- **Using the stack and the stack pointer**
    - The Stack and Stack Pointer
    - The stack is a memory area for temporary storage
    - The stack pointer points to the last byte in the stack
    - Some instructions which use the stack, and how data is saved and retrieved off the stack
    - Subroutines and the stack
    - An example of a simple subroutine
    - Using a subroutine with PORTA to make a binary counter on LEDs

TIPS FOR WRITING PROGRAMS

1. Think about how data will be stored in memory.

- Draw a picture

2. Think about how to process data

- Draw a flowchart

3. Start with big picture. Break into smaller parts until reduced to individual instructions

- Top-down design

4. Use names instead of numbers

The Stack and the Stack Pointer

- Sometimes it is useful to have a region of memory for temporary storage, which does not have to be allocated as named variables.

- When we use subroutines and interrupts it will be essential to have such a storage region.

- Such a region is called a *Stack*.

- The **Stack Pointer** (SP) register is used to indicate the location of the last item put onto the stack.

- When you put something onto the stack (**push onto the stack**), the SP is decremented before the item is placed on the stack.

- When you take something off of the stack (**pull from the stack**), the SP is incremented after the item is pulled from the stack.

- Before you can use a stack **you have to initialize the Stack Pointer** to point to one value higher than the highest memory location in the stack.

- For the MC9S12 put the stack **at the top of the data space**

  – For most programs, use $1000 through $2000 for data.
  – For this region of memory, initialize the stack pointer to $2000.
  – If you need more space for data and the stack, and less for your program, move the program to a higher address, and use this for the initial value of the stack pointer.

- Use the **LDS** (Load Stack Pointer) instruction to initialize the stack pointer.

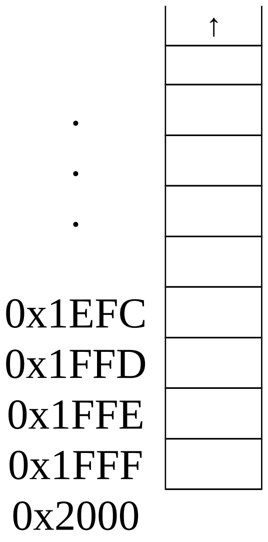
- The LDS instruction is usually the first instruction of a program which uses the stack.

- The stack pointer is **initialized only one time** in the program.

- For microcontrollers such as the MC9S12, it is up to the programmer to know how much stack his/her program will need, and to make sure enough space is allocated for the stack.

<u>If not enough space is allocated the stack can overwrite data and/or code, which will cause the program to malfunction or crash.</u>

The stack is an array of memory dedicated to temporary storage



SP points to the location last item placed in block

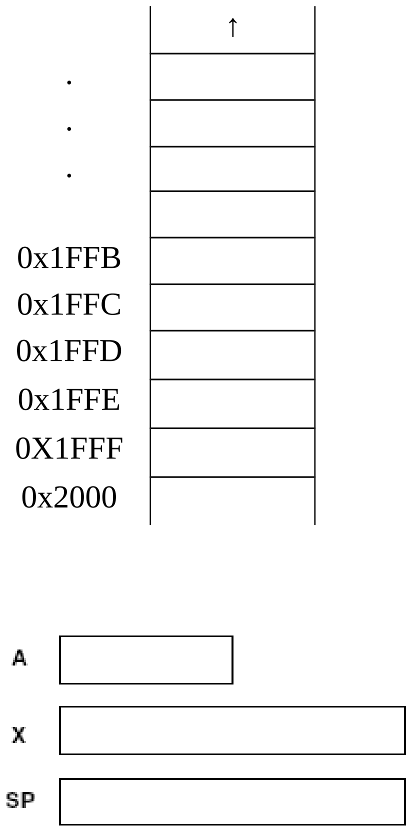
SP **decreases** when you put an item on stack

SP **increases** when you pull item from stack

For HC12 EVBU, use **0x2000** as initial SP:

```
STACK:  EQU   $2000
        LDS   #STACK
```

An example of some code which uses the stack



Stack Pointer

Initialize ONCE before first use (LDS #STACK)

Points to last used storage location

Decreases when you put something on stack

Increases when you take something off stack

```
STACK:  equ   $2000
CODE:   org   $2000

        lds    #STACK
        ldaa   #$2e
        ldx    #$1254
        psha
        pshx
        clra
        ldx    #$ffff
```

CODE THAT USES A & X

```
        pulx
        pula
```

PSHA Push A onto Stack PSHA

Operation: $(SP) - \$0001 \Rightarrow SP$
$(A) \Rightarrow M_{(SP)}$

Description: Stacks the content of accumulator A. The stack pointer is decremented by one. The content of A is then stored at the address the SP points to.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHA	INH	36	Os	Os

PSHX Push Index Register X onto Stack PSHX

Operation: $(SP) - \$0002 \Rightarrow SP$
$(X_H : X_L) \Rightarrow M_{(SP)} : M_{(SP+1)}$

Description: Stacks the content of index register X. The stack pointer is decremented by two. The content of X is then stored at the address to which the SP points. After PSHX executes, the SP points to the stacked value of the high-order half of X.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHX	INH	34	OS	OS

PULA

Pull A from Stack

Operation: $(M_{(SP)}) \Rightarrow A$
$(SP) + \$0001 \Rightarrow SP$

Description: Accumulator A is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine, to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
PULA	INH	32	ufO	ufO

PULX

Pull Index Register X from Stack

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L$
$(SP) + \$0002 \Rightarrow SP$

Description: Index register X is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
PULX	INH	30	UfO	UfO

Subroutines

- A subroutine is a section of **code which performs a specific task**, usually a task which needs to be executed by different parts of a program.

- Example:
  - Math functions, such as *square root*

- Because a subroutine can be called from different places in a program, you cannot get out of a subroutine with an instruction such as

jmp label

because you would need to jump to different places depending upon which section of code called the subroutine.

- When you call a subroutine, your code saves the address where the subroutine should return to. It does this by saving the return address on the stack.

  - This is done automatically for you when you get to the subroutine by using the **JSR** (Jump to Subroutine) or **BSR** (Branch to Subroutine) instruction. This instruction **pushes the address** of the instruction following the **JSR**/**BSR** instruction **on the stack**.

- After the subroutine is done executing its code, it needs to return to the address saved on the stack.

- This is done automatically for you when you return from the subroutine by using the **RTS** (Return from Subroutine) instruction. This instruction **pulls the return address off of the stack** and loads it into the program counter, so the program resumes execution of the program with the instruction following that which called the subroutine.

The subroutine will probably need to use some MC9S12 registers to do its work. However, the calling code may be using its registers for some reason - the calling code may not work correctly if the subroutine changes the values of the MCs9S12 registers.

- To avoid this problem, the subroutine should save the MC9S12 registers before it uses them, and restore the MC9S12 registers after it is done with them.

JSR Jump to Subroutine JSR

Operation:

$(SP) - \$0002 \Rightarrow SP$
$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$
$\text{Subroutine Address} \Rightarrow PC$

Description:

Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction following the JSR as a return address.

Decrements the SP by two to allow the two bytes of the return address to be stacked.

Stacks the return address. The SP points to the high order byte of the return address.

Calculates an effective address according to the rules for extended, direct, or indexed addressing.

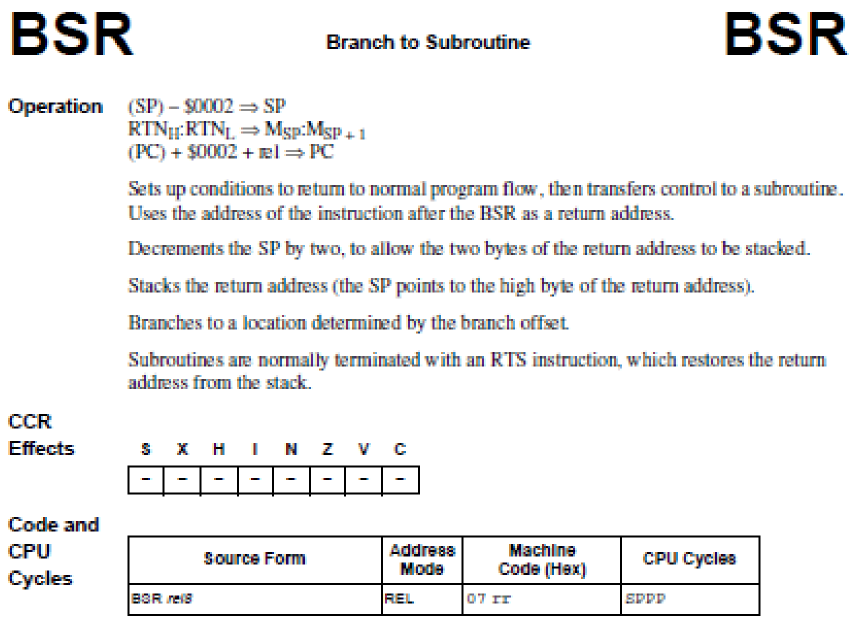
Jumps to the location determined by the effective address.

Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
JSR *opr8a*	DIR	17 dd	SPPP	PPPS
JSR *opr16a*	EXT	16 hh ll	SPPP	PPPS
JSR *oprx0_xysp*	IDX	15 xb	PPPS	PPPS
JSR *oprx9,xysp*	IDX1	15 xb ff	PPPS	PPPS
JSR *oprx16,xysp*	IDX2	15 xb ee ff	fPPPS	fPPPS
JSR [D,*xysp*]	[D,IDX]	15 xb	fIfPPPS	fIfPPPS
JSR [*oprx16,xysp*]	[IDX2]	15 xb ee ff	fIfPPPS	fIfPPPS



BSR

Branch to Subroutine

BSR

Operation

$(SP) - \$0002 \Rightarrow SP$
$RTN_H{:}RTN_L \Rightarrow M_{SP}{:}M_{SP+1}$
$(PC) + \$0002 + rel \Rightarrow PC$

Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction after the BSR as a return address.

Decrements the SP by two, to allow the two bytes of the return address to be stacked.

Stacks the return address (the SP points to the high byte of the return address).

Branches to a location determined by the branch offset.

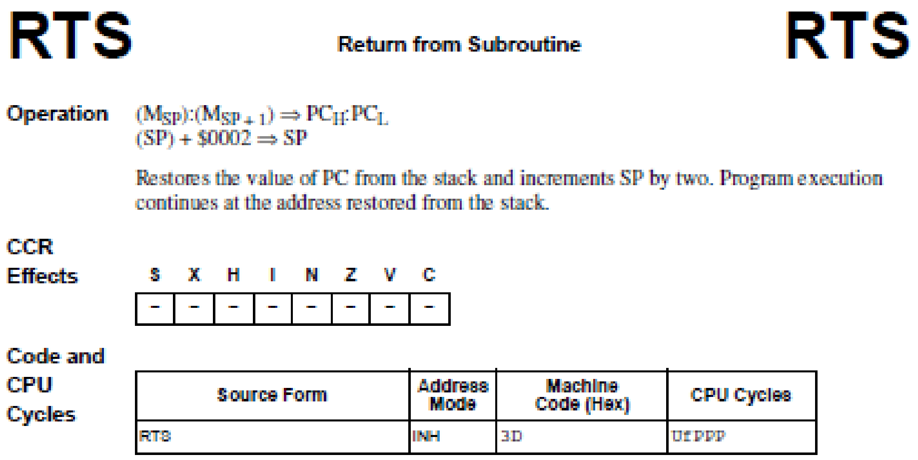
Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BSR *rel8*	REL	07 rr	SPPP



RTS

Return from Subroutine

RTS

Operation

$(M_{SP}){:}(M_{SP+1}) \Rightarrow PC_H{:}PC_L$
$(SP) + \$0002 \Rightarrow SP$

Restores the value of PC from the stack and increments SP by two. Program execution continues at the address restored from the stack.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
RTS	INH	3D	UfPPP

Example of a subroutine to delay for a certain amount of time

```
; Subroutine to wait for 100 ms
delay:      ldaa    #100        ; execute outer loop 100 times
loop2:      ldx     #8000       ; want inner loop to last 1ms
loop1:      dbne   x,loop1      ; inner loop – 3 cycles x 8000 times
            dbne    a,loop2
            rts
```

- Want inner loop to last for 1 ms. MC9S12 runs at 24,000,000 cycles/second, so 1 ms is 24,000 cycles.

- Inner loop should be 24,000 cycles/ (3 cycles/loop) = 8,000 loops (times)

- **Problem:** The subroutine changes the values of registers A and X

- To solve this problem, save the values of A and X on the stack before using them, and restore them before returning.

```
; Subroutine to wait for 100 ms
delay:      psha              ; save registers
            pshx
            ldaa   #100       ; execute outer loop 100 times
loop2:      ldx    #8000      ; want inner loop to last 1ms
loop1:      dbne x,loop1      ; inner loop – 3 cycles x 8000 times
            dbne  a,loop2
            pulx              ; restore registers in opposite order
            pula
            rts
```

```
; Program to make a binary counter on LEDs
; The program uses a subroutine to insert a delay between counts
; Does not work on Dragon12-Plus.  Need to write to PTJ to
; enable LEDs

prog:      equu  $2000
```

```
; Program to make a binary counter on LEDs
; The program uses a subroutine to insert a delay between counts
; Does not work on Dragon12-Plus.  Need to write to PTJ to
; enable LEDs

prog:     equ   $2000
data:     equ   $1000
STACK:    equ   $2000
PORTB:    equ   $0001
DDRB:     equ   $0003

          org prog

          ; ------------------------
          ; code to enable LEDs
          ;-------------------------

          lds   #STACK   ;  initialize stack pointer
          ldaa  #$ff     ;  put all ones into DDRB
          staa  DDRB     ;  to make PORTB output
          clr   PORTB    ;  put $00 into PORTB
loop:     jsr   delay    ;  wait a bit
          inc   PORTB    ;  add one to PORTB
          bra   loop     ;  repeat forever
```

```
; Subroutine to wait for a few milliseconds

delay:    psha              ; save registers
          pshx
          ldaa   #100       ; Execute outer loop 100 times
loop2:    ldx    #8000      ; Want inner loop to last 1 ms
loop1:    dbne x,loop1      ; Inner loop – 3 cyclesx8000 times
          dbne  a,loop2
          pulx              ; restore registers
          pula
          rts
```

Another example of using a subroutine

Using a subroutine to wait for an event to occur, then take an action.

- Wait until bit 7 of address $00CC is set.

- Write the value in ACCA to address $00CF.

```
; This routine waits until the MC9S12 serial port is ready, then
; sends a byte of data to the MC9S12 serial port

putchar:  brclr  $00CC,#$80,putchar
          staa   $00CF
          rts
```

- Program to send the word hello, world! to the MC9S12 serial port

```
; Program fragment to write the word “hello, world!” to the
; MC9S12 serial port

        ldx  #str
loop:   ldaa 1,x+         ; get next char
        beq  done         ; char == 0 => no more
        jsr  putchar
        bra  loop
done:   swi

str:    dc.b “hello, world!”
        fc.b $0A,$0D,0 ; LF CR
```

Here is the complete program to write a message to the screen

```
prog:     equ  $2000
data:     equ  $1000
stack:    equ  $2000
SCI0SR1:  equ  $00CC          ; SCI0 status reg 1
SCI0DRL:  equ  $00CF          ; SCI0 data reg low

          org prog
          lds  #stack
          ldx  #str
loop:     ldaa 1,x+           ; get next char
          beq  done           ; char == 0 ⇒ no more
          jsr  putchar
          bra  loop
done:     swi
```

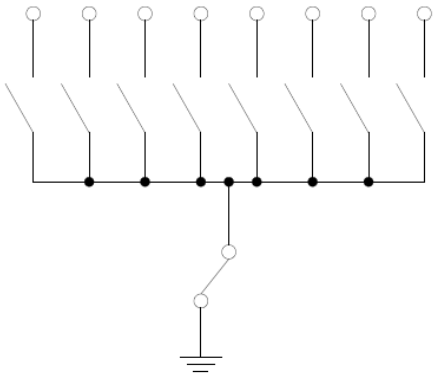
```
putchar:    brclr  SCI0SR1,$80,putchar  ; check for SCI ready
            staa   SCI0DRL              ; put character onto SCI
                                        ; port
            rts

            org    data
str:        fcc    "hello, world"
            dc.b   $0a,$0d,0           ; LF CR terminating zero
```

Using DIP switches to get data into the MC9S12

- DIP switches make or break a connection (usually to ground)

DIP Switches on Breadboard



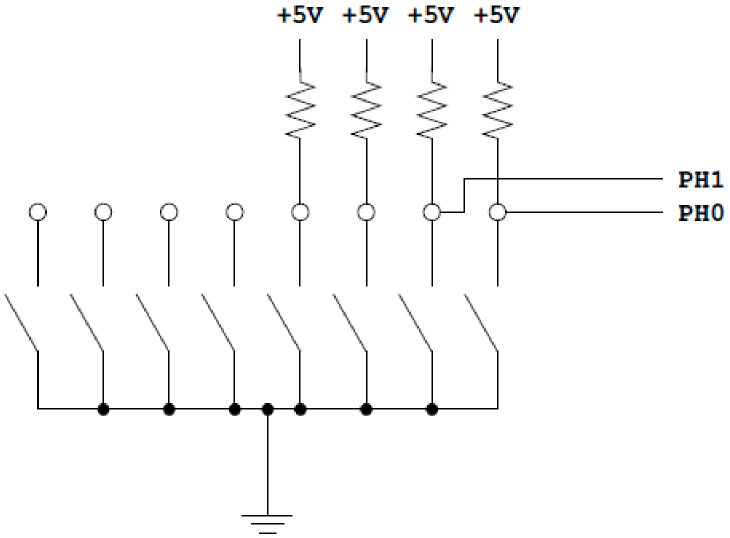
- To use DIP switches, connect one end of each switch to a resistor

- Connect the other end of the resistor to +5 V

- Connect the junction of the DIP switch and the resistor to an input port on the MC9S12.

- The Dragon12-Plus has eight DIP switches connected to Port H (PTH) (these switches have already resistors connected to them in the new Dragon12-Plus).

Using DIP Switches



- When the switch is **open**, the input port sees a **logic 1** (+5 V)

- When the switch is **closed**, the input sees a **logic 0** (0 V)

Looking at the state of a few input pins

- Want to look for a particular pattern on 4 input pins
  - For example want to do something if pattern on PH3-PH0 is 0110
- Don’t know or care what are on the other 4 pins (PH7-PH4)
- Here is the wrong way to doing it:

```
ldaa   PTH
cmpa  #$06
beq    task
```

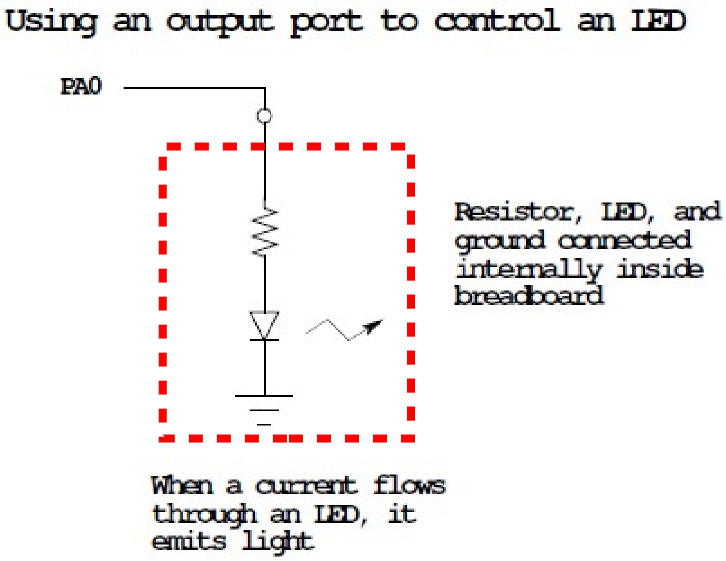
- If PH7-PH4 are anything other than 0000, you will not execute the task.
- You **<u>need to mask out the Don’t Care bits before</u>** checking for the pattern on the bits you are interested in

```
ldaa  PTH
anda #%00001111
cmpa #%00000110
beq   task
```

- Now, whatever pattern appears on PH7-4 is ignored

Using an MC9S12 output port to the 7-segment LEDs

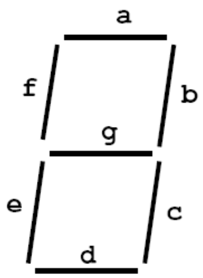
• Each of the segments in the 7-segment LEDs are connected to an output pin.



• To generate a pattern on each of the 7-segment LEDs, we need to set to a logic 1 the LEDs connected to specific pins.

Making a pattern on a seven-segment LED

- Want to generate a particular pattern on a seven-segment LED:



- Determine a number (hex or binary) which will generate each element of the pattern:
  - For example, to display a 0, turn on segments a, b, c, d, e and f, or bits 0, 1, 2, 3, 4 and 5 of PTB. The binary pattern is 0011 1111, or $3f.

  - For example, to display a 2, turn on segments a, b, d, e and g, or bits 0, 1, 3, 4, and 6 of PTB. The binary pattern is 0101 1011, or $5b.

  - To display numbers 0 2 4 6 8 on the 4 7-segment LEDs, the hex numbers are $3f, $5b, $66, $7d, $7f.

- Put the numbers in a table

- Go through the table one by one to display the pattern

- When you get to the last element, repeat the loop

```
; Program to display a pattern or lights
; on a 7-segment display
; First need to disable LEDs and enable 7-segment displays

prog:     equ  $2000
data:     equ  $1000
stack:    equ  $2000
PORTB:    equ  $0001
DDRB:     equ  $0003

          org prog

          lds   #stack        ; Initialize stack pointer
          ldaa  #$ff          ; Make PTB output
          staa  DDRB          ; 0xFF -> DDRB
l1:       ldx   #table        ; Start pointer at table
l2:       ldaa  1, x+         ; Get value; point to next
          staa  PORTB         ; Update 7-seg LEDs
          jsr   delay         ; Wait a bit
          cpx   #table_end         ; More to do?
          bls  l2             ; Yes, keep going through table
          bra  l1             ; At end; reset pointer

delay:    psha               ; Save A and X
          pshx
          ldaa  #100         ; Delay for 100 ms
loop2:    ldx   #8000
loop1:    dbne  x,loop1
          dbne  a,loop2
          pulx               ; Restore X and A
          pula
          rts                ; Return from subroutine
```

```
            org  data
table:      dc.b     $3f       ; 0
            dc.b     $5b       ; 2
            dc.b     $66       ; 4
            dc.b     $7d       ; 6
table_end:  dc.b     $7f       ; 8
```