Introduction

A data structure describes how information is organized and stored in a computer system. Although data structures are usually presented in the context of large computers, the same principles can be applied to embedded 8-bit processors. The efficient use of appropriate data structures can improve both the dynamic (time-based) and static (storage-based) performance of microcontroller software.

This application note presents data structures which are useful in the development of microcontroller software. The applications presented here are by no means absolute. One can find an infinite variety of ways to apply these basic data structures in a microcontroller application.

Strings

A string is a sequence of elements accessed in sequential order. The string data structure usually refers to a sequence of characters. For example, a message which is to be output to a display is stored as a string of ASCII character bytes in memory.
Storing Strings

A string of elements must be identified by a starting and ending address. A starting address for a string can be defined using an absolute address label or by using a base address of a group of strings and identifying particular strings with an offset into the group.

There are several methods of terminating string information. One common way of terminating a string is by using a special character to mark the end of the string. One terminating character to use is the value $04$, an ASCII EOT (end-of-transmission) byte.

Figure 1 shows an example of string data.

![Table of String Data](image)

Figure 1. String Data Structure

Another method of terminating a string is to identify its length. Its length can then be used as a counter value, eliminating the need for an extra byte of storage for the end of the string.

A string of ASCII characters can be terminated without using an extra byte of storage by using the sign bit (most significant bit) as an indicator of the last byte of the string. Because ASCII character data is only seven bits long, the last byte of a string can be indicated by a 1 in its most significant bit location. When using this method, the programmer must be careful to strip off the sign bit before using the ASCII character value.
Accessing Strings

An efficient way of accessing a string is with the indexed addressing mode and the INCX or DECX instruction. Listing 1. String Storage Example and Listing 2. String Access Example illustrate this string storage scheme and how to use it.

Listing 1. String Storage Example

*-----------------------------------------------------------------------------------
* Absolute string addresses
* One way of specifying string data
*-----------------------------------------------------------------------------------
Message1 FCB 'This is a string'
FCB $04
Message2 FCB 'This is another string'
FCB $04

*-----------------------------------------------------------------------------------
* Indexed string addressing
* Another way of specifying string data
*-----------------------------------------------------------------------------------
Msgs EQU *
Message3 EQU *-Msgs
FCB 'This is a string'
FCB $04
Message4 EQU *-Msgs
FCB 'This is another string'
FCB $04

Listing 2. String Access Example

*-----------------------------------------------------------------------------------
* String display code
* A generic method of displaying an entire string.
*-----------------------------------------------------------------------------------
LoadMsg LDX #Message1 ;Offset into X
Loop LDA Messages,X ;Load character
CMP #$04 ;Check for EOT
BEQ StringDone ;End of string
JSR ShowByte ;Show character
INCX ;Point to next
BRA Loop

*-----------------------------------------------------------------------------------
* String storage code
*-----------------------------------------------------------------------------------
Messages EQU *
Message1 EQU *-Messages
FCB 'This is a string'
FCB $04
String Applications

Practical applications of strings include storing predefined "canned" messages. This is useful for applications which require output to text displays, giving users information or prompting users for input.

Strings are also effective for storing initialization strings for hardware such as modems. Strings may also store predefined command and data sequences to communicate with other devices.

Stacks

A stack is a series of data elements which can be accessed only at one end. An analogy for this data structure is a stack of dinner plates; the first plate placed on the stack is the last plate taken from the stack. For this reason, the stack is considered a LIFO (last in, first out) structure. The stack is useful when the latest data is desired. A stack will typically have a predefined maximum size.

shows a representation of a stack.

Figure 2. Stack Data Structure
Just like a physical stack of items, the software stack has a bottom and a top. Software should keep track of the location of the top of the stack. This address can either point to the first piece of valid data or it can point to the next available location. For the following examples it will be pointing to the next available location.

### Stack Reading and Writing

The read operation of a stack is called "pulling" (or "popping"), and the write operation of a stack is called "pushing." When one pulls data from the stack, the data is removed from the stack and the stack pointer is adjusted. When data is pushed onto the stack, the stack pointer is adjusted and the data is added to the stack.

So, in the implementation of Figure 2, a push operation would first decrement the stack pointer and then store the data to the address pointed to by (stack pointer) +1. A pull operation would retrieve the data at (stack pointer) +1 and then increment the stack pointer.

Two error conditions are intrinsic to this data structure; underflow and overflow. A stack underflow occurs when a user attempts to pull information off an empty stack. A stack overflow occurs when a user attempts to push information onto a stack which is full. When using this data structure, these conditions should be attended to. An underflow condition should return an error. On an overflow, one can either reject the data and return an error, or the stack can "wrap" around to the bottom, destroying the data at the bottom of the stack.

### MCU Hardware Stack

Motorola MCUs utilize a stack structure for saving program context before transferring program control. This interaction may be the result of a jump or interrupt. As a result of an interrupt, the stack is used to push the values in the X, A, and CCR (condition code register) registers, as well as the 16-bit PC (program counter) value. When a jump instruction is encountered, the PC value is pushed on to the stack. On returning from an interrupt (RTI instruction) the program context (registers and PC) are pulled from the stack. When returning from a jump (RTS instruction) the PC is pulled from the stack.
HC05 Stack

The HC05 Family of MCUs have limited stack access. The only operation that can be performed with the MCU’s stack pointer is to reset it. The RSP instruction will reset the stack pointer to $FF. The HC05 stack pointer also has a limited size of 64 bytes. When the stack pointer grows beyond address $C0, the stack pointer wraps around to $FF, destroying any existing data at that address.

HC08 Stack

The HC08 Family of MCUs has a more flexible stack structure. The stack pointer can be set to any address. The HC08 MCUs also have an added addressing mode which is indexed by the stack pointer. In this way, a user can pass parameters to subroutines using the hardware stack, accessing the parameters using stack pointer indexed addressing.

Other HC08 Family instructions allow data to pushed on and pulled off the stack. The stack pointer can also be transferred to the X index register and vice-versa. With the addition of these instructions and addressing modes, a user has good control over the stack in the HC08 MCU.

Stack Applications

A stack is useful for dynamically allocating memory or passing parameters to and from subroutines. Typically, MCU RAM variables are statically allocated at assembly time.

For example:

```assembly
; Statically allocated RAM variables
ORG RAMSPACE
MyVar1 RMB 1
MyVar2 RMB 1
MyVar3 RMB 2
; Another method to statically allocate variable
MyVar4 EQU RAMSPACE+4
MyVar5 EQU RAMSPACE+5
```

This is appropriate for global variables, which need to be available throughout the program flow. However, for local variables which are only used in specific subroutines, this method is not the most efficient. The RAM space these variables use can be dynamically allocated using a software stack or MCU stack, freeing up RAM memory. The same
method can be applied to subroutine input and output parameters, passing them on the stack instead of in the A or X register.

Listing 3. Software Stack shows a software implementation of a stack, which would be appropriate for the HC05 Family of microcontrollers.
Listing 3. Software Stack

* STACK.ASM
* A simple software stack implementation Simply shows the PUSH and PULL operations on
* a stack; not intended to be a complete application.
* StackPtr points to next (empty) available location
* Written for the MC68HC705P6A MCU

* Memory map equates

RAMSPACE EQU $50
ROMSPACE EQU $100
RESETVEC EQU $1FFE

* Stack equates

STACKSIZE EQU $08
STACKBOT EQU $70           ;Bottom of software stack
STACKMAX EQU (STACKBOT-STACKSIZE+1)           ;Maximum address of stack

* RAM variables

ORG RAMSPACE           ;First address of RAM
StackPtr RMB 1           ;Pointer to next stack byte

* Start of program code

ORG      ROMSPACE          ;Start of ROM
Init LDA      #STACKBOT         ;Initialize the stack pointer
STA      StackPtr

* Some simple read and write operations
* For illustration only

LDA      #$01
JSR      PushA            ;Write to stack
BCS      FullErr
JSR      PushA            ;Write to stack
BCS      FullErr
JSR      PushA            ;Write to stack
BCS      FullErr
JSR      PushA            ;Write to stack
BCS      FullErr
JSR      PushA            ;Write to stack
BCS      FullErr
JSR      PushA            ;Write to stack
BCS      FullErr
JSR      PushA            ;Write to stack
BCS      FullErr
JSR      PushA            ;Write to stack
BCS      FullErr
JSR      PushA            ;Write to stack
BCS      FullErr
BCS FullErr
JSR PushA ;Write to FULL stack
BCS FullErr
JSR PushA ;Write to FULL stack
BCS FullErr
JSR PullA ;Read from stack
BCS EmptyErr
JSR PullA ;Read from stack
BCS EmptyErr
JSR PullA ;Read from stack
BCS EmptyErr

Loop BRA * ;Your code here
EmptyErr BRA * ;Your code here
FullErr BRA * ;Your code here

*---------------------------------------------------------------
* Subroutines - The code to access the data structure
*---------------------------------------------------------------
*---------------------------------------------------------------
* PUSH subroutine
* Push the contents of the accumulator onto stack
* Use C bit of CCR to indicate full error
*---------------------------------------------------------------
PushA LDX StackPtr ;Get stack pointer
CPX #STACKMAX ;Check for full stack
BLO Full
DECX ;Decrement stack pointer
STA 1,X ;Store data
STX StackPtr ;Record new stack pointer
CLC ;Clear carry bit
RTS ;Return
Full SEC ;Set carry bit for error
RTS ;Return

*---------------------------------------------------------------
* PULL subroutine
* PULL a byte off the stack into accumulator
* Use C bit of CCR to indicate empty stack error
*---------------------------------------------------------------
PullA LDX StackPtr ;Get stack pointer
CPX #STACKBOT ;Check for empty stack
BEQ Empty
LDA 1,X ;Get data
INCX ;Increment stack pointer
STX StackPtr ;Record stack pointer
CLC ;Clear carry bit
RTS ;Return
Empty SEC ;Set carry bit
RTS ;Return

*---------------------------------------------------------------
* Vector definitions
*---------------------------------------------------------------
ORG RESETVEC
FDB Init
Using the software stack, a subroutine can allocate variables by pushing (allocating) bytes on the stack, accessing them with indexed addressing (relative to the stack pointer variable) and pulling them (deallocating) before returning. In this way, the same RAM space can be used by multiple subroutines.

Parameters can be passed to and from subroutines as well. An input parameter can be pushed on the stack. When a subroutine is entered, it can access the input parameter relative to the stack pointer. By the same token, a subroutine can push an output parameter onto the stack to be passed back to the calling routine.

The MCU hardware stack and stack pointer can also be used for these purposes. Because of the expanded instruction set, the use of the MCU stack is easily exploited in the HC08 Family of microcontrollers. **Listing 4. Using the HC08 Stack Operations** shows an example of using the HC08 MCU stack to pass parameters and allocate local variables.

### Listing 4. Using the HC08 Stack Operations

Using the stack to pass parameters and allocate variables optimizes memory usage.

*------------------------------------------------------------------------*
* Code segment example of using the HC08 stack to pass parameters and    *
* allocate local variables.                                               *
* Not intended to be a complete application.                            *
*------------------------------------------------------------------------*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA #$AA</td>
<td>Load some data to be passed</td>
</tr>
<tr>
<td>PSHA</td>
<td>Push parameter for subroutine</td>
</tr>
<tr>
<td>PSHA</td>
<td>Push parameter for subroutine</td>
</tr>
<tr>
<td>JSR Sub</td>
<td>Call subroutine</td>
</tr>
<tr>
<td>PULA</td>
<td>Parameter passed back</td>
</tr>
<tr>
<td>STA Result2</td>
<td>Parameter passed back</td>
</tr>
<tr>
<td>PULA</td>
<td>Parameter passed back</td>
</tr>
<tr>
<td>STA Result1</td>
<td></td>
</tr>
<tr>
<td>Loop BRA *</td>
<td>Your code here</td>
</tr>
</tbody>
</table>
* Subroutine which uses the stack for variable access

* SP--->

* LOCAL2
* LOCAL1
* PCH
* PCL
* PARAM2
* PARAM1

PARAM1   EQU      6        ;Parameters passed in
PARAM2   EQU      5
LOCAL1   EQU      2        ;Local variables
LOCAL2   EQU      1

Sub      PSHA              ;Allocate local variable
          PSHA              ;Allocate local variable
          LDA      PARAM1,SP ;Load the parameter passed in
          ROLA               ;Do something to it
          STA      LOCAL1,SP ;Store in a local variable
          LDA      PARAM2,SP ;Load the parameter passed in
          ROLA               ;Do something to it
          STA      LOCAL2,SP ;Store in a local variable
          LDA      LOCAL1,SP
          STA      PARAM1,SP ;Store value to be passed back
          LDA      LOCAL2,SP
          STA      PARAM2,SP ;Store value to be passed back
          PULA               ;Deallocate local variable memory
          PULA               ;Deallocate local variable memory
          RTS                ;Return
Queues

A queue is a series of elements which accepts data from one end and extracts data from the other end. An analogy for this data structure would be a checkout line at the supermarket. The first people in are the first people out. For this reason, it is considered a FIFO (first in, first out) structure. This is useful when accessing data in the order it is received. A queue will usually have a predefined maximum size.

Figure 3 illustrates a queue.

![Figure 3. Queue](image)

Reading and Writing

The read operation of a queue is called "dequeue," and the write operation is "enqueue." Two pointers are necessary for a queue, one for the head of the line, and one for the tail. For an enqueue operation, after checking the size of the queue, the data is stored at the location pointed to by the "put" pointer, and the put pointer is adjusted. For a dequeue operation, the data is read from the "get" pointer location, and the pointer is adjusted.

Queues usually have a fixed size, so it is important to keep track of the number of items in the queue. This can be done with a variable containing the size of the queue or with pointer arithmetic.
Queue Errors

As with the stack structure, a queue can be subject to underflow and overflow errors. The write, or "enqueue" operation, should be non-destructive and should error if the queue is full. The read, or "dequeue" operation, should be destructive (remove the data element) and should error if the queue is empty.

Queue Applications

A practical application of a FIFO queue is for a data buffer. Queues can be used as buffers for transmitted or received data and for use with printers or serial communication devices.

Listing 5. Queue Example shows an example of queue software. A good application for this would be to store data received from the SIOP (serial input/output port) for processing later.

Listing 5. Queue Example

*-----------------------------------------------------------------------------------
* Illustrates an implementation of a queue For the 705P6A
*-----------------------------------------------------------------------------------
* Register definitions

* Memory map definitions
RAMSPACE EQU $50
ROMSPACE EQU $100
RESETVEC EQU $1FFE

* Queue data structure definitions
* These three equates defines the data structure
* To change the queue, change the data structure,
* and not the code.
QMAX EQU !4 ;Maximum Q size
QTOP EQU $A0 ;Top of Q array
QBOT EQU QTOP+QMAX-1 ;Bottom of Q array

* RAM variables

<table>
<thead>
<tr>
<th>ORG</th>
<th>RMB</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TempA</td>
<td>RMB</td>
<td>1</td>
</tr>
<tr>
<td>TempX</td>
<td>RMB</td>
<td>1</td>
</tr>
</tbody>
</table>
| GetPtr| RMB  | 1              ;8-bit pointer
| PutPtr| RMB  | 1              ;8-bit pointer
| QCount| RMB  | 1              ;Counter for Q size


* Start of program code

ORG       ROMSPACE
Start     EQU      *
InitQ     LDA      #QTOP    ;Initialize Q pointers and variables
            STA      GetPtr
            STA      PutPtr
            CLR      QCount

* Write and read from the Q
* A good application of this is to place bytes received
* from the SCI into the queue, and retrieve them later

JSR     DeQ
LDA     #$FF
JSR     EnQ
SR      EnQ
JSR     EnQ
JSR     EnQ
JSR     EnQ
JSR     EnQ
SR      DeQ
LDA     #$55
JSR     EnQ
JSR     EnQ
Loop    BRA      *

* Subroutines

* EnQ - enqueues a data byte passed in accumulator A
* Checks for a full Q, and returns a set carry bit if full.
* Otherwise returns a cleared carry bit on successful enqueue.

EnQ     STX      TempX    Save X register contents
            LDX      QCount    ;Check for a full Q
            CMPX     #QMAX    ;Q full error
            BNE      QFull
            LDX      PutPtr
            STA      0,X      ;Store the data in A
            CMPX     #QBOT    ;Check for wrap
            BEQ      WrapPut
            INX
            BRA      EnQDone
WrapPut  LDX      #QTOP    ;Successful enqueue
EnQDone  STX      PutPtr   ;Store new put pointer  
LDX      TempX    ;Restore X register  
INC      QCount   ;Increment count variable  
CLC               ;Clear carry bit  
RTS               ;Return  

;Unsuccessful enqueue  
QFull    LDX      TempX    ;Restore X register  
SEC               ;Set carry bit  
RTS               ;Return  

*-----------------------------------------------------------------------------------  
* DeQ - Dequeue a byte from the queue, and return the byte in the accumulator A.  
* If the queue is empty,  
* return a set carry bit to indicate an error. Otherwise,  
* return a clear carry bit and the data in A.  
*-----------------------------------------------------------------------------------  
DeQ      STX      TempX    ;Save X register contents  
LDX      QCount   ;Check for empty Q  
CMPX     #$00  
BEQ     QEmpty  
LDX      GetPtr  
LDA      0,X  
CMPX     #QBOT    ;Check for wrap condition  
BEQ      WrapGet  
INCX  
BRA      DeQDone  
WrapGet  LDX      #QTOP  
                  ;Successful dequeue  
DeQDone  STX      GetPtr  
LDX      TempX    ;Record new get pointer  
DEC      QCount   ;Decrement Q counter  
CLC               ;Clear carry bit  
RTS               ;Return  

QEmpty     LDX      TempX    ;Restore X register  
SEC               ;Set carry bit  
RTS               ;Return  

*-----------------------------------------------------------------------------------  
* Vector definitions  
*-----------------------------------------------------------------------------------  
ORG      RESETVEC  
FDB      Start  

AN1752 — REV 1  
MOTOROLA  
15
MACQ (Multiple Access Circular Queue)

A multiple access circular queue (or circular buffer) is a modified version of the queue data structure. It is a fixed-length, order-preserving data structure, and always contains the most recent entries. It is useful for data flow problems, when only the latest data is of interest. Once initialized it is always full, and a write operation always discards the oldest data.

Figure 4 depicts a MACQ.

Reading and Writing

After being initially filled, a write operation will place new data at the top of the MACQ, and shift existing data downward. The last byte will be discarded, so the result is the latest data existing in the buffer.

A read operation is non-destructive and can return any number of data bytes desired from the MACQ.

Figure 4. Result of a MACQ Write
Applications

A MACQ is useful for data streams which require the latest data and can afford to have a destructive write operation. For example, to predict the weather a forecaster might use temperature readings from the last five days to predict the next day’s temperature. Daily temperature readings can be recorded in a MACQ, so the latest data is available.

MACQs are also useful for digital filters. For example, they can be used to calculate a second derivative, running average, etc.

Example

Listing 6. MACQ illustrates the implementation of a MACQ or circular buffer. This could be effectively used for storing A/D converter readings. In this way, the latest A/D conversion results would be accessible through the circular buffer.

Listing 6. MACQ

*-----------------------------------------------------------------------------------
* Illustrates an implementation of a multiple-access circular queue. (MACQ)
* The MACQ is a fixed-length, order-preserving, indexable data structure.
* Once initialized, the MACQ is always full.
* A write to the MACQ is destructive, discarding the oldest data.
* A read from the MACQ is non-destructive. For the 705P6A
*-----------------------------------------------------------------------------------
* Register definitions

* Memory map definitions
RAMSPACE EQU $50
ROMSPACE  EQU $100
RESETVEC  EQU $1FFE

* MACQueue data structure definitions
* These three equates defines the data structure
* To change the queue, change the data structure, and not the code.
QSIZE    EQU      8                 ;Maximum Q size
QTOP     EQU      $A0               ;Top of Q array
QBOT     EQU      QTOP+QSIZE-1      ;Bottom of Q array

* RAM variables

TempA    RMB      1
TempX    RMB      1
TempData RMB      1
QPtr     RMB      1        ;8-bit pointer
* Start of program code

ORG ROMSPACE
Start EQU *
InitQ LDA #QBOT ;Initialize Q pointer
STA QPtr

* Write and read from the MACQ
* A useful application of this would be to store A/D converter readings, so the latest * n readings are available.

LDA #$55
JSR WriteQ
LDA #$56
JSR WriteQ
LDA #$57
JSR WriteQ
LDA #$58
JSR WriteQ
LDA #$AA
JSR WriteQ
LDA #$AB
JSR WriteQ
LDA #$AC
JSR WriteQ
LDA #$AD
JSR WriteQ
LDA #0
JSR ReadQ
LDA #1
JSR ReadQ
LDA #2
JSR ReadQ
Loop BRA *

* Subroutines

* WriteQ, A contains data to be written write is destructive on full Q, once initialized Q is always full.

WriteQ STX TempX ;Store X register value
LDX QPtr ;Load Q pointer
CMPX #QTOP-1 ;See if Q is full
BEQ QFull
STA 0,X ;Store data to Q
Tables

A table can be viewed as a vector of identically structured lists. A table is a common way of storing "lookup" data, such as display data or vector bytes.

**Figure 5** shows an example of a table.
A table is commonly used to look up information. Table entries can be accessed with an offset from the base address of the table. Therefore, a read from a table is typically done by computing the offset of the desired data and accessing it using an indexed addressing mode.

**Table Applications**

The table data structure is common in MCU applications. One way of using tables is to perform character conversions. For example, a table can be used to convert binary numbers to BCD equivalents. For LCD (liquid crystal display) displays, an ASCII character byte may need to be converted to segment bitmaps for the display. A table could be used for this purpose.

Another application of a table is a "jump" table. This is a table of vector values which are addresses to be loaded and vectored to. Some program parameter can be converted to an offset into a jump table, so the appropriate vector is fetched for a certain input.

For example, in their memory maps Motorola MCUs have a built-in vector table, used for interrupt and exception processing. These vector tables allow preprogrammed addresses to be defined for certain MCU exceptions. When an exception occurs, a new program counter value is fetched from the appropriate table entry.
Another way of utilizing the table data structure is to store predefined values for lookup. An example of this is storing interpolation data in a table, to perform mathematical functions. This use of a table is documented in the application note, *M68HC08 Integer Math Routines*, Motorola document order number AN1219.

Another example involves using a table of sinusoidal values to produce sine wave output as in the application note *Arithmetic Waveform Synthesis with the HC05/08 MCUs*, Motorola document order number AN1222. If an equation to calculate data is CPU-intensive and can be approximated with discrete values, these values can be precalculated and stored in a table. In this way, a value can be quickly fetched, saving CPU time.

**Table Example**

Listing 7. Table is an example of the use of tables to convert ASCII data to LCD segment values.

**Listing 7. Table**

```
*-----------------------------------------------------------------------------------
* Code segment example of using a table to store LCD segment values
* Could be used when 2 data registers define the segment values for a display position.
* Takes in an ASCII character, converts it to an offset into the table of segment values, and uses the offset to access the segment bitmap values.
*-----------------------------------------------------------------------------------
Loop    LDA Character   ;Load an ASCII character
        JSR Convert    ;Convert the character
        TAX            ;Offset into table is in A
        LDA 0,X        ;Load the first byte
        STA LCD1       ;Store to data register
        LDA 1,X        ;Load the second byte
        STA LCD2       ;Store to data register
        BRA Loop       ;Repeat

*-----------------------------------------------------------------------------------
* Convert ASCII character byte in A to an offset value into the table of LCD segment values. Valid ASCII values are (decimal): 32-47, 48-57, 65-90
*-----------------------------------------------------------------------------------
Convert  CMP #!48        ;Check for "special" character
        BLO Special     ;Check for numeric character
        CMP #!65        ;Check for numeric character
        BLO Numeric     ;Check for invalid value
Alpha    CMP #!90        ;Check for invalid value
        BHI ConvError
```

SUB       #!39       ;Convert to table offset
BRA       ConvDone
Special   CMP       #!32       ;Check for invalid value
BLO       ConvError
SUB       #!32       ;Convert to table offset
BRA       ConvDone
Numeric   CMP       #!57       ;Check for invalid value
BHI       ConvError
SUB       #!32       ;Convert to table offset
RA        ConvDone
ConvError CLRA                  ;Invalid value shows as blank
ConvDone  ROLA                  ;Multiply offset by 2
RTS                   ;2 bytes data per LCD position

*-----------------------------------------------------------------------------------
* Lookup table of LCD segment values for ASCII character values
* Some characters can not be displayed on 15-segment LCD, so they are marked as
* invalid, and will be displayed as a blank space.
*-----------------------------------------------------------------------------------
Table      FDB     $0000     ;' '  
FDB     $0000     ;'!' INVALID  
FDB     $0201     ;'"'  
FDB     $0000     ;'#' INVALID  
FDB     $A5A5     ;'$'  
FDB     $0000     ;'%' INVALID  
FDB     $0000     ;'&' INVALID  
FDB     $0001     ;'('  
FDB     $5000     ;')'  
FDB     $F00F     ;'*'  
FDB     $A005     ;'+'
FDB     $0000     ;',' INVALID  
FDB     $2004     ;'-'  
FDB     $0800     ;'.'
FDB     $4002     ;'/'  
FDB     $47E2     ;'0'
FDB     $0602     ;'1'
FDB     $23C4     ;'2'
FDB     $2784     ;'3'
FDB     $2624     ;'4'
FDB     $21A8     ;'5'
FDB     $25E4     ;'6'
FDB     $0700     ;'7'
FDB     $27E4     ;'8'
FDB     $27A4     ;'9'
FDB     $2764     ;'A'
FDB     $8785     ;'B'
FDB     $01E0     ;'C'
FDB     $8781     ;'D'
FDB     $21E4     ;'E'
FDB     $2164     ;'F'
Linked Lists

A list is a data structure whose elements may vary in precision. For example, a record containing a person’s name, address, and phone number could be considered a list. A linked list is a group of lists, each of which contains a pointer to another list.

Figure 6 represents a linked list.
Each list in the structure contains the same type of information, including a link to the next item in the list. The link might be an absolute address or an offset from some base address. In a doubly linked list, pointers are kept to both the next and the previous item in the list. A linked list can be traversed easily by simply following the pointers from one list to the next.

**Linked List Applications**

A linked list is used traditionally to define a dynamically allocated database, in which the elements can be ordered or resorted by adjusting the links. However, in a small microcontroller, there are more appropriate applications of linked lists.

A linked list can be used as a structure for a command interpreter. Each command could contain the string of characters, an address of a subroutine to call on that command, and a link to the next command in the linked list. In this way, a command string could be input, searched for in a linked list, and appropriate action taken when the string is found.

**State Machines**

Another useful application of a linked list is to define a state machine. A state machine can be represented by a discrete number of states, each of which has an output and pointers to the next state(s). See Figure 7.
A state machine can be considered a Mealy or a Moore machine. A Mealy machine’s output is a function of both its inputs and its current state. A Moore machine has an output dependent only on its current state.

This state machine model can be useful for controlling sequential devices such as vending machines, stepper motors, or robotics. These machines have a current internal state, receive input, produce output, and advance to the next state.

One can first model a process as a sequential machine, then convert this behavior to a linked-list structure and write an interpreter for it. An important goal is to be able to make modifications to the state machine by changing the data structure (linked list) and not the code.

**State Machine Example**

As an example, consider a traffic light controller which determines the light patterns for an intersection. Two light patterns are needed, one for the north/south directions and one for the east/west directions. Consider that the bulk of traffic travels on the north/south road, but sensors are placed at the east/west road intersection to determine when traffic needs to cross. See **Figure 8**.
Figure 8. Traffic Light Controller Example

This example can be modeled as a Moore state machine, with its output a function of its current state. The next state is a function of the current state and the state of the input. Figure 9 shows a state graph for this example. The initial state will be a green light in the north/south direction and a red light in the east/west direction. The controller remains in this state, until input is seen in the east/west direction. The flow continues as shown in the diagram. The output shown in the diagram is a pattern for the light array to activate the lights for the state.
**Simulation**

This example can be simulated using LEDs and a 68HC705P6A MCU. A pushbutton switch can be used to simulate the input sensor. Figure 10 illustrates the simulation circuit. Using six bits of an output port, a pattern can be generated to display the appropriate north/south and east/west lights (LEDs). Table 1 shows the bitmap in this application.
Figure 10. Circuit Simulation of Traffic-Light Controller

Table 1. Traffic Light Bitmap for Port A

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meaning</td>
<td>Not used</td>
<td>North/South Signal</td>
<td>East/West Signal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>G</td>
<td>Y</td>
<td>R</td>
<td>G</td>
<td>Y</td>
<td>R</td>
</tr>
</tbody>
</table>

With the hardware in place, all that is left is to define the state machine in software. This can be done by implementing a linked-list data structure and the code to access and interpret the machine.

For this particular example, each list in the data structure defines the current state of the traffic light. Each list contains:

1. The byte which is the bitmap for the lights.
2. A delay value; the time the controller remains in the state
3. The next state pointer for an input of 0
4. The next state pointer for an input of 1
The main loop of the program should execute the program flow charted in Figure 11. The software for this simulated traffic light controller is documented in Listing 8. Traffic Controller State Machine.

**Figure 11. State Machine Program Flow**

**Listing 8. Traffic Controller State Machine**

```
*-----------------------------------------------------------------------------------
* Traffic light controller example Illustrates a linked list implementation of a state
* machine For the 705P6A
*-----------------------------------------------------------------------------------
* Register definitions
PORTA   EQU      $00
PORTD   EQU      $03
DDRA    EQU      $04
DDRD    EQU      $07

* Memory map definitions
RAMSPACE EQU      $50
ROMSPACE EQU      $100
RESETVEC EQU      $1FFE
* RAM variables
ORG      RAMSPACE
```

AN1752 — REV 1

MOTOROLA
TempA    RMB      1
TempX    RMB      1

*-----------------------------------------------------------------------------------
* Start of program code
*-----------------------------------------------------------------------------------

Start    LDA     #$00
          STA      PORTA             ;Predefine output levels
          LDA      #$FF
          STA      DDRA              ;Make Port A all outputs
          BCLR     7,PORTD          ;Make Port D pin 0 an input
          LDX      #INITST
          Loop     LDA      STATES+LIGHTS,X  ;Get light pattern
          STAA     PORTA ;Output light pattern
          LDA      STATES+DELAY,X    ;Get delay in seconds
          JSR      SecDelay          ;Cause delay
          BRCLR    7,PORTD,In0       ;Check for input of 0
In1      LDX      STATES+NEXT0,X    ;Get next state offset
          BRA      Loop              ;(input = 1)
In0      LDX      STATES+NEXT1,X    ;Get next state offset
          BRA      Loop              ;(input = 0)

*-----------------------------------------------------------------------------------
* DATA STRUCTURE FOR STATE MACHINE LINKED LIST (05)
* Offsets and base address scheme is adequate for a small table (<255 bytes)
*-----------------------------------------------------------------------------------

LIGHTS   EQU      0                 ;Offset for light pattern
DELAY    EQU      1                 ;Offset for time delay
NEXT0    EQU      2                 ;Offset for pointer 0
NEXT1    EQU      3                 ;Offset for pointer 1
STATES   EQU      *                 ;Base address of states
INITST   EQU      *-STATES          ;Initial state offset

* North/South green light, East/West red light
NSG      EQU      *-STATES          ;Offset into STATES
          FCB      %11011110         ;Output for state
          FCB      !10               ;Delay for state
          FCB      NSG               ;Next state for input of 0
          FCB      NSR               ;Next state for input of 1

* N/S yellow light, E/W red light
NSY      EQU      *-STATES          ;Offset into STATES
          FCB      %11101110
          FCB      !5
          FCB      NSR
          FCB      NSR

* N/S red light, E/W green light
NSR      EQU      *-STATES          ;Offset into STATES
          FCB      %11110011
          FCB      !5               ;Delay for state
          FCB      EWY

AN1752 — REV 1
FCB EYW
* E/W yellow light, N/S red light
EWY EQU *-STATES
FCB %11110101  ;Delay for state
FCB !5
FCB NSG
FCB NSG

* Delay subroutines
*-----------------------------------------------------------------------------------
* Cause a delay of ~(1 second * Accumulator value) @ fop = 1MHz
*-----------------------------------------------------------------------------------
SecDelay CMP #$00
BEQ SecDone
JSR Delay0
JSR Delay0
DECA
BRA SecDelay
SecDone RTS

* Cause a delay of ~1/2 of a second
*-----------------------------------------------------------------------------------
Delay0 STX TempX
LDX #$B2

DLoop0 CMPX #$00
BEQ DDone0
JSR Delay1
DECX
BRA DLoop0
DDone0 LDX TempX
RTS

* Cause about 2.8msec delay @ fop of 1MHz
*-----------------------------------------------------------------------------------
Delay1 STA TempA
LDA #$FF
DLoop1 CMP #$00
BEQ DDone1
DECA
BRA DLoop1
DDone1 LDA TempA
RTS

* Vector definitions
*------------------------------------------------------------------
ORG RESETVEC
FDB Start
The use of data structures is not necessarily limited to large, complicated computers. Although the data structure is a powerful concept in such a context, the same principles can be applied to smaller processors such as 8-bit microcontrollers.

The code to implement these data structures does not necessarily have to be complex or confusing. The goal of programming should be to modularize commonly used functions, so that they may be reused in other applications with a minimal amount of modification.

The appropriate use of data structure concepts can improve the static and dynamic performance of an MCU application, without affecting its portability or legibility.