8051 addressing modes

Contents

1 A Big Distinction: Direct vs Indirect Addressing 1
  1.1 Familiar case: indirect addressing using “@DPTR” 1
  1.2 A less familiar case: indirect addressing using internal RAM 2
  1.3 Downright Obscure Case of Indirect Addressing: “Indexed” Table-read from code memory 3

2 Seemingly Ambiguous Cases Disambiguated by Context 3
  2.1 Bit versus Byte 3
  2.2 Two meanings for an internal RAM address 5

3 One More Oddness: You may be able to do it if you say it nicely 6
  3.1 Specifying an internal address rather than register name 6
  3.2 ... another similar case: PUSH R0 fails, but PUSH 0 succeeds 6

You don’t need to know a lot about the 8051’s addressing; of its modest set, we use most modes, but not all. When you are learning assembly language, it’s good news that the 8051 isn’t very versatile (in contrast, say, to Motorola’s 68000, which we used some years ago; it offered fourteen addressing modes). There’s less to learn than for a more complex machine. On the other hand, when you’re trying to write code to get something done, the 8051’s restrictions are less pleasing. Many addressing modes that make perfect sense—such as MOVX @DPTR, #012h; CLR R5; MOV R3, R4—just aren’t available.

1 A Big Distinction: Direct vs Indirect Addressing

Chances are, you’re already comfortable with this distinction: it’s the difference between—

- putting a value into a register (MOV DPTR, #8000h), on one hand, versus...
- putting a value into a location pointed to by a register (MOVX @DPTR, A)

1.1 Familiar case: indirect addressing using “@DPTR”

Here’s a diagram to remind you of what goes on in the two contrasted cases:

---

1Revisions: add example of PUSH restrictions, because of register-set ambiguity (4/08).
1.2 A less familiar case: indirect addressing using internal RAM

You know that R0 and R1 can be used as pointers to external addresses, where P2 is used to hold the high byte of the 16-bit address. R0 and R1 can also point to internal RAM (where an 8-bit pointer is sufficient, because of the 8051’s modest allotment of RAM). This less familiar case is useful to make the contrast between direct and indirect addressing clear.

We are making the same distinction as in the “@DPTR…” example, above—namely

- putting a value into a register (MOV R0, #034h), on one hand, versus…
- putting a value into a location pointed to by a register (MOV R0, #0ABh)

The first operation puts the value 34h into register R0; the second operation puts the value ABh into the location pointed to by R0, namely, address 34h.

Again, a sketch to illustrate the idea:
1.3 Downright Obscure Case of Indirect Addressing: “Indexed” Table-read from code memory

Instead of moving a pointer, by incrementing or decrementing, sometimes it is convenient to let a register value determine which of several entries in a table is to be read. This is called “indexed” addressing: an index value is added to a base address, so as to form the effective address from which the read is to occur. Some processors offer a rich set of such operations (for example, Motorola’s 8-bit workhorse, the HC11, a competitor of the 8051). The 8051 knows just two ways to do this trick.

The two addressing modes use the instructions “MOVC A, @A+PC” or “MOVC A, @A+DPTR.” These instructions...

- use the A register (accumulator) to hold the “index” value;
- use, as the “base” address, either
  - the DPTR or
  - the “program counter” (just the program address, after execution of this indexed operation).

Here’s an example, using the program counter as base address (this scheme offers the advantage that it does not tie up the DPTR). The following program fragment uses a value taken from the keypad (a 4-bit value\(^2\)) as index into a short table that determines what mask to apply to an ADC value\(^3\).

```assembly
MOVX A, @DPtr ; pick up keypad value
ANL A, #0Fh    ; keep only the low nybble
...          
MOVC A, @A+PC ; ...and use that as index into table of mask values
RET
```

TABLE: DB 80h   ; Table of mask values: 1 from keypad (DB means “define byte”)  
         DB 0C0h  ; 2 from keypad (and so on); 0 from keypad is illegitimate choice  
         DB 0E0h  
         DB 0F0h  
         DB 0F8h  
         DB 0FCh  
         DB 0FEh  
         DB 0FFh  

2 Seemingly Ambiguous Cases Disambiguated by Context

2.1 Bit versus Byte

If we write

```assembly
SOFTFLAG EQU 0 ; a pseudo op addressed to assembler, not to the 8051, by the way
...          
CLR SOFTFLAG
```

\(^2\)The high 4 bits are masked out, so that only the most recent key-press matters.

\(^3\)This fragment is a piece of a demonstration program we used in the first digital class, to try out the effect of varying the number of bits in a digital sample.
... the assembler inserts zero in place of the human-friendly name “SOFTFLAG.” But how does the 8051 know whether to treat this as byte zero of internal RAM (which happens to be the other name for register R0) or bit zero (which happens to be the first bit-addressable location, and lives at byte 20h)?

The answer is boringly simple: since no CLR byte operation is available, the 8051 sees no ambiguity⁴: this is a bit clear, so it clears bit zero.

⁴Isn’t “disambiguated” a horrible word? We apologize to our readers for this use. But it does make the point firmly.
2.2 Two meanings for an internal RAM address

Internally-defined port zero (P0) lives at address 80h. But this is also the first byte of our usual stack—the “scratch-pad” RAM area on the 8052/DS89C420.

Below is an annotated version of the “Weird address space…” image from the micro notes, showing this point, along with the location of R0 and “bit zero.”

![Image of the 8051 address space]

Figure 3: P0 and bottom of Stack share an address!

Again, the resolution of this seeming-ambiguity is simple: if the addressing mode is indirect (as it is in stack use), then it’s the scratch RAM that is accessed; if the addressing mode is direct, then it is P0 that is addressed:

```
MOV P2, #12h ; (DIRECT): write a byte to internally-defined Port Zero (at address 80h)
ACALL DELAY ; (INDIRECT): store return address automatically on stack—which, in our Lab 19
; program, began at address 80h (return address was 064h, as indicated on sketch, above

```

80h in internal RAM lives in two places, but each is accessible only in its peculiar way—directly (P2) or indirectly (scratch RAM, used as stack, in our applications).

---

5It isn’t quite right to call the controller “8051,” here, since it’s only the 8052—along with our Dallas versions imitating it—that do offer RAM at this address. The 8051 offered only half as much on-chip RAM, and thus ran out of RAM exactly at that point, 80h.
3 One More Oddness: You may be able to do it if you say it nicely

3.1 Specifying an internal address rather than register name...

One cannot do the simple operation MOV R1, R2. If you need to do it, you may find yourself writing, first, MOV A, R2; then MOV R1, A. That works, but seems very clumsy. You are permitted to do the following:

```
MOV R1, 2 ; this is permitted though address 2 is R2. Very strange.
```

This is strange, but not quite crazy (and an assembler can’t quite fix it): the two operations, though they ought to give the same results, would be coded differently in machine language; in other words, they are different operations. This point may come clearer if we compare two permitted operations that give the same result but are encoded differently: MOV R1, A is a different instruction from MOV R1, 0E0h, even though the two operations do exactly the same action: copy the contents of the accumulator to register R1.

3.2 ...another similar case: PUSH R0 fails, but PUSH 0 succeeds

PUSH R0 fails for a good reason: the 8051 provides 4 sets of scratch registers, R0 through R7. Nothing in the name of the register identifies which of the 4 “R0’s” one intends; it’s just “the current R0” (which one is determined by two bits in the “program status word,” PSW). The PUSH and POP operations aren’t allowed to make the assumption usually made, that we intend the current register set. So,

```
PUSH R0 ; fails: assembler considers this
PUSH 0 ; succeeds: 0 is the address at which the lowest of the 4 “R0’s” lives: unambiguous
```

This is hard to remember—particularly since R0 often is considered not ambiguous—as in MOV A, R0. Context determined which rule the assembler chooses to apply. The PUSH and POP operations apparently were considered cases where programmers might well be switching among register sets, as is likely in an interrupt response. Ordinary MOV’s were not so classified.

---

\(^6\) An assembler *could* decide that when you write the forbidden “MOV R1,R2” it would implement that as the permitted “MOV R1,2;” but that might be too high-handed for most people’s tastes: when you write assembly code you expect to get what you specify, not an assembler’s clever correction of your code.
1 What is assembly-language? Why bother using it?

It’s the quasi-English that humans use to speak to one another and to “assembler” programs, so as to define what operations we want a processor to carry out. “MOV R0, #38h,” for example, is the assembly-language way to express what in “machine code” is expressed as 78 38h. Assembly-language differs from higher-level languages, like C, in that one line of assembly language corresponds to one line of machine language. In C, a single line of code typically expresses a more complex operation, one that requires multiple lines of machine code (and assembly-language code).

Assembly-language programming is going out of style, because it costs too much, in human programming time. A common claim, in the programming industry, holds that a person can produce a couple of lines of debugged code in a day. If this is about right, then surely it pays to produce a couple of lines that get more done. We use assembly-language programming in this course because we like to see the one-to-one correspondence between the code we write and an action by the computer (though sometimes the “action” is invisible to us—as in the example given above, where a constant is loaded into a register that is internal to the 8051). Assembly language programs also require no overhead: no “libraries,” no special support routines.

The penalties for using assembly language will become obvious to you, as you work: the details are fussy and processor-dependent. The quirks and deficiencies of the 8051’s instruction set, in particular, are shocking—reflecting the old age of the 8051’s design (it was born around 1976). A higher-level language could hide these uglinesses from you; assembly-language, however, forces you to confront, for example, the amazing fact that the 8051 can increment its only 16-bit pointer, but cannot decrement that pointer. Horrible, but true.

Memory, internal and external A microcontroller includes some on-chip memory; most controllers rely on this memory exclusively. Our lab 8051 computer does not use the controller this way; we provide an external memory to hold both code and data. Our 8051 stores all of its program code and most of its data in external memory—the 32K RAM. Our controller includes no on-chip ROM at all, and only a few hundred bytes of on-chip RAM. (This lack of ROM is unusual, in a microcontroller; late in the course you will have a chance to try an 8051 that includes on-chip ROM.)

---

1Revisions: correct “MOVX @RN text and figure, to limit it to first two registers only; add figure to illustrate “MOVX @RN (12/04); add Port 2 circuit figure; fix typo in label of last pgm listing; add discussion of Rn pseudo-indexing (4/04); change SETB to CLR, in ext-bus LED drive code (5/03).
2 External memory and I/O access, for the 8051

The 8051’s way of doing business with the outside world through an external bus, as in our computer, is extremely rigid. This rigidity is a weakness—but for a student meeting the 8051 for the first time, the rigidity may be good news: you don’t need to learn 14 addressing modes (as for the 68000); one is available—well, about one and one half!

2.1 The usual method: use DPTR

Specifically, when the 8051 accesses data (in contrast to instructions) that is stored in external memory, or in I/O devices tied to the external buses, the controller nearly always uses the address stored in the “Data Pointer” (DPTR), an on-chip 16-bit register; and the on-chip source or destination of that data must be the A register—the “accumulator.” Here’s that same proposition, stated graphically:

![Diagram of DPTR and A for data accesses](image)

**Figure 1: 8051 uses DPTR and A to do data accesses, on external buses**

So, you know that nearly any\(^2\) input from an external device will be coded as

```
MO VX A, @DP TR;
```

And any output to an external device will be coded as

```
MO VX @DP TR, A;
```

Note the strange word-order: opcode, then destination, then source: “bite Tom dog.” The @ indicates that we want to copy the contents of A (an 8-bit register on the 8051) not into DPTR\(^3\) (or from DPTR, when the transfer is going in the opposite direction) but into (or from) the location to which DPTR points. DPTR is called a “pointer,” and this mode of addressing is called “indirect.”

Given the 8051’s reliance on DPTR, the program’s getting data to and from the right place depends not on the line of code that does the transfer (we’ve seen that those are rigidly fixed), but, instead, on setting up the DPTR properly beforehand.

---

\(^2\)With the exception noted below: using \(R_n\).

\(^3\)Incidentally, the operation wouldn’t quite make sense, anyway, since A holds 8 bits and DPTR holds 16.
Setting up the DPTR

Nearly every program that does a data transfer with the outside world, then, needs to initialize the DPTR. The code to do that always looks like

```
MOV DPTR,#PORT_ADDRESS;
```

...where PORT_ADDRESS is a 16-bit address. The # mark indicates an addressing mode known as immediate. This mode takes the value of PORT_ADDRESS (let’s suppose it is 8000h, to make things more concrete), and places that value into the 16-bit register, DPTR.

Notice, also, that this time there is no @, because this time we do intend to put the value into the DPTR (not where it points): “MOV DPTR...” is not a transaction with external memory or I/O device; it is an operation on a register within the 8051.

External moves (MOVX) contrasted with internal operations

That last point reminds us that even when the 8051 is set up to use external buses, as in our lab computer, it still can (and must) do operations on its internal registers.

```
MOV DPTR,#PORT_ADDRESS;
```

is one such operation, as we have said. In addition to such MOV’s, all other processor operations work on the processor’s internal registers: ADD’s, logical operations, bit tests, and so on. External operations are limited to the simple MOVX: just a transfer. In order to do something to a value picked up from an INPUT device—from the keypad, for example—one must always bring the value into the 8051 first, storing it in a register; only then can the processor do anything to the value.
2.2 A Nifty alternative: set base register and use $R_n$ as index

For operations accessing a few locations not too far apart (within 256 of each other) there is another way to proceed: one can hold the high part of the address fixed, and then use an 8-bit register to play with the lower part of address. This mode is well-suited to I/O addressing, where the several devices live close to one another—as in our little machine.

Using this mode, the 16-bit address, though off-chip is defined not by DPTR, but by P2—the 8051’s internally-defined “PORT 2,” just an on-chip special register, the one wired to the high half of the external address bus—and one of two of the 8 internal “Working Registers,” $R_0$ or $R_1$. Here’s the idea, showing use of Port 2 and register $R_1$ to define a 16-bit I/O address:

```
"MOVX @R1, A" illustrated
```

![Diagram of MOVX @R1, A](image)

Figure 2: Another way to define an external address: P2 and $R_n$

Here’s an example, reading ADC and writing to DAC, at ports 2 and 3 (using the external bus). First, we’ll show a listing using DPTR, then a listing using this second method (fixed high half, $R_0$ and $R_1$ providing lower half of address):

Here’s the way you might do it using DPTR:

```assembly
; ADC_DAC_dptr_404.a51 ADC to DAC using one DPTR  4/04
; read ADC, write to DAC, start ADC

$NOSYMBOLS
$INCLUDE (C:\MICRO\8051\RAISON\INC\REGS320.INC)
$INCLUDE (C:\MICRO\8051\RAISON\INC\VECTORS320.INC)

ORG 0H
AJMP STARTUP

ORG 380h

STARTUP: MOV DPTR #8003H ; point to ADC

PICKUP: MOVX A, @DPTR ; Read ADC (first pass, this gets bad data)
MOVX @DPTR, A ; start ADC for next time (ADC will convert while program is busy doing the 5 lines below)
DEC DPL ; point to DAC
MOVX @DPTR, A ; ...send sample to DAC
INC DPTR ; restore DPTR
SJMP PICKUP

END
```
...And here's the code using fixed High (P2) and \( R_n \) for the low half:

```assembly
; ADC_DAC.w1.Rn.a51 Try using R.n registers as index, in I/O transfer
; use explicit PORT2 drive to set up high half of address, while using
; R0, R1 to provide low-half of address

$NOSYMBOLS ; keeps listing short, lest...
$INCLUDE (C:\MICRO\RIDE\INC\REG320.INC) ; ...this line should produce huge list
 ; of symbol definitions (all '51 registers)

ORG 0 ; tells assembler the address at which to place this code

LJMP STARTUP ; here code begins--with just a jump to start of
 ; real program. ALL our programs will start thus

ORG 500h ; ...and here the program starts

STARTUP: MOV P2,#80h ; set up top half of address as I/O base

AGAIN: MOV R0,#3h ; set up 8-bit index register to ADC
  MOV R1,#2h ; ...and another register to point to DAC

TFR: MOVX A, @RO ; read ADC (bad, first pass, OK ever after)
  MOVX @R0,A ; start ADC for next pass
  MOVX @R1, A ; ...and send it to DAC

SJMP TFR ; and do it all again

END
```

The \textit{DPTR} code looks a little more compact—but it doesn't run quite so fast as the second method (since it includes an extra INC and DEC operation). That small difference may not matter. Probably a more important point in favor of the \( R_n \) method is simply that it leaves the DPTR free for other operations.

2.3 The hardware that allows this double use of Port 2, to define High 8 address lines

At a glance it seems impossible for us to control Port 2, defining the high addresses, without interfering with the operation of DPTR. The scheme works because of a switch (really, a 2:1 mux) that selects the source of the high-8 addresses: most of the time the switch is set to use DPTR; but it is set to use the output of the Port 2 latch (8 flops), when one uses the MOVX \( R_n \) form. Here's the circuit diagram, showing the switch. (I guess one could say that the switch \textit{had to be there} to permit this operation; but I feel reassured, seeing it in the diagram!)

![Circuit Diagram](image)

Figure 3: How Port 2 can use \textit{either} DPTR or Port 2 flops to define High-8 Address Lines
3 I/O Operations: external bus versus 8051's internally-defined ports ("Portpins")

3.1 Using the external buses

Most of the I/O that we do in the labs makes use of the external buses. Here, for example, is a program that takes in a byte from port 0 and writes the value back to the same port:

GETIT: MOVX A, @DPTR ; read the keypad
      MOVX @DPTR, A ; ...show it on display
      SJMP GETIT ; ...again, forever

This would work properly, of course, only if you had first loaded DPTR to point to keypad and display. You know how to do that.

3.2 Using the internally-defined ports

If you chose to use the internally-defined 8051 ports, you could do the same task. You could tie the keypad to port 1 (using it as an input), and the display to port 3 (using it as output). Then your code would be more compact than the code using the external buses: no need to pass the value through A, the accumulator, and no use of the DPTR:

GETIT: MOV P3, P1 ; read the keypad, send to display
       SJMP GETIT ; ...again, forever

This might not be a smart way to do the task, though: you would tie up half of the 8051's 32 I/O lines. The internally-defined ports look better when you use them for operations on bits, rather than on bytes.

4 BIT operations

4.1 BIT operations, using the external bus

The external bus provides the hard way to do the job, but it makes clear what hardware is at work (whereas using the 8051 port pins hides the flip-flops and drivers that are doing the magic). So, let's start with this method. Let's drive an LED at one bit of Port 1 (external bus), and sense an input bit at the same port. We'll toggle the LED if that input bit is high, hold the present LED state if the input bit is low.

In order to drive the LED, we need a flip-flop on the output bit. Without that flop the computer could only send a level for the very short duration of the OUT1* pulse. The flop sustains the LED drive level. To take in and test an input bit, the computer needs a 3-state buffer between the source of that input and the data bus. Here's a sketch of the hardware needed:
And here's some code to use this hardware:

```assembly
DDBITS: MOV DPTR, #8001h ; Point to LED and input bit
        CLR ACC.0 ; set up LED OFF state, for startup
SEND:  MOVX @DPTR, A ; send bit level to LED drive

TEST: MOVX A, @DPTR ; pick up the input bit (plus junk)
        JNB ACC.2,TEST ; Test the single interesting bit. Leave LED untouched if input is low
        CPL ACC.0 ; ...but toggle that bit if the input is high: set up the bit level, to be sent soon...
        SJMP SEND
```

Not bad; but we'll see in a moment that the same is easier to do with the 8051's port pins.

### 4.2 BIT operations, using 8051 port pins

**Hardware: practically nothing**

If, instead, we use the 8051's port pins, the hardware is nil: just connect the input switch and LED to two port pins:

One detail worth noticing is the changed LED drive: the 8051 can source almost no current (50µA), but can sink a lot more: 1.6 mA at Port 3 (still not a lot; Port 0 or 2 can sink double that—but they're not available to us, since we use them for the external bus). So, we had to drive the LED by *sinking* current—and we had to boost the R value a little, because the 8051 $I_{OL}$ is much less than the current available from an HC gate (either high-source—or low-sink). (This asymmetric drive—strong sink, feeble source—is characteristic of older devices, TTL and NMOS; the CMOS 8051 has chosen to follow this tradition.)
Code: simpler when using portpins

The code to test a bit and toggle a bit, as before, is simpler than when we used the external bus:

```
SETP3.0 ; start high: LED off
TEST:  JNB P3.1,TEST ; hang here while input bit is low
        ; ...fall through when goes high:
FLIPIT: CPL P3.0 ; flips only LSB (bit 0) of port
        ; SJMP TEST
```

This code also runs faster than the one written for the external bus, requiring far fewer bus accesses (both in execution, and in instruction fetches).
1 Subroutines: CALL

Suppose you write a patch of code, and would like to use it more than once. For example, in Lab 19 we need a software delay, in order to slow execution: the program makes the display value increment, and that incrementing has to be taken far below the processor’s full-speed rate in order to be intelligible to human eyes.

You can, of course, simply write the patch of code anew, each time you want to use it. The left-hand sketch, below, shows such clumsy in-line coding: the Delay code is written twice. The right-hand sketch begins to improve on the in-line arrangement—but raises the question, “how is execution to resume at the proper place, after Delay is invoked?”

![Figure 1: Inline coding can be clumsy; it would be nice to write once, use multiple times](image)

The computer needs a way to know where it should go after running the Delay code: where it should “return,” in computer lingo. Each time the computer goes off to the Delay code—or “routine”—it must make a temporary record of the return address.

Computers do this by automatically saving the return address on the stack—a region in RAM defined as “where the Stack Pointer Points”\(^2\). On the 8051, the Stack Pointer is an 8-bit register, and the Stack RAM must be on-chip 8051 RAM (not the RAM we have attached to the processor).

In order to take advantage of the processor’s ability to use the stack in this way, all we need to do is use the instruction, Call (in one of its two flavors) rather than a Jump operation. (The generic Call comes in two flavors on the 8051, the Jump comes in three\(^3\)

---

1Revisions: forced-priority section and example added; interrupt figure added (5/04).
2Does this sound like a circular definition? I don’t think it is, but I agree it sounds that way!
3ACALL does a relative branch, using an 11-bit offset value, so ACALL can branch within a 2K range, roughly plus-or-minus 1K; LCALL does a branch to a location specified by its 16-bit address, and thus can go anywhere in the 8051’s address space. You may recall that AJMP and LJMP present the same contrast; in addition, Jump comes in the short flavor, called SJMP (8-bit offset: range +127, -128).
Here's an illustration of the way the Stack can save a *return* address, when one uses Call:

![Diagram showing Stack usage](image)

*Figure 2: Subroutine Call knows how to get back, after running the subroutine*

The process of saving the return address is entirely automatic. You finish the called “subroutine” with a RET (return).

### 1.1 Stack as general-purpose short-term storage place

Call uses the Stack automatically and implicitly. But one can also use the Stack explicitly, to store the contents of registers, temporarily. Often, in fact, a subroutine needs to do just that, because the routine needs to make use of registers (especially A, the *accumulator*, the privileged register that always is involved in input and output (when using the external buses), and that is involved in most arithmetic and logical operations.

So, the Delay routine might save A and the “flags” (Carry, and a few others; the register is called PSW: “Program Status Word”). The code would look like:

```
DELAY: PUSH ACC ; A likes to be called ACC, for some operations. Annoying!
PUSH PSW ; this is a phoney--this routine doesn't affect the flags!
MOV A, #80h ; initialize with delay value
KILLTIME: DJNZ A,KILLTIME ; count down till A hits zero
POP PSW
POP ACC ; notice that I have to match the order of the stacking, as I unstack!
RET
```
2 Interrupt

Interrupt is a subroutine Call that is initiated not by a line of code (ACALL, for example), but by a hardware event: for example, by a logic Low on the INTO* pin. The 8051 will pay no attention to an interrupt unless one enables interrupts: one must enable both the particular interrupt source, and the global interrupt response.

![Diagram of Interrupt]

Figure 3: Interrupt Sketched

Here is code that enables external interrupt zero—and that also makes the INTO* pin edge-sensitive (this code appears in Lab 20):

```assembly
; ---NOW ENABLE INTERRUPTS---
SETB I0 ; make INTO Edge-sensitive (p. 22)
SETB EX0 ; ...and enable INTO
SETB EA ; Global int enable (pp.31-32)
```

Any interrupt response routine must end with a RETI command—very similar to RET: like RET, RETI pulls the return address off the stack. But RETI does one more task: it restores to its previous level the record showing the current level of interrupt priority. We'll meet the question of priority below. For now, just note that any interrupt response code must be terminated with RETI.

2.1 Where the ISR code lives: Interrupt “Vectors”

In response to an interrupt from any particular source\(^4\), the 8051 hops to a particular dedicated address, where it begins to execute code\(^5\). Each of these dedicated locations is called an interrupt “vector.” The vector for INTO is 03h; the vector for INT1 is 13h, for example.

Each vector allows only 8 bytes before the next vector, so only a very simple response routine—“interrupt service routine”: ISR—can be written there. If the ISR is longer than 8 bytes, the program must branch from the vector location, out to a memory location where there is room for a longer ISR.

---

\(^4\)Interrupts can be requested by any of various sources. INTO* or INT1* are external pins; the Dallas variant you are using, DS89C420, offers: a timer can finish counting and generate an interrupt; a serial port can demand attention by using an interrupt; there is even a software interrupt.

\(^5\)This is the simplest of several possible responses to an interrupt—and the only one of which the 8051 is capable. Later processors, like the 68000, offered more complex responses, such as a read of the data bus that would steer the processor to a particular response address.
2.2 Priority Among Interrupts

More than one interrupt can occur at one time—in two senses—and the processor needs a predictable response to such a coincidence. One sense of “more than one at a time” is more than one at the time when interrupt requests are sampled—at the end of each instruction. The other sense is a second request occurs while the processor is servicing an earlier request. We'll treat these two cases in succession.

Priority in First Sense: Two requests are found at once: “Natural priority” rules

The several interrupt sources have been assigned “natural priority” in the 8051, in order to resolve such ties—simultaneous requests. The table appears at p. 98 of the High Speed Microcontroller User's Guide.

The program below demonstrates use of two interrupts, with opposed effects:

```
; INT_natural_404.a51 show effect of "natural priority" among interrupts April '04
; should increment display each time interrupted by INTO, decrement for INT1

$NOSYMBS
$INCLUDE (C:\MICRO\8051\RAIISON\INC\REG320.INC) ; Raison's DS320 register defs. file
$INCLUDE (C:\MICRO\8051\RAIISON\INC\VECTORS320.INC) ; Tom's vectors definition file
STACKBOT EQU 07Fh ; put stack at start or scratch indirectly-addressable block (80h and up)
ORG 0h
LMP STARTUP
ORG 00h
STARTUP: MOV SP, #STACKBOT
MOV DPTR, #8000h

; -----NOW ENABLE INTERRUPTS-----
SETB ITO ; make INTO Edge-sensitive (p. 22)
SETB IT1 ; ditto for INT1
SETB PX1 ; give higher priority to INT1 (SFR "IPO", p. 25 of user's guide); name is from RIDE list, "IP"
SETB EX0 ; ...and enable INTO
SETB EX1 ; ...and INT1
SETB EA ; Global int enable

CLR A ; (for clean startup, as usual)

STUCK: MOVX @DPTR, A ; show display--constant, till interrupt inc's it
JMP STUCK ; (responds to falling edge--pseudo-edge sensitive,
; so you must clock several times while high, then low)

;--------------------
; ISR0: This is response to INT ZERO: INCREMENT A
ORG INTOVECTOR ; this is defined in VECTORS320.INC, included above.
; It is address 03h, the address to which micro hops
; in response to interrupt ZERO
ISR0: NOP ; pointless--but gives time to interrupt with higher priority
INC A
RETI

; ISR1: This is response to INT ONE: DECREMENT A
ORG INTOVECTOR ; this is defined in VECTORS320.INC, included above.
; It is address 13h, the address to which micro hops
; in response to interrupt ONE
ISR1: NOP
DEC A
RETI
END
```
If one interrupt request comes later than the other, that interrupter will have to wait until the ISR for the first-to-come has been completed. The processor does not record the fact that an interrupt request has been made and ignored. So, the interrupt requests must be held in hardware, off the 8051, in order to be sure of not being lost. An interrupt request could be used to clock an edge-triggered flop fed a constant high (our usual edge-sensing trick); Q¹ from the flop could drive the INT¹ request line, and the flop could be reset under program control when the processor reached the ISR for that interrupt.

Priority in a Second Sense: One interrupt can break into another’s ISR (Forced Priority)

If one is not satisfied to let “natural priority” rule—allowing any ISR to finish before even a higher-priority interrupt can be serviced—then one can assign higher priority to a particular interrupt. In the program below, we have assigned external INT1* higher priority than INT0*. As a result, INT1* will be allowed to break into the execution of the ISR for INT0*.

; INTPRIOR_D03.asm show priority for one int over another 12/03

; should increment display each time interrupted by INTO, decrement for INT1

$SYMBOLS
$INCLUDE (C:\MICRO\8051\RAISON\INC\REG320.INC) ; Raison’s DS310 register defs. file
$INCLUDE (C:\MICRO\8051\RAISON\INC\VECTORS320.INC) ; Tom’s vectors definition file
STACKBOT EQU 07Fh
; put stack at start of scratch indirectly-addressable block (80h and up)

ORG 0h
JMP STARTUP
ORG 000h
STARTUP: MOV SP, #STACKBOT
MOV DPTR, #8000h

; ------NOW ENABLE INTERRUPTS------
SETB ITO ; make INTO Edge-sensitive (p. 22)
SETB IT1 ; ditto for INT1
SETB PX1 ; give higher priority to INT1 (SFR "IP0", p. 25 of user’s guide); name is from RIDE list, "IP"...
SETB EXO ; ...and enable INTO
SETB EX1 ; ...and INT1
SETB EA ; Global int enable (pp.31-32)
SETB ET2;
CLR ET2;

CLR A ; (for clean startup, as usual)
STUCK: MOVX @DPTR, A ; show display--constant, till interrupt inc’s it
JMP STUCK ; (responds to falling edge--pseudo-edge sensitive,
; so you must clock several times while high, then low)

-------------------------------
; ISR0: This is response to INT ZERO: INCREMENT A
ORG INTOVECTOR ; this is defined in VECTORS320.INC, included above.
; It is address 03h, the address to which micro hops
; in response to interrupt ZERO
ISR0: NOP ; pointless—but gives time to interrupt wi higher priority
INC A
RETI

; ISR1: This is response to INT ONE: DECREMENT A
ORG INTOVECTOR ; this is defined in VECTORS320.INC, included above.
; It is address 13h, the address to which micro hops
; in response to interrupt ONE
ISR1: NOP
DEC A
RETI

END

(Call interrupt_404.txt; May 2, 2004)
The Weird Address Space of the Venerable 8051/2

```
"SPECIAL FREGISTERS"
  (ACC, B, PC, P2, DPTR, SP, SERIAL, TIMERS, FLAGS, INTERRUPTS, OTHER CONTROL...)

FFh  128 bytes (some bit addr)

FFh  128 bytes

80h

Scratchpad

7F  80 bytes

30h 20h 10h 08h 00h R0 R7

2AH AF

1FH 17H 0FH 07H reg bank 2 reg bank 1 reg bank 0

Direct addr

mov A, R2
inc A
setb P0.3

Indirect addr

mov A, @R0

"CODE" (read only)

"DATA"
(PSEN FALSE)
read/write
(strobes RD, WR)

external ROM

64K

external ROM

64K

2K 4K

(internal ROM)

"CODE" (read only)

copy (alias) of

4x 8 bytes

16 bytes: 128 bits

0

0

mov A, @R0

mov A, @R0"