Document Number: AN3334
Rev. 0, 11/2006

Data Structures for RS08 Microcontrollers

by: Inga Harris
8-bit Microcontroller Applications Engineer
East Kilbride, Scotland

1 Introduction

This application note presents data structures useful in developing microcontroller software. You can apply these basic data structures in a microcontroller application.

A data structure describes how information is organized and stored in a computer system. Although data structures are usually presented in the context of 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.

The RS08 core differs from other Freescale 8-bit cores, in that it does not have a stack pointer or index register (data structures use both). Software can recover these feature, as shown in this application note. For other Freescale 8-bit core examples, refer to Freescale document-order number AN1752.

Contents

The code in this application note is written for the MC9RS08KA2 and tested using CodeWarrior™ 5.1 software and the DEMO9RS08KA2 board.

2 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 output to a display is stored in memory as a string of ASCII character bytes.

2.1 Storing Strings

A start and end address identify a string of elements. A string's starting address can be defined in two ways: using an absolute address label or a base address with an offset.

You can terminate string information in several ways. One common way is by using a special character to mark the end of the string. One terminating character is $04, which is an ASCII EOT (end-of-transmission) byte.

Figure 1 shows an example of string data.

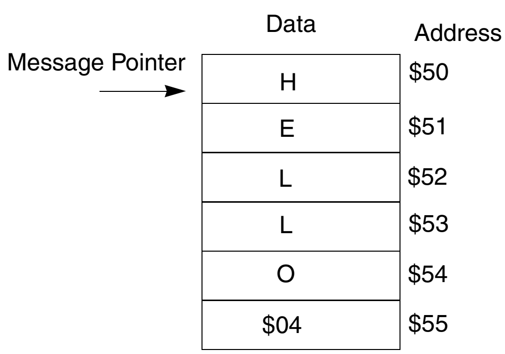

Figure 1. String Data Structure

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.

If you use the sign bit (the most significant bit) to indicate the last byte of the string, you can terminate a string of ASCII characters without using an extra byte of storage. 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, strip off the sign bit before using the ASCII character value.

2.2 Accessing Strings

An efficient way to access a string is with the indexed addressing mode and the INC or DEC instructions.

String storage and access:

```
;*********************************************************************
;* String Display Code                                               *
;* A generic method of displaying an entire string                   *
;*********************************************************************
                        ORG ROMStart
_Startup:
mainLoop:               LDA #Message
                        TAX
Loop                    LDA $0E                   ;Load Accumulator with the
                                                  ;contents of the memory address
                                                  ;pointed to by X
                        CMP #$04                  ;Is it EOT?
;User needs to write following routines
                        ;BEQ StringDone
                        ;JSR ShowByte
                        INCX                      ;Move to next byte
                        BRA  Loop

;********************** ***********************************************
;* String Storage Example                                            *
;* String is stored in RAM                                           *
;*********************************************************************
                        ORG RAMStart
Message                 EQU *

Message1                DC.B 'This is a string'
                        DC.B $04
Message2                DC.B "This is another string"
                        DC.B $04
```

2.3 String Applications

Practical applications of strings include storing predefined canned messages. This is useful for applications requiring 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.

3 Stacks

A stack is a series of data elements 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 last-in, first-out (LIFO) structure. The stack is useful when the latest data is desired. A stack typically has a predefined maximum size.

Figure 2 shows a representation of a stack.

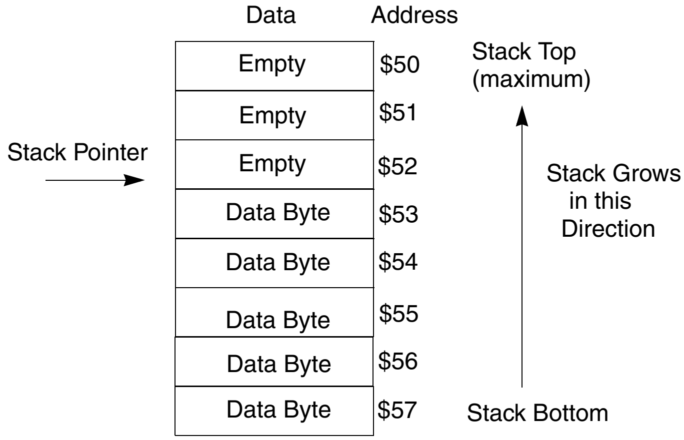

Figure 2. Stack Data Structure

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 point to the first piece of valid data or to the next available location. The code in Section 3.3, "RS08 Stack Applications," uses the latter option; it points to the next available location.

3.1 Stack Reading and Writing

A stack-read operation is called pulling, and a stack write operation is pushing. When you pull data from the stack, the data is removed and the stack pointer adjusts. When you push data onto the stack, data adds to the stack, and the stack pointer adjusts.

In the implementation of Figure 2, a push operation first stores the data to the address pointed to by the stack pointer and then decrement the stack pointer. A pull operation retrieves the data the stack pointer points to and then increments the stack pointer.

Two error conditions are intrinsic to this data structure: underflow and overflow. A stack underflow occurs when you attempt to pull information off an empty stack. A stack overflow occurs when you attempt to push information onto a full stack. When using this data structure, these conditions should be attended to. An underflow condition should return an error. On an overflow, you can 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.

3.2 MCU Hardware Stack

MCUs use a stack structure for saving program content before transferring program control. This interaction may be the result of a jump or interrupt. In the event of an interrupt, the stack pushes the values in the X (index register), A (accumulator), and CCR (condition code register) registers, as well as the PC (program counter) value. When encountering a jump instruction, the PC value is pushed onto the stack. On returning from an interrupt (RTI instruction), the program registers and PC are pulled from the stack. When returning from a jump (RTS instruction), the PC is pulled from the stack.

3.2.1 RS08 Stack

The RS08 family of MCUs have no stack-pointer registers in the core and, therefore, no automatic program control. Section 7, "Linked Lists," shows a macro managing the use of the shadow program counter (SPC) for nested subroutines. The rest of this chapter described a generic stack application adaptable for any application need.

3.3 RS08 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:

```
; 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 only used in specific subroutines, this method is not most efficient. These variables' RAM space can be dynamically allocated by using a software stack or MCU stack, freeing up RAM memory. The same method can apply to subroutine input and output parameters, passing them on the stack instead of in the A or X register.

The following code shows a software implementation of a stack appropriate for RS08 family of MCUs.

Software stack:

```
;***********************************************************************
;* 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                    *
;***********************************************************************
;Stack Equates
StackTop: equ $00000048
StackBottom: equ $0000004F

;
; variable/data section
;
                        ORG RAMStart
StackPointer            DC.B 1                            ;Pointer to next stack byte
temp                    DC.B 1                            ;Temporary storage location
```

```
                                                ; code section
                        ORG ROMStart
_Startup:
mainLoop:
Init                    LDA #StackBottom        ;Initialize Stack Pointer
                        STA StackPointer
                        feed_watchdog
                        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 Full Stack
                        BCS FullErr
Read                    JSR PullA               ;Read from Stack
                        BCS EmptyErr
                        JSR PullA               ;Read from Stack
                        BCS EmptyErr
                        JSR PullA               ;Read from Stack
                        BCS EmptyErr
Loop                    BRA Init                ;your code here
EmptyErr                DEC StackPointer        ;your code here
                        BRA Loop
FullErr                 INC StackPointer        ;your code here
                        BRA Read

;************************************************************************
;* Push Subroutine                                                      *
;* Push the contents of the accumulator onto stack                      *
;* Use C bit of CCR to indicate full error                              *
;************************************************************************
PushA                   STA temp                ;place A in temporary storage
                        LDA StackPointer        ;Get Stack Pointer
                        CMP #StackTop           ;Check for full stack
                        BLO Full
                        LDX StackPointer
                        LDA temp                ;get A from temporary storage
                        STA $0E                 ;and save in stack
                        DEC StackPointer        ;Decrement Stack Pointer
                        CLC
```

```
                        RTS
Full                    LDA temp                          ;get A from temporary storage
                        SEC                               ;Set Carry Bit for error
                        RTS

;***********************************************************************
;* Pull Subroutine                                                     *
;* Pull the contents off the stack into accumulator                    *
;* Use C bit of CCR to indicate empty error                            *
;***********************************************************************
PullA                   LDA StackPointer                 ;Get Stack Pointer
                        CMP #StackBottom                 ;Check for empty stack
                        BEQ Empty
                        LDX StackPointer
                        INCX                             ;Increment Stack Pointer
                        LDA ,X                           ;Get Data off stack
                        STX StackPointer                 ;Record New Stack Pointer
                        CLC                              ;Clear Carry Bit
                        RTS
Empty                   SEC                              ;Set Carry Bit for error
                        RTS
```

Using the software stack, a subroutine can allocate variables by pushing (allocating) bytes on the stack, accessing them with X (tiny address $0F) and D[X] (tiny address $0E), and pulling them (deallocating) before returning. In this way, multiple subroutines can use the same RAM space.

Parameters can also be passed to and from subroutines. 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.

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

4 Queues

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

Figure 3 illustrates a queue.

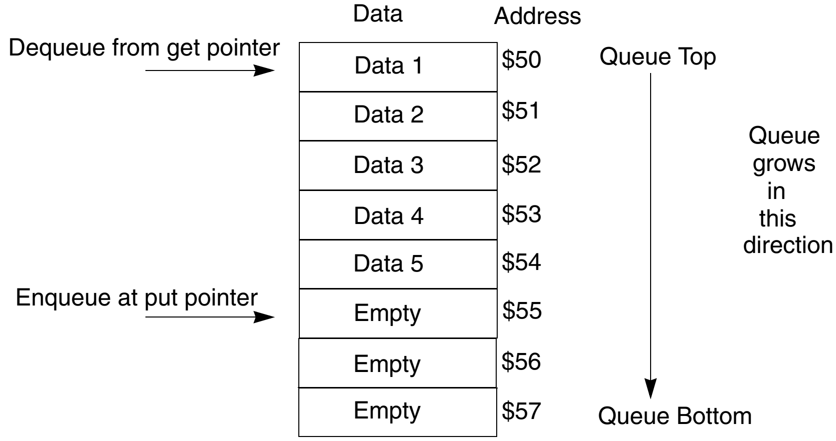

Figure 3. Queue

4.1 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 the put pointer points to, and the put pointer adjusts. For a dequeue operation, the data is read from the get-pointer location, and the pointer adjusts.

Queues usually have a fixed size, so 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.

4.2 Queue Errors

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

4.3 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.

An effective application for this is storing data received from the serial input/output port for processing later.

Queue software example:

```
;*********************************************************************
;*Illustrates an example of a queue for RS08                         *
;*********************************************************************
;*********************************************************************
;*variable/data section                                              *
;*********************************************************************
                        ORG RAMStart                  ;Insert your data definition here
```

```
TempA                   DC.B 1                        ;Temporary Accumulator
TempX                   DC.B 1                        ;Temporary X register

GetPointer              DC.B 1
PutPointer              DC.B 1
QCount                  DC.B 1
QMax                    DC.B 1

QueueTop:               equ $44
QueueBottom:            equ $47

;**********************************************************************
;*Program Code                                                        *
;**********************************************************************
                        ORG ROMStart

_Startup:
mainLoop:               LDA #QueueBottom             ;calculate maximum Queue size
                        SUB #QueueTop
                        INCA
                        STA QMax

InitQ                   LDA #QueueTop                ;Initialize Q pointer and
                                                                                    ;
variables
                        STA GetPointer
                        STA PutPointer
                        CLR QCount

;**********************************************************************
;* Write and Read from the Queue                                      *
;* A good application of this is to place bytes received from         *
;* the SCI into the queue and retrieve them later                     *
;* This code does not deal with the error conditions                  *
;**********************************************************************
                        JSR Dequeue                  ;Will return Empty error
                        feed_watchdog
                        LDA #$FF
                        JSR Enqueue                  ;Will load FF in to $44
                        JSR Enqueue                  ;Will load FF in to $45
                        JSR Enqueue                  ;Will load FF in to $46
                        JSR Enqueue                  ;Will load FF in to $47 and
                                                     ;wraps back to $44
                        JSR Enqueue                  ;Will return a Full error as
                                                     ;QCount is 4
                        JSR Dequeue                  ;Will Pull FF from $44
                        JSR Dequeue                  ;Will Pull FF from $45
                        feed_watchdog
                        LDA #$55
                        JSR Enqueue                  ;Will load 55 in to $44
                        JSR Enqueue                  ;Will load 55 in to $45
                        BRA mainLoop
```

```
;*********************************************************************
;* Subroutines                                                       *
;*********************************************************************
;*********************************************************************
;* Enqueue - enqueues a data byte passed in accumulator
*
;* Checks for a full queue and returns a set carry bit if
*
;* full otherwise returns a cleared carry bit if successful
*
;*********************************************************************
Enqueue                 STX TempX                   ;Save X register contents
                        STA TempA                   ;Save accumulator contents
                        LDA QCount                  ;Check for a full Q
                        CMP QMax
                        BEQ QFull
                        LDA TempA                   ;If Queue has space restore A
                        LDX PutPointer
                        STA $0E                     ;Place A in the queue
                        LDA PutPointer
                        CMP #QueueBottom
                        BEQ WrapPut
                        INC PutPointer              ;Increment Pointer if not

                                                    ;wrapping
                        BRA EnQDone

WrapPut                 LDA #QueueTop               ;If OK move pointer back to
                                                    ;Top of Queue
                        STA PutPointer

EnQDone                 LDX TempX                   ;Restore X register
                        LDA TempA                   ;Restore accumulator contents
                        INC QCount                  ;Increment Q Counter
                        CLC                         ;Clear Carry Bit
                        RTS

QFull                   LDX TempX                   ;Restore X register
                        LDA TempA                   ;Restore accumulator contents
                        SEC                         ;Set Carry Bit
                        RTS

;*********************************************************************
;* Dequeue - dequeues a data byte from queue and return in A         *
;* If Queue is empty returns a carry set to indicate error           *
;* otherwise returns a cleared carry bit and data in A               *
;*********************************************************************
Dequeue                 STX TempX                    ;Save X register contents
                        LDA QCount                   ;Check for an empty Q
                        CMP #$00
                        BEQ QEmpty
```

```
                LDX GetPointer          ;If Queue has population
                LDA $0E                 ;get item from Queue
                STA TempA
                LDA GetPointer
                CMP #QueueBottom
                BEQ WrapGet
                INC GetPointer          ;Increment Pointer
                BRA DeQDone

WrapGet         LDA #QueueTop           ;If OK move pointer back to
                                        ;Top of Queue
                STA GetPointer

DeQDone         LDX TempX               ;Restore X register
                LDA TempA
                DEC QCount              ;Decrement Q Counter
                CLC                     ;Clear Carry Bit
                RTS
QEmpty          LDX TempX               ;Restore X register
                SEC                     ;Set Carry Bit
                RTS
```

5 Multiple Access Circular Queue (MACQ)

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 contains the most recent entries. It is useful for data-flow problems, when only the latest data is of interest. Once initialized, it is full, and a write operation discards the oldest data.

Figure 4 depicts a MACQ.

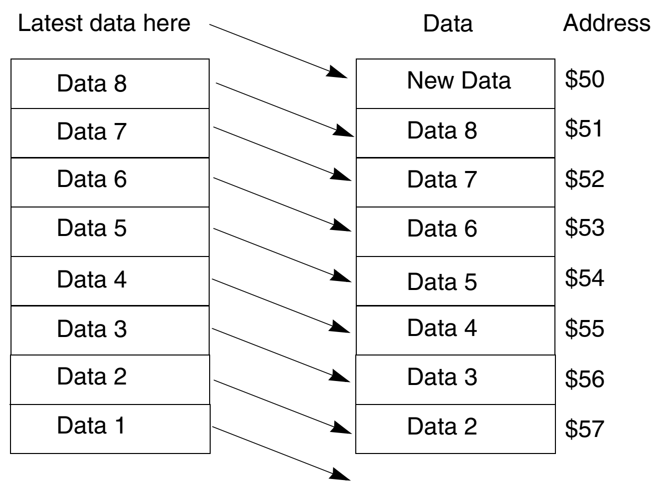

Figure 4. Result of a MACQ Write

5.1 Applications

A MACQ is useful for data streams requiring the latest data and can afford to have a destructive write operation. For example, a weather 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; they can calculate running averages, etc.

5.2 Example

MACQ illustrates the implementation of a circular buffer. This could store A/D converter readings. In this way, the latest A/D conversion results are accessible through the circular buffer.

MACQ:

```
;*********************************************************************
;*Illustrates an example of a MACQ for RS08                          *
;*********************************************************************
;*********************************************************************
;*variable/data section                                              *
;*********************************************************************
                        ORG RAMStart                  ;Insert your data definition here

TempA                   DC.B 1                        ;Temporary Accumulator
TempX                   DC.B 1                        ;Temporary X register
TempData                DC.B 1                        ;Temporary data storage

QPointer                DC.B 1
QSize                   DC.B 1

QueueTop:               equ $40
QueueBottom:   equ $47
;*********************************************************************
;*Program Code                                                       *
;*********************************************************************
                        ORG ROMStart
_Startup:
mainLoop:              LDA #QueueBottom              ;calculate maximum Queue size
                       SUB #QueueTop
                       INCA
                       STA QSize

InitQ                  LDA #QueueBottom              ;Initialize Q pointer
                       STA QPointer

;*********************************************************************
;* Write and Read from the MACQ                                       *
;* A good application of this is to store ACMP Readings, so          *
;* the latest readings are always available                          *
;*********************************************************************
                        LDA #$55
```

```
                    JSR WriteQ              ;Writes 55 to $47
                    LDA #$56
                    JSR WriteQ              ;Writes 56 to $46
                    LDA #$57
                    JSR WriteQ              ;Writes 57 to $45
                    LDA #$58
                    JSR WriteQ              ;Writes 58 to $44
                    LDA #$59
                    JSR WriteQ              ;Writes 59 to $43
                    LDA #$5A
                    JSR WriteQ              ;Writes 5A to $42
                    LDA #$5B
                    JSR WriteQ              ;Writes 5B to $41
                    LDA #$5C
                    JSR WriteQ              ;Writes 5C to $40
                    feed_watchdog
                    JSR WriteQ              ;Queue is full on this write
                                            ;Shifts all entries down one
                                            ;Writes 5C to $40
                    LDA #$00
                    JSR ReadQ               ;Read newest item
                    LDA #$01
                    JSR ReadQ               ;Reads 2nd newest item
                    LDA #$02
                    JSR ReadQ               ;Reads 3rd newest item
                    feed_watchdog
                    BRA mainLoop

;***********************************************************************
;* Subroutines                                                         *
;***********************************************************************
;***********************************************************************
;* WriteQ - A contains data to be written. Write is                    *
;* destructive on a full Q, once initialized Q is always full          *
;***********************************************************************
WriteQ              STX TempX                 ;Save X register contents
                    STA TempA                 ;Save A contents
                    LDA QPointer              ;Load Q Pointer
                    CMP #QueueTop-1           ;See if Queue is full
                    BEQ QFull
                    LDX QPointer
                    LDA TempA
                    STA $0E                   ;Store data to the Queue
                    DEC QPointer              ;Decrement Pointer
                    BRA QDone

;Once queue is initialized, it is always full
QFull               LDA TempA
                    STA TempData
                    LDX #QueueBottom-1         ;Start shifting data down

SwapLoop            LDA $0E                    ;Get 1st item to shift - 2nd
```

```
                                                            ;last one
                        INC X
                        STA $0E                             ;Store in next queue space
                                                            ;overwritting last item
                        DEC X
                        DEC X
                        TXA
                        CMP #QueueTop                      ;Check to see whether any
                                                           ;more item to shift
                        BHS SwapLoop
                        LDX #QueueTop
                        LDA TempData
                        STA $0E                            ;Place new item at top of
                                                           ;queue
QDone                   LDX TempX
                        LDA TempA
                        RTS

;***********************************************************************
;* ReadQ - A contains queue index location to be read.                 *
;* Returns value in A                                                  *
;***********************************************************************
ReadQ                   STX TempX                          ;Save X register contents
                        STA TempA                          ;Save A contents
                        ADD #QueueTop                      ;Add QueueTop to A
                        TAX                                ;X is adress of desired value
                        LDA $0E
                        RTS
```

6 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.

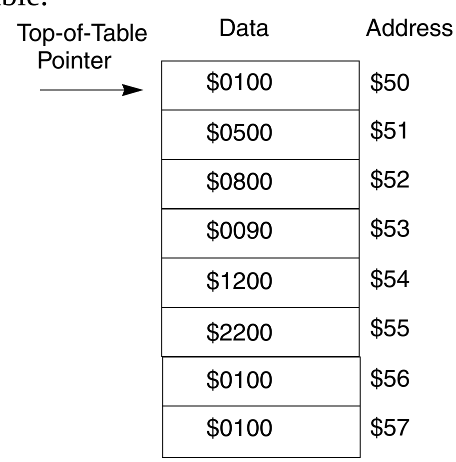

Figure 5. Table Representation

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.

6.1 Table Applications

The table data structure is common in MCU applications. One way to use tables is by performing character conversions. For LCDs (liquid crystal displays), an ASCII character byte may need to be converted to segment bitmaps for the display. A table could be used for this.

Another table application is a jump table. This is a table of vector values that are addresses to be loaded and vectored to. Some program parameters 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, Freescale MCUs have a built-in vector table used for interrupt and exception processing. These vector tables allow pre-programmed addresses to be defined for certain MCU exceptions. When an exception occurs, a new program-counter value is fetched from the appropriate table entry.

You can also use the table data structure by storing predefined values for lookup. (for example, storing interpolation data in a table performing mathematical functions). This use of a table is documented in the application note, “Integer Math routines for RS08,” Freescale document order number, AN3348.

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,” Freescale 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.

6.2 Table Example

An example of the use of tables to convert ASCII data to LCD segment values:

```
;*********************************************************************
;*variable/data section                                              *
;*********************************************************************
                        ORG RAMStart                   ;Insert your data definition here
LCD1                    DC.B 1
LCD2                    DC.B 1

;*********************************************************************
;*Program Code                                                       *
;*********************************************************************
                        ORG ROMStart
_Startup:
mainLoop:               LDA #73                         ;Load an ASCII character - I
                        JSR Convert                     ;Convert the character into a
                                                        ;table offset
                        MOV #$E1,PAGESEL                ;Change memory page to access
                                                        ;Table
```

```
                        ADD   #$C0

;alternative code for "Change memory page to access Table"
                        ;MOV #HIGH_6_13(Table),PAGESEL
                        ;STA MAP_ADDR_6(Table)

                        TAX                                  ;Transfer offset in to X
                        LDA   $0E                             ;Load the first byte
                        STA   LCD1                            ;Store in data register
                        INCX
                        LDA   $0E                             ;Load the second byte
                        STA   LCD2                            ;Store in data register
                        BRA   mainLoop

;*********************************************************************
;* Convert ASCII character byte in A to an offset value into          *
;* the table of LCD segment values. Valid ASCII values are            *
;* (DECIMAL): 65-90                                                   *
;*********************************************************************
Convert                 CMP #65                              ;Check for numeric
                        BLO ConvError
                        CMP #91                              ;Check for invalid values
                        BHS ConvError
                        SUB #65                              ;Convert to table offset
                        BRA ConvDone

ConvError               CLRA                                 ;Invalid value shows as blank

ConvDone                ROLA                                 ;Multiply offset by 2 as
                                                             ;2 bytes per LCD location
                        RTS


;*********************************************************************
;* LCD LookUp Table                                                   *
;* 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.                                                  *
;* ENSURE TABLE FITS WITHIN ONE PAGE                                  *
;*********************************************************************
                        ORG $3840
Table                   FDB $2764                            ;'A'
                        FDB $8785                            ;'B'
                        FDB $01E0                            ;'C'
                        FDB $8781                            ;'D'
                        FDB $21E4                            ;'E'
                        FDB $2164                            ;'F'
                        FDB $05E4                            ;'G'
                        FDB $2664                            ;'H'
                        FDB $8181                            ;'I'
                        FDB $06C0                            ;'J'
```

```
                        FDB $206A                    ;'K'
                        FDB $00E0                    ;'L'
                        FDB $1662                    ;'M'
                        FDB $1668                    ;'N'
                        FDB $07E0                    ;'O'
                        FDB $2364                    ;'P'
                        FDB $07E8                    ;'Q'
                        FDB $236C                    ;'R'
                        FDB $25A4                    ;'S'
                        FDB $8101                    ;'T'
                        FDB $06E0                    ;'U'
                        FDB $4062                    ;'V'
                        FDB $4668                    ;'W'
                        FDB $500A                    ;'X'
                        FDB $9002                    ;'Y'
                        FDB $4182                    ;'Z'
EndTable                EQU *-Table                  ;End of table label
```

7 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 containing a pointer to another list.

Figure 6 represents a linked list.

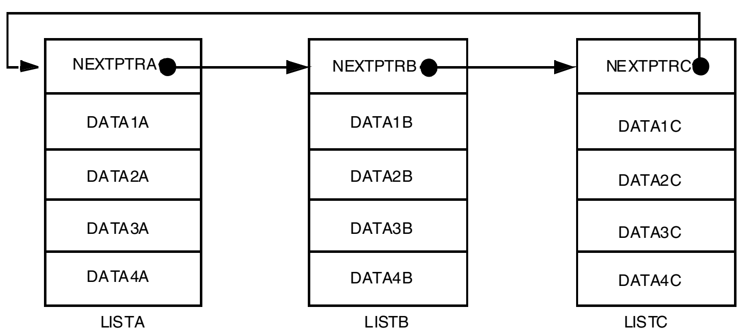

Figure 6. Linked List

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 a base address. In a doubly linked list, pointers are kept to the next and previous item in the list. A linked list can be traversed easily by simply following the pointers from one list to the next.

7.1 Linked List Applications

Traditionally, a linked list defines a dynamically allocated database, in which the elements can be ordered or resorted by adjusting the links. However, in a small MCU, there are more appropriate applications of linked lists.

A linked list can be 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.

7.2 State Machines

Another useful application of a linked list is defining a state machine. A state machine can be represented by a discrete number of states, each having an output and pointers to the next state(s). See Figure 7.

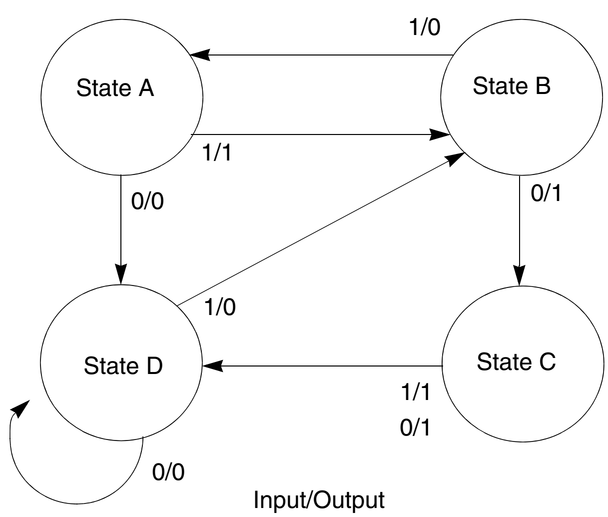

Figure 7. State Machine

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.

You can first model a process as a sequential machine, then convert this behavior to a linked-list structure and write an interpreter for it. Modify the state machine by changing the data structure (linked list) and not the code.

7.3 State Machine Example

Imagine you want to cross the street. Before you can safely cross, you must push the pedestrian-crossing controller. The controller has two light patterns: one for automobile lights and one for the pedestrian lights. To activate the pedestrian-crossing, you must press a button at the side of the road. See Figure 8.

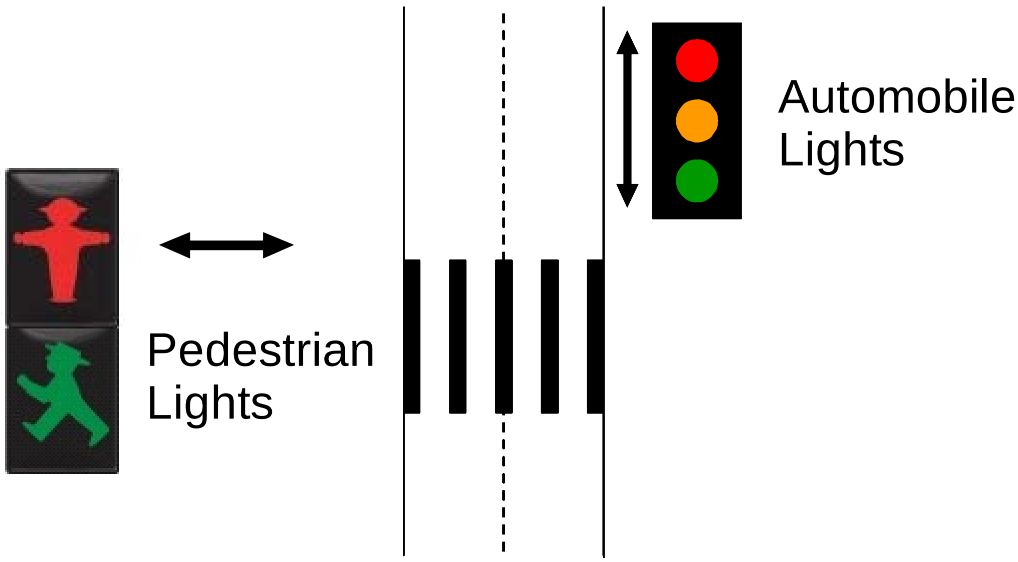

Figure 8. Pedestrian Crossing Controller Example

This is like a Moore state machine: its output is 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 is a green light on the automobile lights and a red light for the pedestrians. The controller remains in this state until a pedestrian's input. The flow continues as shown in the diagram. The output is a pattern for the light array to activate the lights for the state.

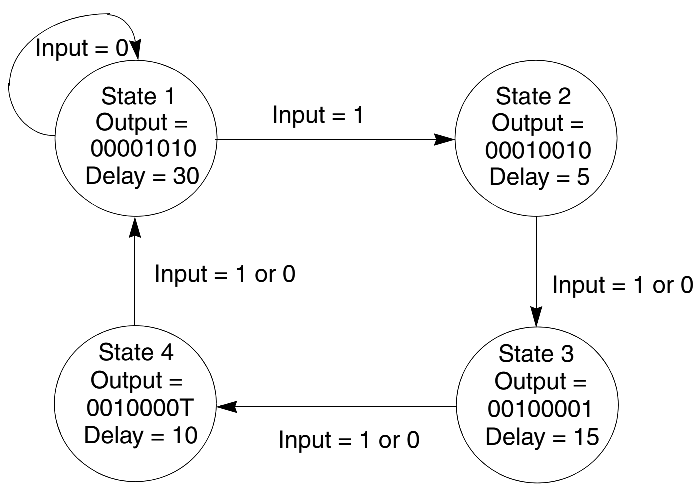

Figure 9. Pedestrian Crossing Controller State Machine

7.4 Simulation

This example can be simulated using LEDs and a MC9RS08KA2 MCU. A push-button switch can simulate the input sensor. Figure 10 illustrates the simulation circuit. Using five bits of an output port, a pattern can be generated to display the appropriate lights (LEDs). Table 1 shows the bitmap in this application.

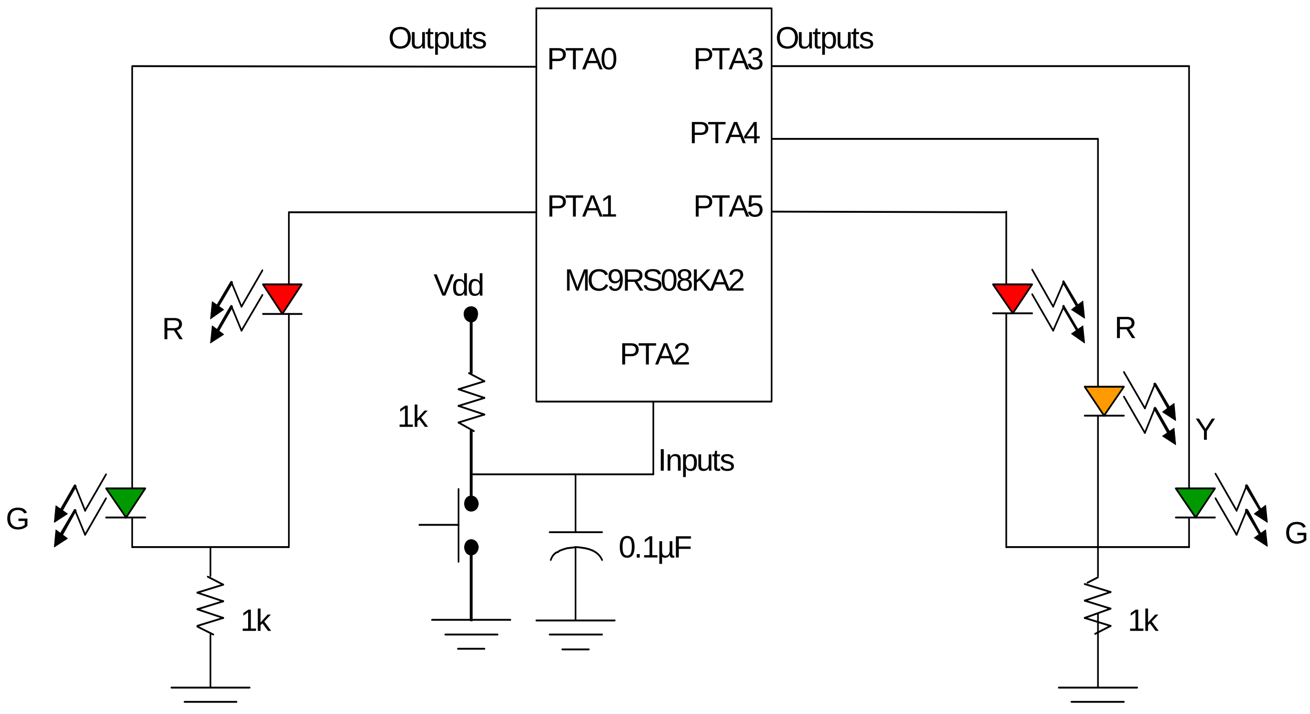

Figure 10. Circuit Simulation of Pedestrian Crossing Controller

Table 1. Pedestrian Crossing Lights Bitmap For Port A

	Car				Ped	
	R	Y	G	Button	R	G
State	PTA5	PTA4	PTA3	PTA2	PTA1	PTA0
1	0	0	1	0	1	0
2	0	1	0	1	1	0
3	1	0	0	X	0	1
4	1	0	0	X	0	Flashing

With the hardware in place, the last step is defining the state machine in software. Do this by implementing a linked-list data structure and the code to access and interpret the machine.

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

- The byte that is the bitmap for the lights.
- A delay value — the time the controller remains in the state
- The next state pointer for an input of 0

- The next state pointer for an input of 1

The program's main loop should execute the program flow charted in Figure 11.

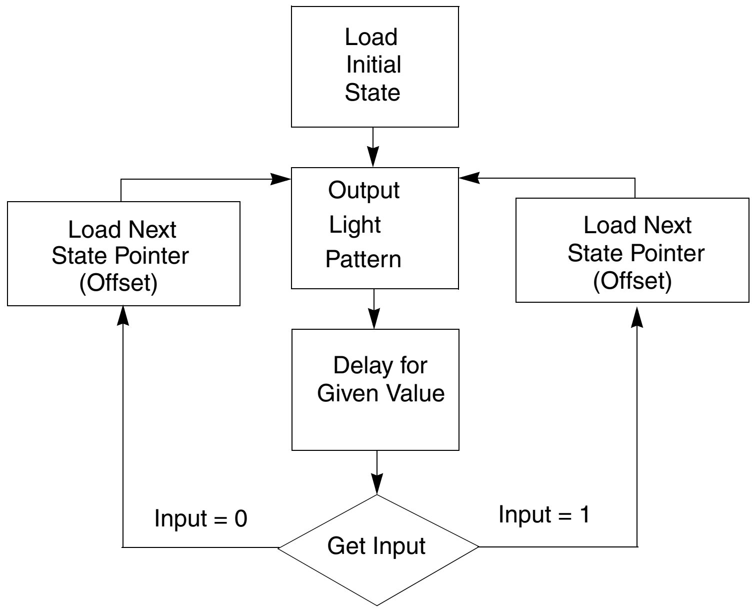

Figure 11. State Machine Program Flow

Pedestrian-crossing controller state machine:

```
;************************************************************************
;* Pedestrian Crossing Signal/Lights Controller example.               *
;* Illustrates a linked list implementation of a state machine for     *
;* the MC9RS08KA2                                                      *
;************************************************************************
;************************************************************************
; Macro to manage nested Subroutine entry code                         *
;************************************************************************
ENTRY_CODE:          MACRO
                     SHA
                     STA pcBUFFER+(2*(\1))
                     SHA
                     SLA
                     STA pcBUFFER+(2*(\1))+1
                     SLA
                     ENDM

;************************************************************************
; Macro to manage nested Subroutine exit code                          *
;************************************************************************
EXIT_CODE:           MACRO
                     SHA
                     LDA pcBUFFER+2*(\1)
                     SHA
```

```
                    SLA
                    LDA pcBUFFER+2*(\1)+1
                    SLA
                    ENDM

; Include derivative-specific definitions
                        INCLUDE 'derivative.inc'

;*******************************************************************
;*variable/data section                                            *
;*******************************************************************
                      XDEF _Startup
                      ABSENTRY _Startup

MAXlevel              EQU 1                           ;Nesting depth for subroutine
                                                      ;macro

                      ORG   RAMStart                  ;Insert your data definition here
TempA                 DC.B 1
TempX                 DC.B 1
DelayCntr             DC.B 1
pcBUFFER               DS.W MAXlevel                  ;Buffer for return address of
                                                      ;nested subroutine macro



;*******************************************************************
;*Program Code                                                     *
;*******************************************************************
                        ORG ROMStart
_Startup:
mainLoop:               MOV #$C0,ICSC2                 ;Select Bus Frequency of 1MHz
                        LDA #$00
                        STA PTAD                       ;Predefine output levels
                        LDA #$33
                        STA PTADD                      ;GPIO PTA 0, 1, 3, 4, 5 Outputs
                        MOV #$E4,PAGESEL               ;Change memory page to access
                                                       ;Table
                        LDA #STATES                    ;Index initial space
                        ADD #$C0

;alternative code for "Change memory page to access Table"
                       ;MOV #HIGH_6_13(State1),PAGESEL
                       ;LDA MAP_ADDR_6(State1)

                        TAX

Loop                    LDA $0E                        ;Get Light Pattern
                        STA PTAD                       ;Output Light Pattern
                        CMP %00100000                  ;Check to see if in State 4
                        BNE LoadDelay
```

```
                         JSR ToggleWalk

LoadDelay                INCX
                         LDA $0E                        ;Get delay
                         BRA SecDelay                   ;Cause delay
NextState                MOV #$E4,PAGESEL               ;Change memory page to access
                                                        ;Table

;alternative code for "Change memory page to access Table"
                        ;MOV #HIGH_6_13(State1),PAGESEL

                         BRCLR 2,PTAD,Input0            ;Check for pedestian input

Input1                   INCX
                         INCX
                         LDA $0E
                         ADD #$C0
                         STA $0F                         ;Get next state offset
                         BRA Loop                        ;input = 1

Input0                   MOV #$E4,PAGESEL                ;Change memory page to access
                                                         ;Table
                         INCX
                         LDA $0E
                         ADD #$C0
                         STA $0F                         ;Get next state offset
                         BRA Loop ;input = 0

ToggleWalk               INCX
                         LDA $0E                         ;Get Delay

FlashLight               BSET 0,PTAD
                         JSR  Delay0                      ;Turn LED on for ~0.5 second
                         BCLR 0,PTAD
                         JSR  Delay0                      ;Turn LED off for ~0.5 second
                         DECA
                         CMP  #00
                         BEQ  Input0                      ;Branch to "input 0" routine
                                                          ;if 10 seconds have passed
                         BRA  FlashLight                  ;Else repeat flash

;*********************************************************************
;* Delay subroutines                                                 *
;*********************************************************************
;* Cause a delay of approx (1 second * Accumulator value) @ fop = 1M   *
;* Delay value passed in through A                                   *
;*********************************************************************
SecDelay:                feed_watchdog
                         CMP #$00
                         BEQ SecDone
                         JSR Delay0
                         JSR Delay0                       ;1 sec delay (2 x 0.5 sec)
```

```
                        DECA
                        BRA SecDelay
SecDone                 BRA NextState

;***********************************************************************
;* Cause a delay of ~1/2 of a second                                   *
;***********************************************************************
Delay0: ENTRY_CODE 0
                        feed_watchdog
                        STA TempA
                        LDA #$B2

DLoop0                  CMP #$00
                        BEQ DDone0
                        JSR Delay1
                        DECA
                        BRA DLoop0

DDone0                  LDA TempA
                        EXIT_CODE 0
                        RTS

;***********************************************************************
;* Cause about 2.8msec delay @ fop of 1MHz                             *
;***********************************************************************
Delay1: ENTRY_CODE 1
                        feed_watchdog
                        STA DelayCntr
                        LDA #$FF

DLoop1                  CMP #$00
                        BEQ DDone1
                        DECA
                        BRA DLoop1

DDone1                  LDA DelayCntr
                        EXIT_CODE 1
                        RTS
;***********************************************************************
;* DataStructure for state machine linked list                         *
;* Offsets and base address scheme is adequate for small               *
;* table (<255 bytes)                                                  *
;***********************************************************************
                        ORG $3900
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

;* Cars Green, Pedestrians Red
State1                  EQU *-STATES                  ;Offset into STATES
```

```
                        FCB %00001010                   ;Output for state
                        FCB 30                          ;Delay for state
                        FCB State1                      ;Next state for input of 0
                        FCB State2                      ;Next state for input of 1

;* Cars Yellow, Pedestrians Red
State2                  EQU *-STATES
                        FCB %00010010
                        FCB 5
                        FCB State3
                        FCB State3

;* Cars Red, Pedestrians Green
State3                  EQU *-STATES
                        FCB %00100001
                        FCB 15
                        FCB State4
                        FCB State4

;* Cars Red, Pedestrians Flashing Green
State4                  EQU *-STATES
                        FCB %00100000                    ;Green initially off when state
                                                         ;entered
                        FCB 10
                        FCB State1
                        FCB State1
```

8 Summary

The use of data structures is not limited to large, complicated computers. Although the data structure is a powerful concept in such a context, the same principles apply to smaller processors such as 8-bit microcontrollers.

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

Data structure concepts can improve the static and dynamic performance of an MCU application without affecting its portability or legibility.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3334
Rev. 0
11/2006

freescale™ semiconductor