EXAMPLE STEP

OBJECTIVE

This example has the following objectives:

- Review the use and control of stepper motors.
- Discuss the stepper motor energizing patterns, 1- and 2-phase energizing types, full- and half-step motion.
- Introduce the concept of creating quasi-continuous motion through a sequence of steps.
- Introduce the concept of sequential accessing a finite set of stored patterns through index addressing with a continuously updating pointer.
- Illustrate a method to ‘load’ a single-precision (2-hex) variable into a double precision (4-hex) register.
- Introduce the concept of automatic incrementation/decrementation of the addressing pointer with a programmable step size.
- Discuss the reset actions to be taken when the pointer hits the ‘roof’ or ‘floor’.

Figure 1  Stepper motor and its controller board. Control signals to the controller board are sent through the 8-pin input connector.

PROGRAM EX_STEP

This program is to be used for stepper motor control (Figure 1). A stepper motor is controlled by sending a binary pattern to its controller board. Eight distinct binary patterns are recognized by the stepper motor controller board in your lab. They are given in Table 1 as sequences S0 – S7.
The stepper motor has four distinct coils, and a 1 in the energizing pattern signifies that the corresponding coil is energized. Note that in the even sequences (S0, S2, S4, S6) only one bit in the energizing pattern is set. This indicates that only one out of the four stepper-motor coils (1 phase) is energized. On the other hand, in the odd sequences (S1, S3, S5, S7), two bit are set, i.e., two coils (2 phases) are energized. The sequences with two coils energized are labeled full step, while those with only one coil energized are labeled half step.

Table 1 Stepper motor energizing patterns and their 2-hex equivalent value

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Energizing pattern</th>
<th>8-bit 2-hex equivalence</th>
<th>Phase energizing type</th>
<th>Step type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>1000</td>
<td>$08</td>
<td>1 phase</td>
<td>Half step</td>
</tr>
<tr>
<td>S1</td>
<td>1001</td>
<td>$09</td>
<td>2 phase</td>
<td>Full step</td>
</tr>
<tr>
<td>S2</td>
<td>0001</td>
<td>$01</td>
<td>1 phase</td>
<td>Half step</td>
</tr>
<tr>
<td>S3</td>
<td>0101</td>
<td>$05</td>
<td>2 phase</td>
<td>Full step</td>
</tr>
<tr>
<td>S4</td>
<td>0100</td>
<td>$04</td>
<td>1 phase</td>
<td>Half step</td>
</tr>
<tr>
<td>S5</td>
<td>0110</td>
<td>$06</td>
<td>2 phase</td>
<td>Full step</td>
</tr>
<tr>
<td>S6</td>
<td>0010</td>
<td>$02</td>
<td>1 phase</td>
<td>Half step</td>
</tr>
<tr>
<td>S7</td>
<td>1010</td>
<td>$0A</td>
<td>2 phase</td>
<td>Full step</td>
</tr>
</tbody>
</table>

When these patterns are hit in increasing order, one after the other, the motion is called ‘half speed forward’. If only the odd sequences are hit, i.e., every second pattern is hit, the motion is ‘full speed forward’. If the patterns are hit in decreasing order, the motion is, respectively, ‘half-speed backward’ and ‘full speed backward’ (Table 2).

Table 2 Stepper motor speed definitions

<table>
<thead>
<tr>
<th>Step type</th>
<th>Step</th>
<th>Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half speed forward</td>
<td>+1</td>
<td>Anywhere</td>
</tr>
<tr>
<td>Full speed forward’</td>
<td>+2</td>
<td>Even pattern</td>
</tr>
<tr>
<td>Half-speed backward</td>
<td>-1</td>
<td>Anywhere</td>
</tr>
<tr>
<td>Full speed backward’</td>
<td>-2</td>
<td>Even pattern</td>
</tr>
</tbody>
</table>

To control a stepper motor, a parallel port is used to send 8-bit pattern to the stepper-motor control board. Note the pin assignment:

<table>
<thead>
<tr>
<th>MSB</th>
<th>N/A</th>
<th>N/A</th>
<th>N/A</th>
<th>I1 (white)</th>
<th>I2 (green)</th>
<th>I3 (gray)</th>
<th>I4 (yellow)</th>
</tr>
</thead>
</table>

The program illustrates the sequentially access to memory locations using a memory pointer:
• The memory locations to be accessed have the variable names S0, S1, S2, S3, S4, S5, S6, S7, and correspond to addresses $0000$ through $0007$.

• The memory pointer has the variable name POINTER and is stored at address $0009$

• The memory pointer is used in index Y addressing. The memory pointer is entered in index register Y, and the offset is taken as zero ($00$). Thus, through Y, the pointer controls directly the address to be accessed.

• The program automatically steps through the set of eight sequences S0—S7 and generates the next pointer value, by adding the variable STEP. As shown in Table 2, for stepper motor control, the values of STEP are +/-1 and +/-2. The positive values correspond to forward motion. The negative values correspond to backward motion. Two's complement 8-bit signed convention is used to generate the negative values (-1=$ff$, -2=$fe$).

• The program takes care not to send the pointer outside the eight-sequence range. Hence, when going forward, whenever the highest sequence, S7, is hit (i.e., we hit the roof) the program resets to pointer to S0 (i.e., to the floor). When going backward, whenever the floor is hit, the program resets the pointer to the roof.

In the initialization phase, the program enters the values of S0 – S7 given in Table 1 in the appropriate memory locations. Then it sets STEP=$00$, and POINTER=$00$

When running the program, you have to enter into memory location STEP one of the following options: STEP=$01$ or $02$, or $ff$ or $fe$.

FLOWCHART AND CODE
The program flowchart is show below. Two flowchart levels are presented: the big-picture and the details. The big-picture is used to understand the overall architecture of the program. The details are used to explain some of the blocks. (Details are given only for those blocks which are somehow new, and have not been used in previous programs.) Note that the main program is only intended to demonstrate the use of the subroutine STEP_SR. For this reason, the main program is a mere call to the STEP_SR subroutine. The essential code for this program is shown to the right of the flowchart. The essential code was incorporated into the standard template asm to generate the code file Ex_Step.asm.
Define variables

START

• Load POINTER to accB
• Add STEP

SWI

• Load $0000 into Y
• Add accB to Y (acc B contains the value of POINTER)

S0 1 byte
S1 1 byte
S2 1 byte
S3 1 byte
S4 1 byte
S5 1 byte
S6 1 byte
S7 1 byte
STEP 1 byte
POINTER 1 byte

Big-picture Flowchart

Define variables

Flowchart Details

• X=REGBAS
• S0=$08
• S1=$09
• S2=$01
• S3=$05
• S4=$04
• S5=$06
• S6=$02
• S7=$0a
• STEP=$00
• POINTER=$00

Code

* Define program variables
ORG DATA
S0 RMB 1
S1 RMB 1
S2 RMB 1
S3 RMB 1
S4 RMB 1
S5 RMB 1
S6 RMB 1
S7 RMB 1
STEP RMB 1
POINTER RMB 1

*Main program
ORG PROGRAM
START LDX #REGBAS
LDAA #$08
STA S0
LDAA #$09
STA S1
LDAA #$01
STA S2
LDAA #$05
STA S3
LDAA #$04
STA S4
LDAA #$06
STA S5
LDAA #$02
STA S6
LDAA #$0a
STA S7

* Initialize controls
LDAA #0
STA STEP
LDAA #$00
STA POINTER

* Start looping
LABEL0 JSR STEP_SR
BRA LABEL0
SWI

* Step subroutine
STEP_SR

LDAB POINTER
ADDB STEP
CMPB #7
BGE LABEL4
LDAB #0

LABEL3

LDY #$0000
ABY
LDAA $00,Y
STAA PORTB,X

RTS
EXECUTION

Open THRSim11. Close the Commands window. View CPU registers AND memory list. Open and assemble Ex_Step.asm. Set breakpoint at SWI. Reset registers. Press the RESET button. Set standard labels (Label/Set Standard Labels). Set breakpoint at LABEL0 and at SWI. Arrange windows for maximum benefit: Your screen should look like this:
Let the program run to the first breakpoint. Initialization of the memory variables is achieved. Scroll the screen to have the LABEL0 line at the top. Rearrange the windows slightly to cover only essential code. Your screen should look like this:
i) **CASE ‘SLOW-FORWARD’ (STEP=$01)**

a) Put STEP=$01. This should make the program go forward in increments of one (half-speed forward). Your screen should look like this:
b) Step through the program. Notice that the program goes directly to the subroutine STEP_SR. The following events happen:

- The current value of the variable POINTER (POINTER=$00) is loaded into accB.
- The value of variable STEP (STEP=$01) is added to accB. The result is accB = $01
- The value in accB is compared with the roof (i.e., perform accB-$07). The result of the comparison is a negative number. Hence, the condition code bit N is set (observe N=1 on the simulator status bar, and bit 3 = 1 in CC register). Your screen looks like this:
- At branch BLE LABEL3, the branch condition is met (since N=1) and the program branches to LABEL3.

- The value of accB is now compared with the floor (i.e., perform accB-$00). The result of the comparison is a $01. Hence, the condition code bit N is clear (observe N=0 on the simulator status bar, and bit 3 = 0 in CC register). Your screen looks like this:
• At branch BGE LABEL4, the branch condition is met (since N=0) and the program branches to LABEL4.

• The value of accB is stored as the new value of POINTER (observe POINTER at memory location $0009 having the value $01)

• At this stage, the value of POINTER must be loaded into index register Y. To achieve this, the value $0000 is loaded in index register Y. Then, the index register Y and the accB (containing the current value of POINTER) are added. (This convoluted way of doing things was needed because there is no simple way to load the single precision variable, POINTER, into the double precision register Y.) Your screen looks like this:

![Screen showing step subroutine](attachment:image.jpg)
The value of the memory where the POINTER points to is now loaded into accA. This is achieved using index addressing with Y as the index register and $00 as the offset (i.e., LDAA $00,Y). In our case, POINTER=$01, hence the memory location $0001 is loaded. This is the step S1, which has the value S1=$09. Hence, you should see the value $09 in accA (i.e., A $09 in CPU registers). Finally, the pattern S1, which was loaded in accA, is sent to Port B. This will make the stepper motor go to the position corresponding to pattern S1. Your program looks like this:

```
(c) Let your program perform another and observe that the POINTER and the value in Port B have reached the next sequence pattern, S2=$01. Your screen should look like this:
```
d) Repeat and observe each loop until the POINTER and the value in Port B have reached the last sequence pattern, S7=$0a. Your screen should look like this:
e) In the next loop, the program should hit the roof and return to the floor. Perform this loop manually, to verify the logic. At the end of the loop, observe that the POINTER and the value in Port B have come back to the first sequence pattern, S1=$08. Your screen should look like this:
ii) **CASE ‘FULL-FORWARD’ (STEP=$02)**

Put the value $02 into the variable STEP and repeat the pattern of observations covered before. You will see that only the odd patterns (S1, S3, S5, S7) are being hit. According to Table 1, these patterns correspond to 2-phase energizing of the stepper motor (full steps). Satisfy yourself that you fully understand the logic of the program and that you have fully observed how the program is generating a full-speed forward motion.

iii) **CASE ‘HALF-BACKWARD’ (STEP=$FF)**

Put the value $ff into the variable STEP and repeat the pattern of observations covered before. You will see all the patterns (S0--S7) being hit, but in reverse order (S1, S0, S7, S6, S5, S4, S3, S2, S1 ...). Satisfy yourself that you fully understand the logic of the program and that you have fully observed how the program is generating a half-speed backward motion.

iv) **CASE ‘FULL-BACKWARD’ (STEP=$FE)**

Put the value $fe into the variable STEP and repeat the pattern of observations covered before. You will see that only the odd patterns are being hit, and that they are hit in reverse order (S1, S7, S5, S3, S1, ...). Satisfy yourself that you fully understand the logic of the program and that you have fully observed how the program is generating a full-speed backward motion.
WHAT YOU HAVE LEARNED

In this example, you have learned:

- The use and control of stepper motors.
- The stepper motor energizing patterns, 1- and 2-phase energizing types, full- and half-step motion. Learn the eight sequences S0—S7, the odd sequences S1, S3, S5, S7 being full steps (2-phase), the even sequences S0, S2, S4, S6 being half-steps (1-phase)
- The concept of creating quasi-continuous motion through a sequence of steps.
- The concept of sequential accessing a finite set of stored patterns through index addressing with a continuously updating pointer. The eight sequences S0—S7, were stored in memory, and the variable POINTER was used to address them. The POINTER was loaded into index register Y, and the memory locations containing the sequences S0—S7 were addressed using the index Y addressing mode.
- A method to ‘load’ a single-precision (2-hex) variable into a double precision (4-hex) register. This situation was encountered in connection with the use of variable POINTER for index Y addressing. Since there is no direct way to store a single precision variable into a double precision register, a convoluted way was used. The double precision register Y was first loaded with zeros ($0000) and then added with the single precision variable POINTER. Thus, the single precision variable POINTER ended up being effectively loaded into index register Y.
- The concept of automatic incrementation/decrementation of the addressing pointer with a programmable step size. The variable STEP was used with values 1, 2, -1, -2. The positive values generated forward motion at half and full speed; the negative values created backward motion. Incrementation and decrementation was done in a loop. Whenever the incremented value hit a value outside the permissible range (0 to 7), reset was implemented. If the ‘roof’ was hit (i.e., a value greater than 7) the reset was to the floor (i.e., to 0). If the ‘floor’ was hit (i.e., less than 0) the reset was implemented to the roof (i.e., to 7).
(This page is left intentionally blank)