is the case for the bit area of RAM. This feature allows the programmer to change only what needs to be altered, leaving the remaining bits in that SFR unchanged.

Not all of the addresses from 80h to FFh are used for SFRs, and attempting to use an address that is not defined, or *empty*, results in unpredictable results.

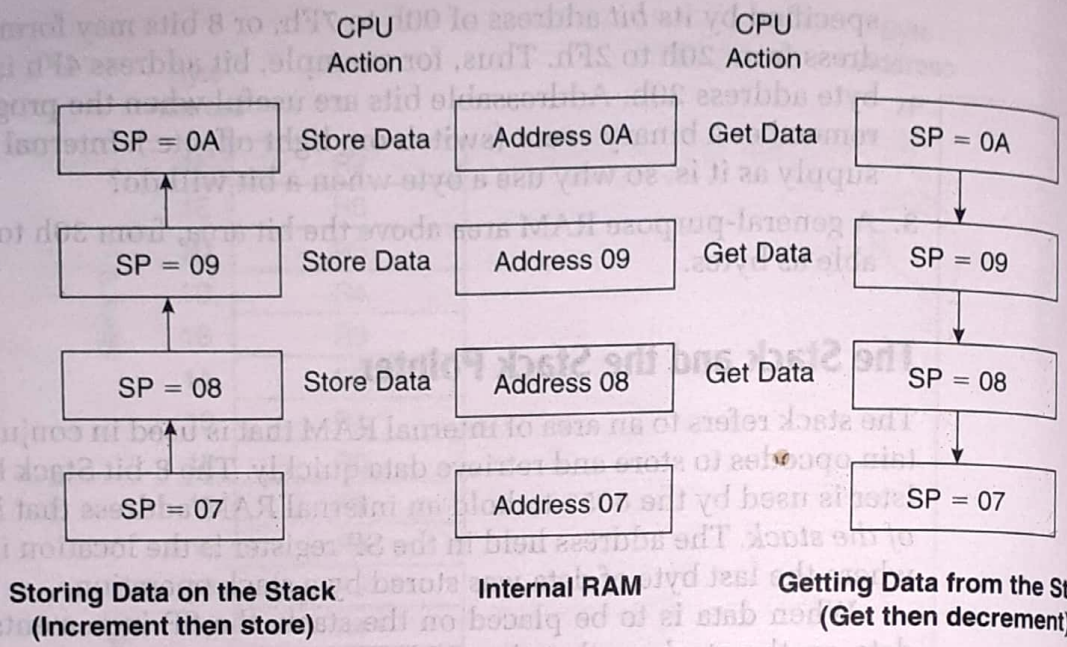



**FIGURE 3.6** ◆ Stack Operation

In Figure 3.1b, the SFR addresses are shown in the upper right corner of each block. The SFR names and equivalent internal RAM addresses are given in Table 3.1. Note that the PC is not part of the SFR and has no internal RAM address. See also Appendix F.

SFRs are named in certain opcodes by their functional names, such as A or TH0, and are referenced by other opcodes by their addresses, such as 0E0h or 8Ch. Note that *any* address used in the program *must* start with a number; thus address E0h for the A SFR begins with 0. Failure to use this number convention will result in an assembler error when the program is assembled.

## Internal ROM

The 8051 is organized so that data memory and program code memory can be in two entirely different physical memory entities. *Each* has the same address ranges.

The structure of the internal RAM has been discussed previously. A corresponding block of internal program code, contained in an internal ROM, occupies code address space 0000h to 0FFFh. The PC is ordinarily used to address program code bytes from addresses 0000h to FFFFh. Program addresses higher than 0FFFh, which exceed the internal ROM capacity, will cause the 8051 to automatically fetch code bytes from external program memory. Code bytes can

**TABLE 3.1**

| Name | Function | Internal RAM Address (HEX) |
|---|---|---|
| A | Accumulator | 0E0 |
| B | Arithmetic | 0F0 |
| DPH | Addressing external memory | 83 |
| DPL | Addressing external memory | 82 |
| IE | Interrupt enable control | 0A8 |
| IP | Interrupt priority | 0B8 |
| P0 | Input/output port latch | 80 |
| P1 | Input/output port latch | 90 |
| P2 | Input/output port latch | A0 |
| P3 | Input/output port latch | 0B0 |
| PCON | Power control | 87 |
| PSW | Program status word | 0D0 |
| SCON | Serial port control | 98 |
| SBUF | Serial port data buffer | 99 |
| SP | Stack pointer | 81 |
| TMOD | Timer/counter mode control | 89 |
| TCON | Timer/counter control | 88 |
| TL0 | Timer 0 low byte | 8A |
| TH0 | Timer 0 high byte | 8C |
| TL1 | Timer 1 low byte | 8B |
| TH1 | Timer 1 high byte | 8D |

also be fetched exclusively from an external memory, addresses 0000h to FFFFh, by connecting the external access pin (EA pin 31 on the DIP) to ground. The PC does not care where the code is; the circuit designer decides whether the code is found totally in internal ROM, totally in external ROM, or in a combination of internal and external ROM.

## 3.2 Input /Output Pins, Ports, and Circuits

One major feature of a microcontroller is the versatility built into the input/output (I/O) circuits that connect the 8051 to the outside world. As noted in Chapter 1, microprocessor designs must add additional chips to interface with external circuitry; this ability is built into the microcontroller.

To be commercially viable, the 8051 had to incorporate as many functions as were technically and economically feasible. The main constraint that limits numerous functions is the number of pins available to the 8051 circuit design-

**TABLE 3.2**

| Parameter | $V_{oh}$ | $I_{oh}$ | $V_{ol}$ | $I_{ol}$ | $V_{il}$ | $I_{il}$ | $V_{ih}$ | $I_{ih}$ | $P_t$ |
|---|---|---|---|---|---|---|---|---|---|
| CMOS | 2.4 V | −60 $\mu$A | .45 V | 1.6 mA | .9 V | \|10 $\mu$A\| | 1.9 V | \|10 $\mu$A\| | 50 mW |
| NMOS | 2.4 V | −80 $\mu$A | .45 V | 1.6 mA | .8 V | −800 $\mu$A | 2.0 V | 10 $\mu$A | 800 mW |

## Port 0

Port 0 pins may serve as inputs, outputs, or, when used together, as a bidirectional low-order address and data bus for external memory. For example, when a pin is to be used as an input, a 1 *must be* written to the corresponding port 0 latch by the program, thus turning both of the output transistors off, which in turn causes the pin to "float" in a high-impedance state, and the pin is essentially connected to the input buffer.

When used as an output, the pin latches that are programmed to a 0 will turn on the lower FET, grounding the pin. All latches that are programmed to a 1 still float; thus, external pullup resistors will be needed to supply a logic high when using port 0 as an output.

When port 0 is used as an address bus to external memory, internal control signals switch the address lines to the gates of the Field Effect Transistors (FETs). A logic 1 on an address bit will turn the upper FET on and the lower FET off to provide a logic high at the pin. When the address bit is a zero, the lower FET is on and the upper FET off to provide a logic low at the pin. After the address has been formed and latched into external circuits by the Address Latch Enable (ALE) pulse, the bus is turned around to become a data bus. Port 0 now reads data from the external memory and must be configured as an input, so a logic 1 is automatically written by internal control logic to all port 0 latches.

## Port 1

Port 1 pins have no dual functions. Therefore, the output latch is connected directly to the gate of the lower FET, which has an FET circuit labeled Internal FET Pullup as an active pullup load.

Used as an input, a 1 is written to the latch, turning the lower FET off; the pin and the input to the pin buffer are pulled high by the FET load. An external circuit can overcome the high-impedance pullup and drive the pin low to input a 0 or leave the input high for a 1.

If used as an output, the latches containing a 1 can drive the input of an external circuit high through the pullup. If a 0 is written to the latch, the lower

FET is on, the pullup is off, and the pin can drive the input of the external circuit low.

To aid in speeding up switching times when the pin is used as an output, the internal FET pullup has another FET in parallel with it. The second FET is turned on for two oscillator time periods during a low-to-high transition on the pin, as shown in Figure 3.7. This arrangement provides a low impedance path to the positive voltage supply to help reduce rise times in charging any parasitic capacitances in the external circuitry.

## Port 2

Port 2 may be used as an input/output port similar in operation to port 1. The alternate use of port 2 is to supply a high-order address byte in conjunction with the port 0 low-order byte to address external memory.

Port 2 pins are momentarily changed by the address control signals when supplying the high byte of a 16-bit address. Port 2 latches remain stable when external memory is addressed, as they do not have to be turned around (set to 1) for data input as is the case for port 0.

## Port 3

Port 3 is an input/output port similar to port 1. The input and output functions can be programmed under the control of the P3 latches or under the control of various other special function registers. The port 3 alternate uses are shown in Table 3.3.

Unlike ports 0 and 2, which can have external addressing functions and change all eight port bits when in alternate use, each pin of port 3 may be individually programmed to be used either as I/O or as one of the alternate functions.

**TABLE 3.3**

| Pin | Alternate Use | SFR |
|---|---|---|
| P3.0 – RXD | Serial data input | SBUF |
| P3.1 – TXD | Serial data output | SBUF |
| P3.2 – INT0 | External interrupt 0 | TCON.1 |
| P3.3 – INT1 | External interrupt 1 | TCON.3 |
| P3.4 – T0 | External timer 0 input | TMOD |
| P3.5 – T1 | External timer 1 input | TMOD |
| P3.6 – WR | External memory write pulse | — |
| P3.7 – RD | External memory read pulse | — |

## 3.3 External Memory

The system designer is not limited by the amount of internal RAM and ROM available on chip. Two separate external memory spaces are made available by the 16-bit PC and DPTR and by different control pins for enabling external ROM and RAM chips. Internal control circuitry accesses the correct physical memory, depending on the machine cycle state and the opcode being executed.

There are several reasons for adding external memory, particularly program memory, when applying the 8051 in a system. When the project is in the prototype stage, the expense—in time and money—of having a masked internal ROM made for each program "try" is prohibitive. To alleviate this problem, the manufacturers make available an EPROM version, the 8751, which has 4K of on-chip EPROM that may be programmed and erased as needed as the program is developed. The resulting circuit board layout will be identical to one that uses a factory-programmed 8051. The only drawbacks to the 8751 are the specialized EPROM programmers that must be used to program the nonstandard 40-pin part, and the limit of "only" 4096 bytes of program code.

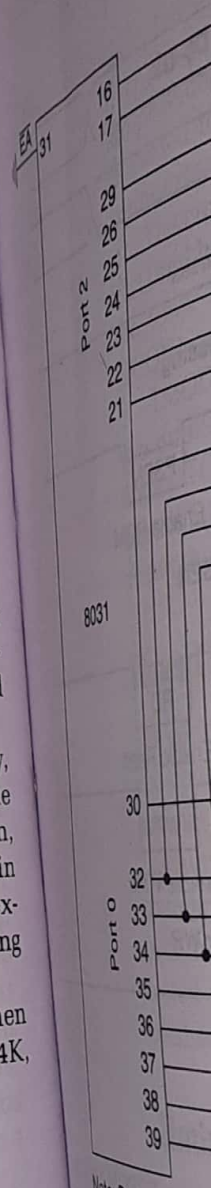
The 8751 solution works well if the program will fit into 4K. Unfortunately, many times, particularly if the program is written in a high-level language, the program size exceeds 4K, and an external program memory is needed. Again, the manufacturers provide a version for the job, the ROMless 8031. The $\overline{EA}$ pin is grounded when using the 8031, and all program code is contained in an external EPROM that may be as large as 64K and that can be programmed using standard EPROM programmers.

External RAM, which is accessed by the DPTR, may also be needed when 128 bytes of internal data storage is not sufficient. External RAM, up to 64K, may also be added to any chip in the 8051 family.

### Connecting External Memory

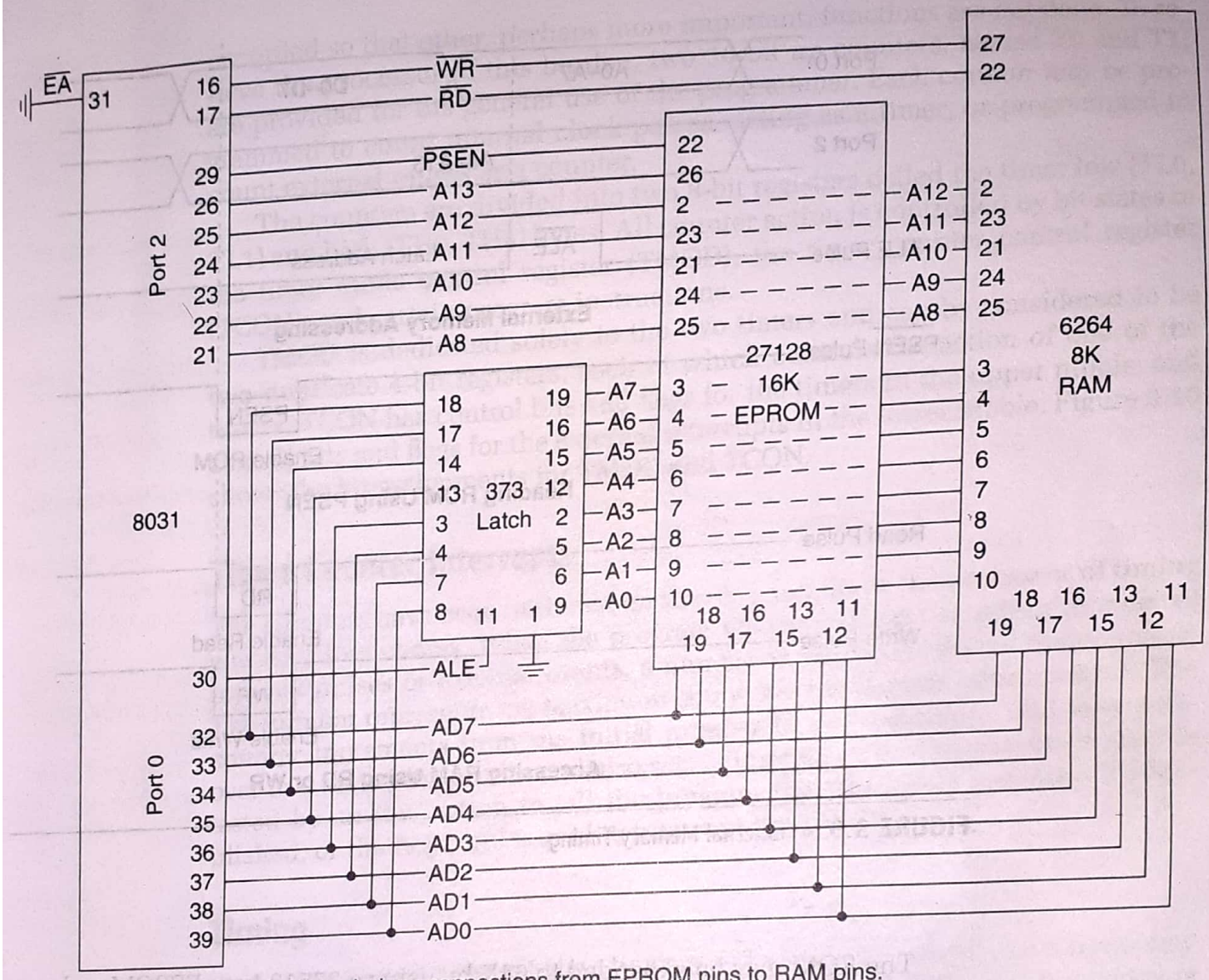
Figure 3.8 shows the connections between an 8031 and an external memory configuration consisting of 16K of EPROM and 8K of static RAM. The 8051 accesses external RAM whenever certain program instructions are executed. External ROM is accessed whenever the $\overline{EA}$ (external access) pin is connected to ground or when the PC contains an address higher than the last address in the internal 4K ROM (0FFFh). 8051 designs can thus use internal and external ROM automatically; the 8031, having no internal ROM, must have $\overline{EA}$ grounded.

Figure 3.9 shows the timing associated with an external memory access cycle. During any memory access cycle, port 0 is time multiplexed. That is, it first provides the lower byte of the 16-bit memory address, then acts as a bidirectional data bus to write or read a byte of memory data. Port 2 provides the high byte of the memory address during the entire memory read/write cycle.




Note: Dashed lines
Note: Vcc and Gnd

FIGURE 3.8 ◆ External




Note: Dashed lines ( – – – ) show connections from EPROM pins to RAM pins.
Note: Vcc and Gnd pin connections are not shown.

**FIGURE 3.8** ◆ External Memory Connections

The lower address byte from port 0 must be latched into an external register to save the byte. Address byte save is accomplished by the ALE clock pulse that provides the correct timing for the '373 type data latch. The port 0 pins then become free to serve as a data bus.

If the memory access is for a byte of program code in the ROM, the $\overline{PSEN}$ (program store enable) pin will go low to enable the ROM to place a byte of program code on the data bus. If the access is for a RAM byte, the $\overline{WR}$ (write) or $\overline{RD}$ (read) pins will go low, enabling data to flow between the RAM and the data bus.

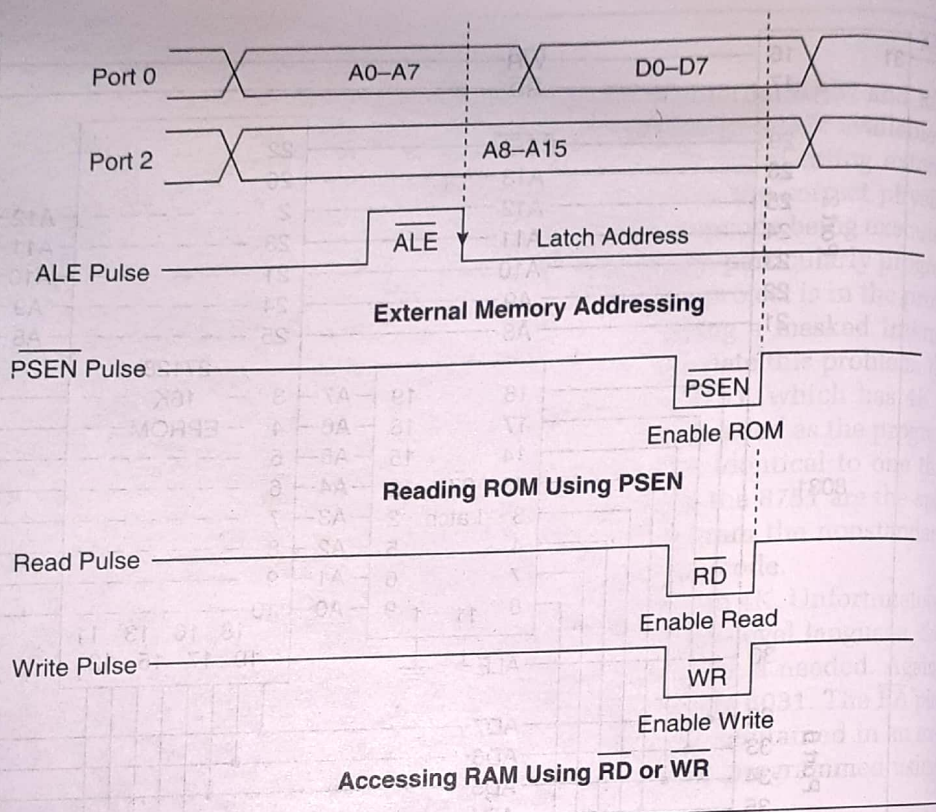



**FIGURE 3.9** ◆ External Memory Timing

The ROM may be expanded to 64K by using a 27512 type EPROM and connecting the remaining port 2 upper address lines A14 – A15 to the chip.

SRAM capacity may be expanded to 64K by using a 62864-type chip.

Note that the $\overline{WR}$ and $\overline{RD}$ signals are alternate uses for port 3 pins 16 and 17. Also, port 0 is used for the lower address byte and data; port 2 is used for upper address bits. The use of external memory consumes many of the port pins, leaving only port 1 and parts of port 3 for general I/O.

## 3.4 Counters and Timers

Many microcontroller applications require the counting of external events, such as the frequency of a pulse train, or the generation of precise internal time delays between computer actions. Both of these tasks can be accomplished using software techniques, but software loops for counting or timing keep the processor

occupied so that other, perhaps more important, functions are not done. To relieve the processor of this burden, two 16-bit *up* counters, named T0 and T1, are provided for the general use of the programmer. Each counter may be programmed to count internal clock pulses, acting as a timer, or programmed to count external pulses as a counter.

The counters are divided into two 8-bit registers called the timer low (TL0, TL1) and high (TH0, TH1) bytes. All counter action is controlled by bit states in the timer mode control register (TMOD), the timer/counter control register (TCON), and certain program instructions.

TMOD is dedicated solely to the two timers and can be considered to be two duplicate 4-bit registers, each of which controls the action of one of the timers. TCON has control bits and flags for the timers in the upper nibble, and control bits and flags for the external interrupts in the lower nibble. Figure 3.10 shows the bit assignments for TMOD and TCON.

## Timer Counter Interrupts

The counters have been included on the chip to relieve the processor of timing and counting chores. When the program wishes to count a certain number of internal pulses or external events, a number is placed in one of the counters. The number represents the maximum count *less* the desired count, plus 1. The counter increments from the initial number to the maximum and then rolls over to 0 on the final pulse and also sets a timer flag. The flag condition may be tested by an instruction to tell the program that the count has been accomplished, or the flag may be used to interrupt the program.

## Timing

If a counter is programmed to be a timer, it will count the internal clock frequency of the 8051 oscillator divided by 12d. As an example, if the crystal frequency is 6.0 megahertz, then the timer clock will have a frequency of 500 kilohertz.

The resultant timer clock is gated to the timer by means of the circuit shown in Figure 3.11. In order for oscillator clock pulses to reach the timer, the C/$\overline{T}$ bit in the TMOD register must be set to 0 (timer operation). Bit TRX in the TCON register must be set to 1 (timer run), and the *gate* bit in the TMOD register must be 0, or external pin $\overline{INTX}$ must be a 1. In other words, the counter is configured as a timer, then the timer pulses are gated to the counter by the run bit *and* the gate bit *or* the external input bits $\overline{INTX}$.

## Timer Modes of Operation

The timers may operate in any one of four modes that are determined by the mode bits, M1 and M0, in the TMOD register. Figure 3.12 shows the four timer modes.

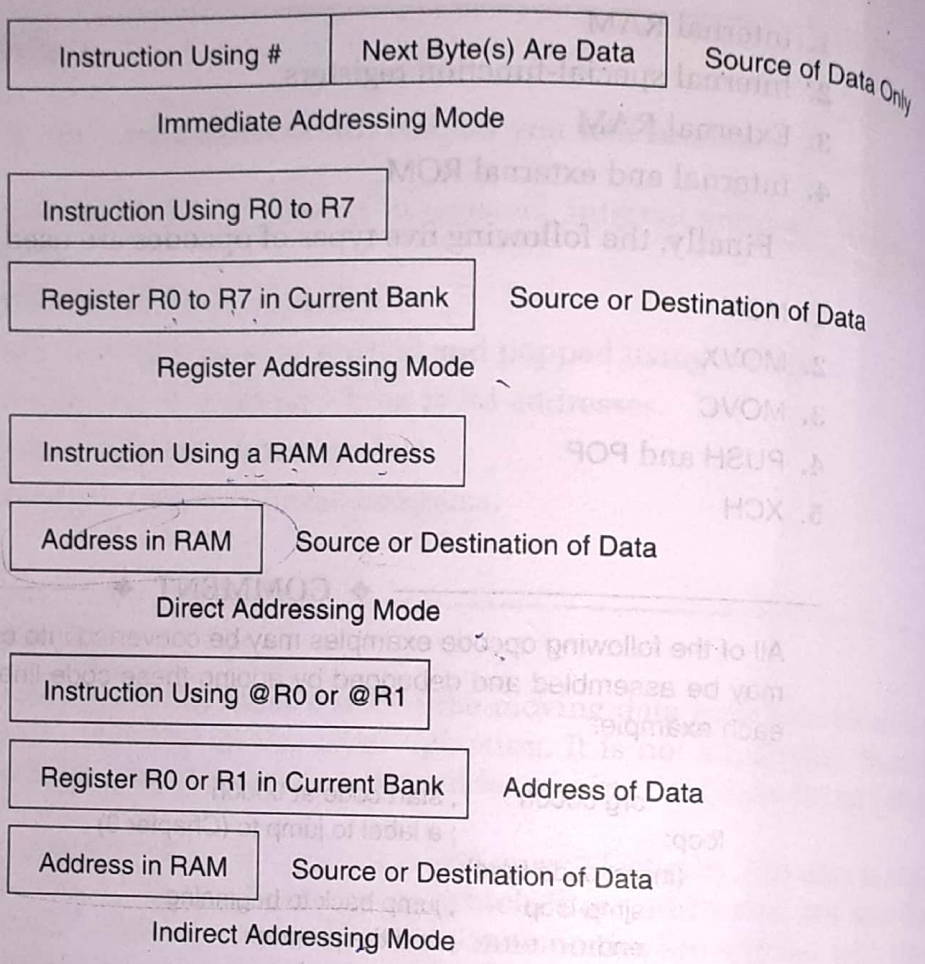



**FIGURE 5.1** ◆ Addressing Modes

The mnemonic for immediate data is the pound sign (#). Occasionally, in the rush to meet a deadline, we might forget to use the # for immediate data. The resulting opcode is often a legal command that is assembled with no objections by the assembler. This omission guarantees that the rush will continue.

Three mnemonics can copy immediate numbers from the opcode into registers R0–R7 (of the currently selected register bank), A, and DPTR:

| Mnemonic | Operation |
|---|---|
| MOV Rr,#n | Copy the 8-bit number n into register Rr (of the current register bank) |
| MOV A,#n | Copy the 8-bit number n into the Accumulator register |
| MOV DPTR,#nn | Copy the 16-bit number nn into the DPTR register |

Immediate addressing modes, using some of the registers from R0 to R7 in the currently selected register bank, register A, and register DPTR, are shown in the following list:

| Mnemonic | Operation |
|---|---|
| MOV R0,#00h | Put the immediate 8-bit number 00h in register R0 |
| MOV R1,#01h | Put the immediate 8-bit number 01h in register R1 |
| MOV R4,#04h | Put the immediate 8-bit number 04h in register R4 |
| MOV R7,#07h | Put the immediate 8-bit number 07 in register R7 |
| MOV A,#0AAh | Put the immediate 8-bit number AAh in register A |
| MOV DPTR,#1234h | Put the immediate 16-bit number 1234h in register DPTR |

## Register Addressing Mode

Certain register names may be used as part of the opcode mnemonic as sources or destinations of data. Registers A, DPTR, and R0 to R7 may be named as part of the opcode mnemonic. Other registers in the 8051 may be addressed using the direct addressing mode. Some assemblers can equate many of the direct addresses to the register name (as is the case with the assembler discussed in this book) so that register names may be used in lieu of register addresses. Remember that the registers used in the opcode as R0 to R7 are the ones that are *currently* chosen by the bank-select bits, RS0 and RS1 in the PSW.

Register-to-register moves are as follows:

| Mnemonic | Operation |
|---|---|
| MOV A,Rr | Copy data from register Rr to register A |
| MOV Rr,A | Copy data from register A to register Rr |

A data MOV does not alter the contents of the data source address. A *copy* of the data is made from the source and moved to the destination address. The contents of the destination address are replaced by the source address contents. The following list shows examples of MOV opcodes with immediate and register addressing modes:

| Mnemonic | Operation |
|---|---|
| MOV A,#0F1h | Move the immediate data byte F1h to the A register |
| MOV A,R0 | Copy the data in register R0 to register A |
| MOV DPTR,#0ABCDh | Move the immediate data bytes ABCDh to the DPTR |
| MOV R5,A | Copy the data in register A to register R5 |
| MOV R3,#1Ch | Move the immediate data byte 1Ch to register R3 |

◆ CAUTION ◆

- It is impossible to have immediate data as a destination.
- All numbers *must* start with a decimal number (0 – 9), or the assembler assumes the number is a *label*.
- Register-to-register moves using the register addressing mode occur between registers A and R0 to R7.

## Direct Addressing Mode

All 128 bytes of internal RAM and the SFRs may be addressed directly using the single-byte address assigned to each RAM location and each special-function register. See Appendix F for an overall SFR/memory map.

Internal RAM uses addresses from 00h to 7Fh to address each byte. The SFR addresses exist from 80h to FFh at the locations shown in Table 5.1.

◆ CAUTION ◆

- Note that there are "gaps" in the addresses of the SFRs; the addresses are not in order.
- Note the use of a leading 0 for all numbers that begin with an alphabetic (alpha) character.

RAM addresses 00 to 1Fh are *also* the locations assigned to the four banks of eight working registers, R0 to R7. This assignment means that R2 of register

**TABLE 5.1**

| SFR | Address (hex) |
|---|---|
| A | 0E0 |
| B | 0F0 |
| DPL | 82 |
| DPH | 83 |
| IE | 0A8 |
| IP | 0B8 |
| P0 | 80 |
| P1 | 90 |
| P2 | 0A0 |
| P3 | 0B0 |
| PCON | 87 |
| PSW | 0D0 |
| SBUF | 99 |
| SCON | 98 |
| SP | 81 |
| TCON | 88 |
| TMOD | 89 |
| TH0 | 8C |
| TL0 | 8A |
| TH1 | 8D |
| TL1 | 8B |

**TABLE 5.2**

| Bank | Register | Address (hex) | Bank | Register | Address (hex) |
|---|---|---|---|---|---|
| 0 | R0 | 00 | 2 | R0 | 10 |
| 0 | R1 | 01 | 2 | R1 | 11 |
| 0 | R2 | 02 | 2 | R2 | 12 |
| 0 | R3 | 03 | 2 | R3 | 13 |
| 0 | R4 | 04 | 2 | R4 | 14 |
| 0 | R5 | 05 | 2 | R5 | 15 |
| 0 | R6 | 06 | 2 | R6 | 16 |
| 0 | R7 | 07 | 2 | R7 | 17 |
| 1 | R0 | 08 | 3 | R0 | 18 |
| 1 | R1 | 09 | 3 | R1 | 19 |
| 1 | R2 | 0A | 3 | R2 | 1A |
| 1 | R3 | 0B | 3 | R3 | 1B |
| 1 | R4 | 0C | 3 | R4 | 1C |
| 1 | R5 | 0D | 3 | R5 | 1D |
| 1 | R6 | 0E | 3 | R6 | 1E |
| 1 | R7 | 0F | 3 | R7 | 1F |

bank 0 can be addressed in the register mode as R2 or in the direct mode as 02h. The direct addresses of the working registers are shown in Table 5.2.

Only one bank of working registers is *active* at any given time. The PSW special-function register holds the bank-select bits, RS0 and RS1, which determine which register bank is in use.

When the 8051 is reset, RS0 and RS1 are set to 00b to select the working registers in bank 0, located from 00h to 07h in internal RAM. Reset also sets SP to 07h, and the stack will grow *up* as it is used. This growing stack will overwrite the register banks above bank 0. Be *sure* to set the SP to a number above those of any working registers the program may use.

The programmer may choose any other bank by setting RS0 and RS1 as desired; this bank change is often done to "save" one bank and choose another when servicing an interrupt or using a subroutine.

The programmer may elect to use the *absolute* numeric address number for an SFR or may use a *symbol* (name) for the SFR. For example, the following instructions both move a *constant* number into port 1:

```
mov 90h,#0a5h
mov p1,#0a5h
```

The A51 assembler, supplied with this book, "looks up" the actual address of an SFR when the programmer uses an SFR symbol. Please refer to the end of Appendix B for a list of SFR symbols.

We shall use both methods of specifying SFRs in this book to emphasize the fact that SFRs are internal RAM addresses.

The moves made possible using direct, immediate, and register addressing modes are as follows:

| Mnemonic | Operation |
|---|---|
| MOV A,add | Copy data from direct address add to register A |
| MOV add,A | Copy data from register A to direct address add |
| MOV Rr,add | Copy data from direct address add to register Rr |
| MOV add,Rr | Copy data from register Rr to direct address add |
| MOV add,#n | Copy immediate data byte n to direct address add |
| MOV add1,add2 | Copy data from direct address add2 to direct address add1 |

The following list shows examples of MOV opcodes using direct, immediate, and register addressing modes:

| Mnemonic | Operation |
|---|---|
| MOV A,80h | Copy data from the port 0 pins to register A |
| MOV 80h,A | Copy data from register A to the port 0 latch |
| MOV 3Ah,#3Ah | Copy immediate data byte 3Ah to RAM location 3Ah |
| MOV R0,12h | Copy data from RAM location 12h to register R0 |
| MOV 8Ch,R7 | Copy data from register R7 to timer 0 high byte |
| MOV 5Ch,A | Copy data from register A to RAM location 5Ch |
| MOV 0A8h,77h | Copy data from RAM location 77h to IE register |

◆ CAUTION ◆

- MOV instructions that refer to direct addresses above 7Fh that are not SFRs will result in errors. The SFRs are physically on the chip; all other addresses above 7Fh do not physically exist.
- Moving data to a port changes the port *latch;* moving data from a port gets data from the port *pins.*
- Moving data from a direct address to itself is not predictable and could lead to errors.

## Indirect Addressing Mode

For all the addressing modes covered to this point, the source or destination of the data is an absolute number or a name. Inspection of the opcode reveals exactly what the addresses are of the destination and source. For example, the opcode MOV A,R7 says that the A register will get a copy of whatever data is in register R7; MOV 33h,#32h moves the hex number 32 to hex RAM address 33.

The indirect addressing mode uses a register to *hold* the actual address that will finally be used in the data move; the register itself is *not* the address, but rather the number *in* the register. Indirect addressing for MOV opcodes uses register R0 or R1, often called a *data pointer*, to hold the address of one of the data locations in RAM from address 00h to 7Fh. The number that is in the pointing register (Rp) cannot be known unless the history of the register is known. The mnemonic symbol used for indirect addressing is the "at" sign, which is printed as @.

The moves made possible using immediate, direct, register, and indirect addressing modes are as follows:

| Mnemonic | Operation |
|---|---|
| MOV @Rp,#n | Copy the immediate byte n to the address in Rp |
| MOV @Rp,add | Copy the contents of add to the address in Rp |
| MOV @Rp,A | Copy the data in A to the address in Rp |
| MOV add,@Rp | Copy the contents of the address in Rp to add |
| MOV A,@Rp | Copy the contents of the address in Rp to A |

The following list shows examples of MOV opcodes, using immediate, register, direct, and indirect modes

| Mnemonic | Operation |
|---|---|
| MOV A,@R0 | Copy the contents of the address in R0 to the A register |
| MOV @R1,#35h | Copy the number 35h to the address in R1 |
| MOV add,@R0 | Copy the contents of the address in R0 to add |
| MOV @R1,A | Copy the contents of A to the address in R1 |
| MOV @R0,80h | Copy the contents of the port 0 pins to the address in R0 |

◆ CAUTION ◆

- The number in register Rp must be a RAM address.
- *Only* registers R0 or R1 may be used for indirect addressing.

## 5.2 External Data Moves

As discussed in Chapter 3, it is possible to expand RAM and ROM memory space by adding external memory chips to the 8051 microcontroller. The external memory can be as large as 64K for each of the RAM and ROM memory areas. Opcodes that access this external memory *always* use indirect addressing to specify the external memory.

Figure 5.2 shows that registers R0, R1, and the aptly named DPTR can be used to hold the address of the data byte in external RAM. R0 and R1 are lim-

A program that does not have to deal unexpectedly with the world outside of the microcontroller could be written using jumps to alter program flow as external conditions require. This sort of program can determine external conditions by moving data from the port pins to a location and jumping on the conditions of the port pin data. This technique is called *polling* and requires that the program does not have to respond to external conditions quickly. (Quickly means in microseconds; slowly means in milliseconds.)

Another method of changing program execution is using *interrupt* signals on certain external pins or internal registers to automatically cause a branch to a smaller program that deals with the specific situation. When the event that caused the interruption has been dealt with, the program resumes at the point in the program where the interruption took place. Interrupt action can also be generated using software instructions named *calls.*

Call instructions may be included explicitly in the program as mnemonics or implicitly included using hardware interrupts. In both cases, the call is used to execute a smaller, stand-alone program, which is termed a *routine* or, more often, a *subroutine.*

## Subroutines

A *subroutine* is a program that may be used many times in the execution of a larger program. The subroutine could be written into the body of the main program everywhere it is needed, resulting in the fastest possible code execution. Using a subroutine in this manner has several serious drawbacks.

Common practice when writing a large program is to divide the total task among many programmers in order to speed completion. The entire program can be broken into smaller parts and each programmer given a part to write and debug. The main program can then call each of the parts, or subroutines, that have been developed and tested by each individual of the team.

Even if the program is written by one individual, it is more efficient to write an often-used routine once and then call it many times as needed. Also, when writing a program, the programmer does the main part first. Calls to subroutines, which will be written later, enable the larger task to be defined before the programmer becomes bogged down in the details of the application.

Finally, it is quite common to buy *libraries* of common subroutines that can be called by a main program. Again, buying libraries leads to faster program development.

## Calls and the Stack

A call, whether hardware or software initiated, causes a jump to the address where the called subroutine is located. At the end of the subroutine the program resumes operation at the opcode address immediately *following* the call. As calls can be located anywhere in the program address space and used many

times, there must be an automatic means of storing the address of the instruction following the call so that program execution can continue after the subroutine has executed.

The *stack* area of internal RAM is used to automatically store the address, called the return address, of the instruction found immediately after the call. The Stack Pointer register holds the address of the *last* space used on the stack. It stores the return address above this space, adjusting itself upward as the return address is stored. The terms *stack* and *stack pointer* are often used interchangeably to designate the *top* of the stack area in RAM that is pointed to by the stack pointer.

Figure 8.2 diagrams the following sequence of events:

1. A call opcode occurs in the program software, or an interrupt is generated in the hardware circuitry.
2. The return address of the next instruction after the call instruction or interrupt is found in the program counter.
3. The return address bytes are pushed on the stack, *low* byte *first*.
4. The stack pointer is incremented for each push on the stack.
5. The subroutine address is placed in the program counter.
6. The subroutine is executed.
7. A RET (return) opcode is encountered at the end of the subroutine.
8. Two pop operations restore the return address to the PC from the stack area in internal RAM.
9. The stack pointer is decremented for each address byte pop.

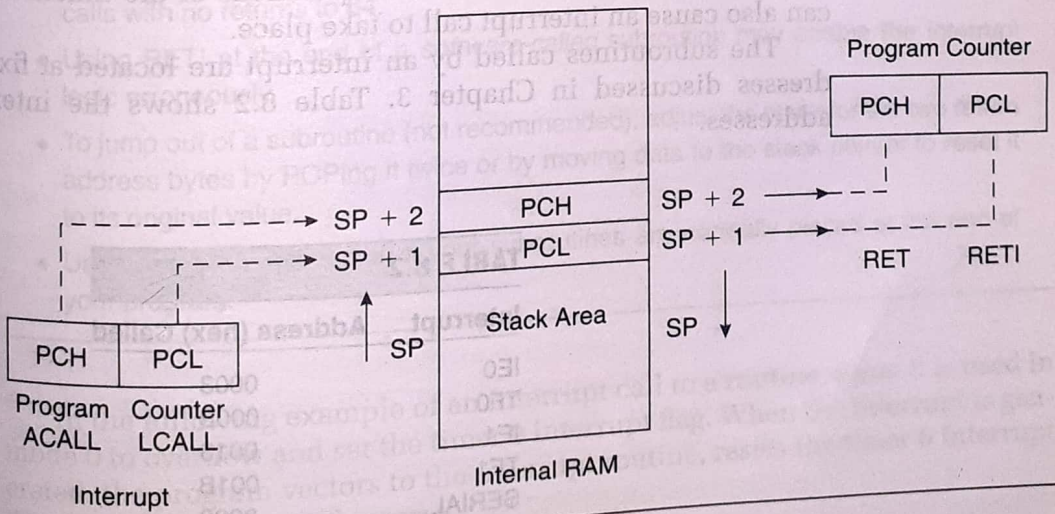



**FIGURE 8.2** ◆ Storing and Retrieving the Return Address

◆ CAUTION ◆

- All of these steps are automatically handled by the 8051 hardware. It is the *responsibility* of the programmer to ensure that the subroutine ends in a RET instruction *and* that the stack does not grow up into data areas that are used by the program.
- Remember to set the SP above your data area.

### Calls and Returns

Calls use short- or long-range addressing; returns have no addressing mode specified but are always long range. The following list shows examples of call opcodes:

| Mnemonic | Operation |
|---|---|
| ACALL sadd | Call the subroutine located on the same page as the address of the opcode immediately following the ACALL instruction; push the address of the instruction immediately after the call on the stack |
| LCALL ladd | Call the subroutine located anywhere in program memory space; push the address of the instruction immediately following the call on the stack |
| RET | Pop 2 bytes from the stack into the program counter |

## 8.4 Interrupts and Returns

As mentioned previously, an *interrupt* is a hardware-generated call. Just as a call opcode can be located within a program to automatically access a subroutine, certain pins on the 8051 can cause a call when external electrical signals on them go to a low state. Internal operations of the timers and the serial port can also cause an interrupt call to take place.

The subroutines called by an interrupt are located at fixed hardware addresses discussed in Chapter 3. Table 8.2 shows the interrupt subroutine addresses.

**TABLE 8.2**

| Interrupt | Address (hex) Called |
|---|---|
| IE0 | 0003 |
| TF0 | 000B |
| IE1 | 0013 |
| TF1 | 001B |
| SERIAL | 0023 |