MC68328 Microprocessor Application: FLEX™ Alphanumeric Chip MC68175 Interface for One-Way Pager

by

Perry Vo

dsp

FLEX Alphanumeric Chip, FLEX One-Way Stack, and Dragonball are trademarks of Motorola, Inc. **dsp**

© MOTOROLA INC., 1998

Order this document by: APR34/D

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EXAMPLES

SECTION   1

INTRODUCTION

1.1   INTRODUCTION

Combined with the Dragonball™ MC68328 microprocessor, FLEX™ One-way Stack software and the FLEX Alphanumeric Chip Integrated Circuit (IC) MC68175 provide a powerful solution for today's and tomorrow's personal portable communication devices.

1.2   SCOPE

This application report describes the hardware and software interfaces between the MC68328 (Dragonball) Microprocessor and the MC68175 (FLEX Alphanumeric Chip) IC.

1.3   MC68328

The Motorola MC68328 (Dragonball) is a low-cost, low-power, highly integrated microprocessor designed for consumer portable devices, such as PDAs, pagers, and cellular phones. The Dragonball provides key features that are suitable for many portable applications. Modules like the Real-Time Clock (RTC), LCD controller, pulse width modulator, timers, master and slave Serial Peripheral Interface (SPI), Universal Asynchronous Receiver Transmitter (UART) with infrared communications capability, and the System Integration Module (SIM28) give product engineers the flexibility and resources to design efficient and innovative products.

1.4   FLEX PROTOCOL

FLEX Protocol is the multispeed, high-performance paging protocol from Motorola that is rapidly becoming the de facto paging standard, used by 70% of the world's paging service providers. The FLEX Protocol increases paging capability up to 10 times over POCSAG, the previous paging protocol. Its synchronous communication capability with the transmitter also enhances pager battery longevity.

1.5   MC68175

The FLEX one-way paging protocol is implemented in the form of the Motorola MC68175 IC. This FLEX Alphanumeric Chip signal decoder enables developers to easily incorporate wireless paging capabilities in a wide range of consumer products. It simplifies FLEX Protocol implementation in end-user products by interfacing with several off-the-shelf paging receivers and many off-the-shelf host microcontrollers/microprocessors.

The FLEX Alphanumeric Chip MC68175 has the following primary functions:

- to process information received from a FLEX radio paging channel,

- to demodulate the audio signal,

- to select messages addressed to the paging device, and

- to communicate the message to the host MPU.

1.6   FLEX ONE-WAY STACK

Motorola FLEX One-Way Stack software runs on the host MPU and performs the following functions:

- to initialize the FLEX Alphanumeric Chip at power-up,

- to perform the tasks of interpreting the message received from this chip in an appropriate manner (numeric, alphanumeric, binary, etc.), and

- to provide the host software with the correct decoded message.

SECTION 2

HARDWARE DEVELOPMENT

2.1   FLEX DEVELOPMENT KIT

This section briefly describes the following: the FLEX one-way pager development system; the FLEX Development Kit (FDK), including the FLEX Development Board (FDB); the receiver; and the MC68328 (Dragonball) Applications Development Board (ADB).

2.1.1   FLEX Alphanumeric Chip Development Board

The FDB is a decoder module containing the FLEX signal processing decoder IC combined with a 2-bit floating audio-to-digital converter that is used to decode 4-level audio signal inputs from the receiver.

The FLEX Alphanumeric Chip IC has eight receiver control lines used for warming up and shutting down a receiver in stages. The FLEX Alphanumeric Chip also has the ability to detect a low battery signal during the receiver control sequences. It interfaces to a host MPU through a standard Serial Peripheral Interface (SPI) and has a 38.4 kHz clock output capable of driving other devices. Its minute timer offers low-power support for time-of-day function to the host.

Figure 2-1 illustrates how the FDB interfaces with the receiver and the host microprocessor. This figure can also be used as a reference design for one-way pagers, or any other consumer device with paging communication capability.
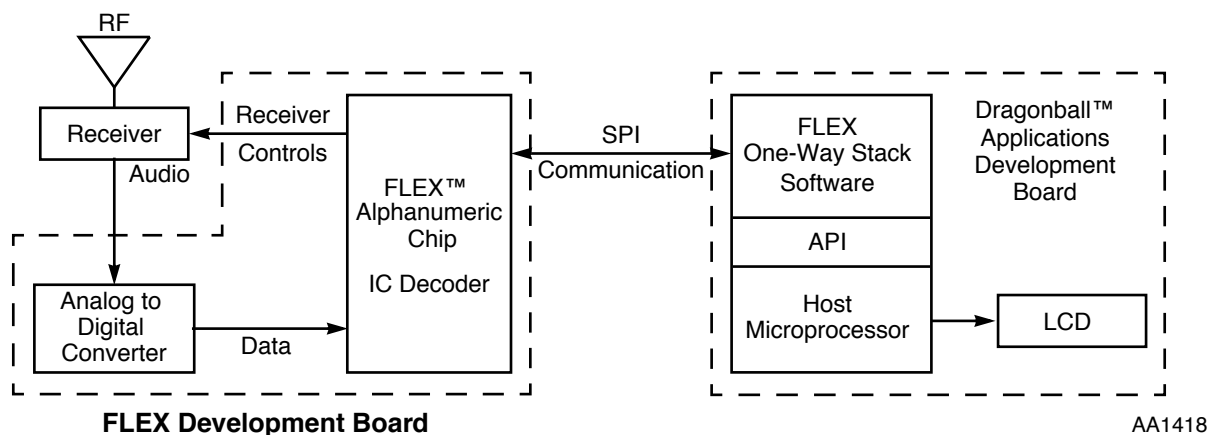
Figure 2-1 FLEX System Components

Motorola Semiconductor Products Sector has a complete FLEX one-way pager solution, as described in **Figure 2-1**, including the FLEX Alphanumeric Chip MC68175, receivers, the MC68328 microprocessor, and LCD panels. For evaluating and prototype purposes, the FDB can also receive an audio signal directly injected (to the BNC connector) from different types of signal encoders, such as the Hewlett Packard 8648A RF signal generator. For over-the-air communication, the receiver end of the FDB can interface with various types of receivers. A receiver connector is available on the FDB as a "plug-in" solution.

2.1.2   FLEX Development/Receiver Board Interface

Figure 2-2 briefly describes the interface between the FLEX Development Board and the Motorola FLEX Receiver Board. Pins 2 (A1), 3 (A2), and 10 (A0) should be connected together to one of the receiver control lines for on/off cycling operation. In the figure, those pins are connected to GND so that the receiver will be turned on at all times. Pins 1, 4, 8, and 9 are not connected.
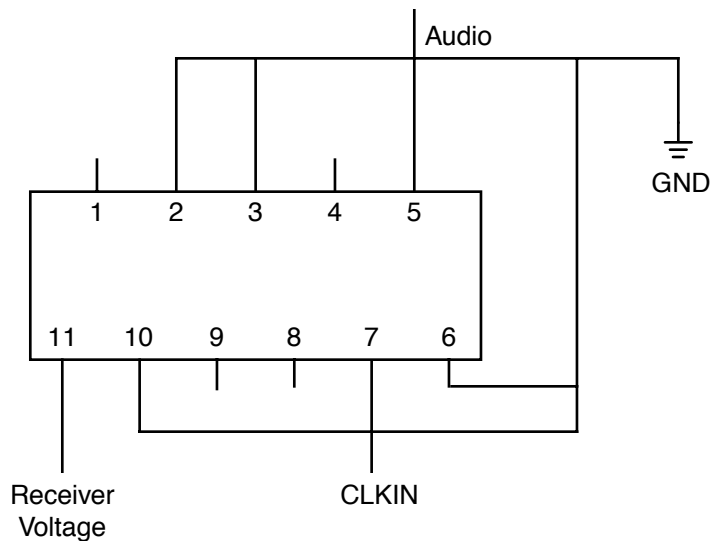
Figure 2-2 Motorola FLEX Receiver Connection Diagram

2.1.3   FLEX Development/Evaluation Board Interface

The FLEX Development Board interfaces with the MC68328 ADS evaluation board via a standard SPI. **Figure 2-3** describes the hardware interface between the FLEX Development Board and the MC68328 ADS board.
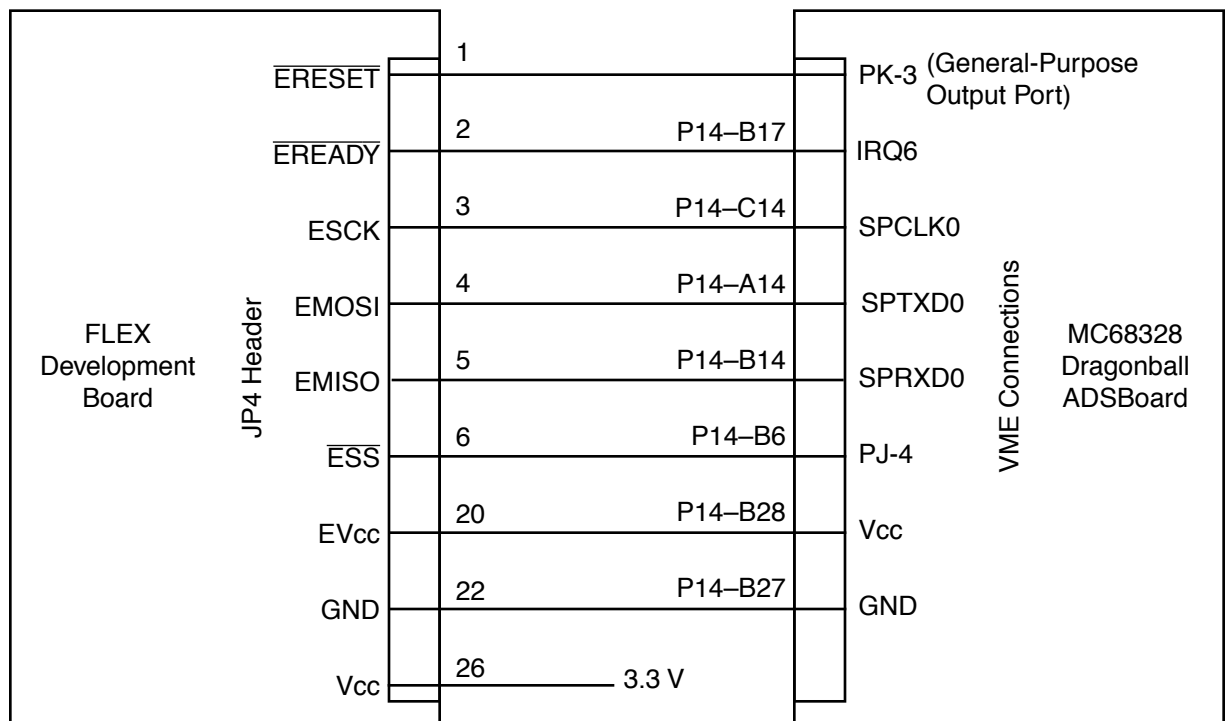
Figure 2-3 FLEX Development Board/Dragonball ADS Board SPI interface

2.1.4   SPI Interface Signals

The signals associated with the SPI shown in **Figure 2-3** are as follows:

- $\overline{\text{ERESET}}$ (JP4 Header, Pin 1) is the reset signal to the FLEX Alphanumeric Chip IC. This pin should connect to a general-purpose output port pin (e.g., Port K Pin 3), so that the FLEX One-Way Stack software running on the host can reset the FLEX Alphanumeric Chip.

- $\overline{\text{READY}}$ (JP4 Header, Pin 2) is connected to an interrupt pin on the host MPU. When the FLEX Alphanumeric Chip would like to talk to the host MPU, it will

assert the $\overline{\text{READY}}$ line low until the end of the 32-bit transfer. As an example, this pin is connected to the IRQ6 pin on the Dragonball ADS board.

- SCK (JP4 Header, Pin 3) is the clock supplied by the host MPU. This input pin is connected to SPI Clock output pin on the host ADS.

- $\overline{\text{SS}}$ (JP4 Header, Pin 6) is used as FLEX chip select. Before every data transfer between FLEX Alphanumeric Chip and the host MPU, the host MPU needs to assert $\overline{\text{SS}}$ low to select the FLEX Alphanumeric Chip IC. This pin is connected to a configured general purpose port on the host microprocessor (e.g., Port J Pin 4).

- MISO (JP4 Header, Pin 5) is the line on which data from the FLEX Alphanumeric Chip is transferred to the host.

- MOSI (JP4 Header, Pin 4) is the line on which data from the host is transferred to the FLEX Alphanumeric Chip IC.

- EVCC (JP4 Header, Pin 20) and GND (Emulator Pin 22) lines are the power supply from the host MPU (+ 5 V).

- Vcc (JP4 Header, Pin 26) is the 3.3 V power supply. This is the voltage supply for the FLEX Alphanumeric Chip and the 2-bit Floating Audio-to-Digital converter chip.

2.1.5   Additional Information

For a more complete description of the FLEX Development Board and the Dragonball MC68328 ADS board, please refer to the corresponding User's Manuals.

Contact your local Motorola sales office for more information on the FLEX one-way pager solution.

The example code presented in this application report is available via the Motorola website, reached at the following address:

```
http://www.motorola-dsp.com/documentation/appnotes
```

SECTION 3

SOFTWARE DEVELOPMENT

3.1 FLEX SYSTEM SOFTWARE

FLEX System Software (FSS) from Motorola is a family of interoperable software components used for building products with paging/messaging capabilities. FLEX One-Way Stack, one of the software components, is specifically designed to support the integration of the FLEX Alphanumeric Chip Integrated Circuit (IC) with many off-the-shelf microprocessors.

3.1.1 FLEX One-Way Stack Overview

The product engineer can regard FLEX One-Way Stack as the FLEX Alphanumeric Chip device driver. As shown in **Figure 3-1**, FLEX One-Way Stack runs on the product's host processor and communicates with FLEX Alphanumeric Chip IC. It fully interprets FLEX code-words (packets of information) received from the FLEX IC and returns the original paging message to the host software so the message can be displayed to the user.
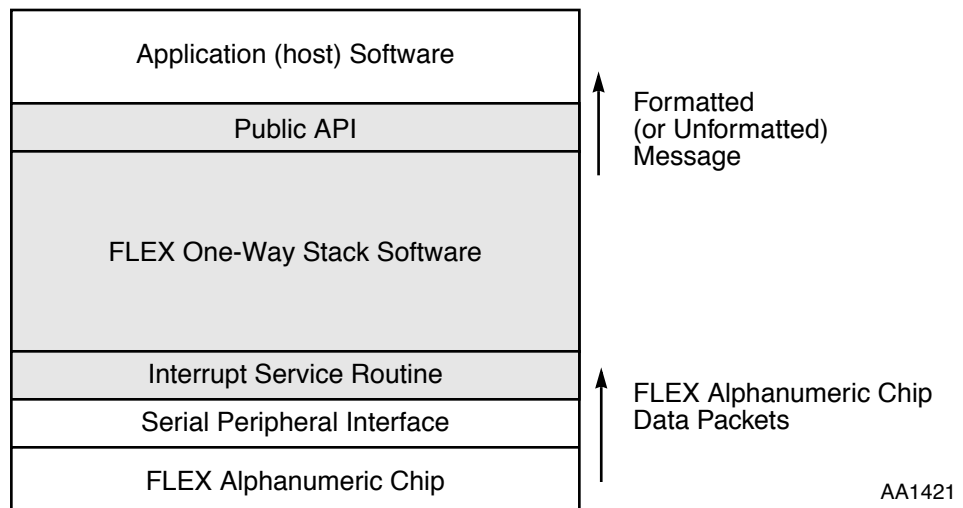
Figure 3-1 FLEX One-Way Stack Software Interfaces

A FLEX Software Development Kit (SDK) is available for downloading from Motorola's website at `http://www.motorola.com/flexstack.` The FLEX One-Way Stack software has been proven to work on various microprocessors, including the MC68328 (Dragonball). Motorola customers can download FLEX One-Way Stack software from the web and apply the recommended porting procedures described in this application report to make the software work on the MC68328 Dragonball platform.

3.1.2   SPI Communication

The FLEX Alphanumeric Chip IC communicates with the host MCU via SPI in 32-bit packets. Each packet consists of an 8-bit ID followed by twenty-four bits of data. The FLEX IC uses SPI in Full Duplex mode; that is, for every transfer, both the FLEX IC and host MCU transmit and receive valid information.

The SPI consists of a $\overline{\text{READY}}$ pin and four SPI pins: $\overline{\text{SS}}$, SCK, MOSI, and MISO, as described in the hardware section.   When the host sends a packet to the FLEX Alphanumeric Chip IC, it first selects the FLEX IC by driving the $\overline{\text{SS}}$ pin low.   When the FLEX IC has a packet for the host to read, it drives the $\overline{\text{READY}}$ line low to assert an interrupt to the host.

3.1.3   Enabling the FLEX Alphanumeric Chip

Depending on the upper 8-bit ID value, data packets can be classified and recognized by the FLEX Alphanumeric Chip and FLEX One-Way Stack software running on the host microprocessor. For example, a packet with an ID of "00" is a *Checksum* packet, while a packet with an ID of "01" is the *Configuration* packet, etc. One of the packets that the FLEX Alphanumeric Chip sends to the host MPU is the *Part ID* packet (ID = "FF"), which was designed to ensure proper communication procedure between the host and the FLEX IC.

Upon reset or at power-up, the entire FLEX Alphanumeric Chip IC is "disabled" from sending any information except the *part ID*. The FLEX Alphanumeric Chip will continue to assert interrupt and broadcast the *part ID* until the host processor, via the interrupt service routine, recognizes the FLEX IC by sending back to the FLEX Alphanumeric Chip the proper *Checksum* packet. The *Checksum* packet can be considered as the required handshaking signal from the host to recognize and "enable" the FLEX IC. The algorithm for calculating the *Checksum* value is embedded in the FLEX One-Way Stack software. Ordinarily, this initialization procedure requires two SPI data transfers, as described in **Figure 3-2**.
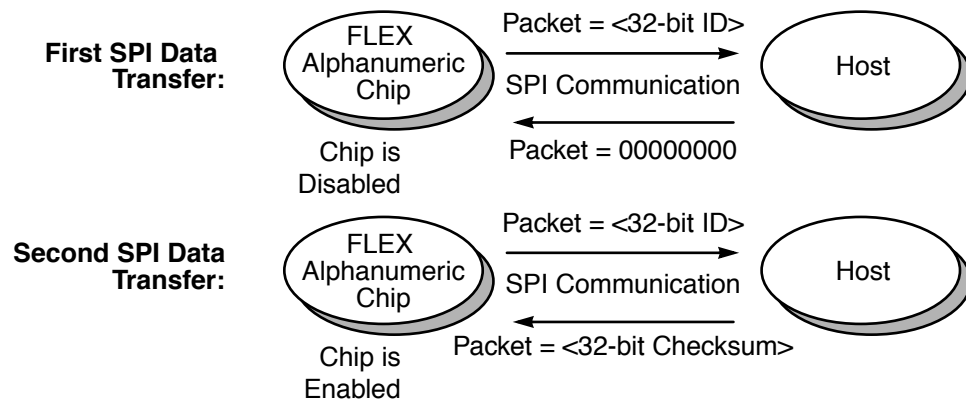
Figure 3-2 FLEX Alphanumeric Chip Enabling Steps

After the FLEX Alphanumeric Chip receives the correct *Checksum* packet from the FLEX One-Way Stack software, the chip is "enabled". It stops asserting interrupts to the host processor until it has new information to send to the host, such as a new page. The FLEX Alphanumeric Chip is "disabled" only in the sense that the host MPU cannot read data from the FLEX Alphanumeric Chip IC.
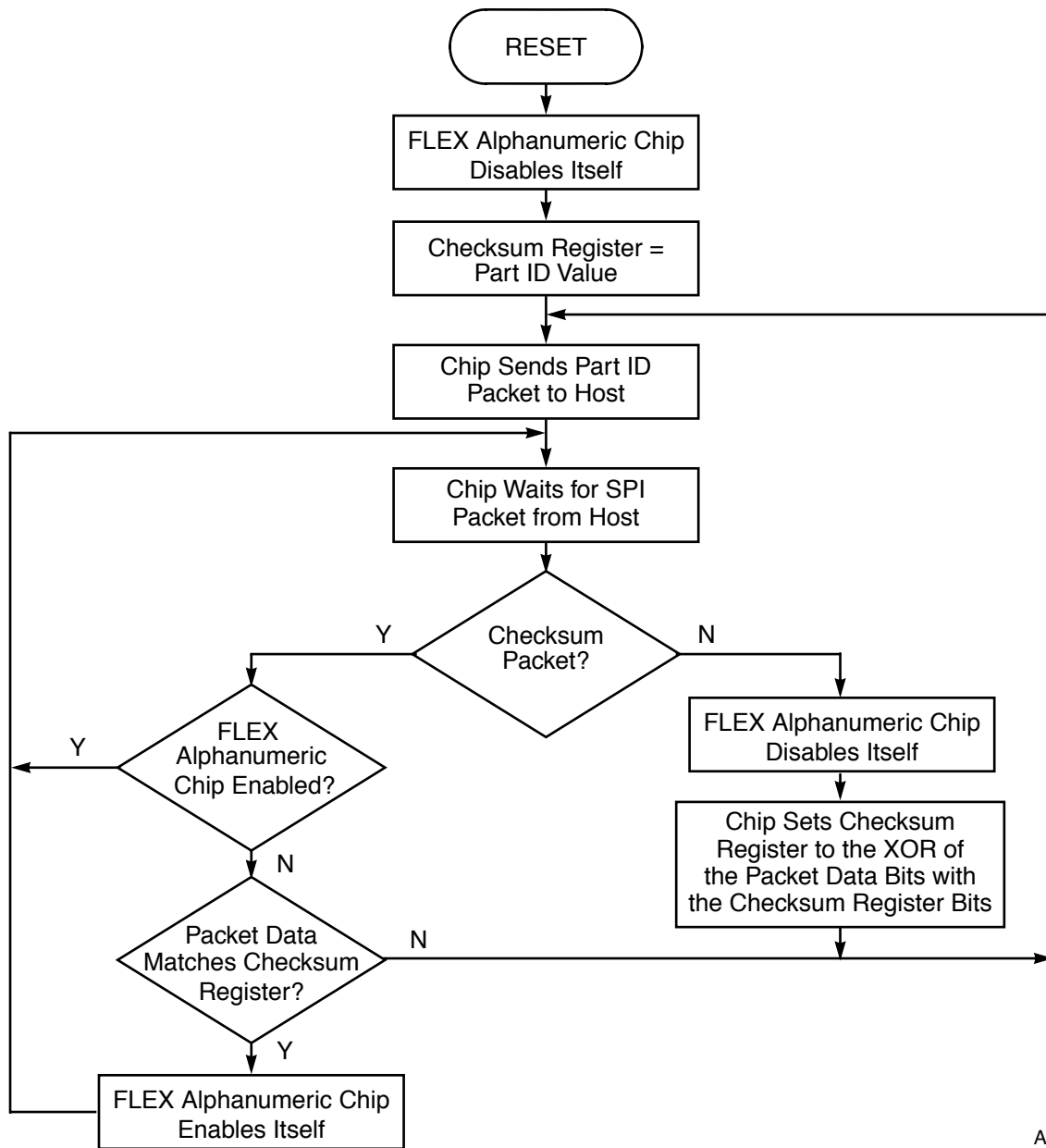
Figure 3-3 FLEX Alphanumeric Chip IC Checksum Flowchart

Note: On power-up, FLEX One-Way Stack software running on the host will attempt to receive the Part ID packet and to send back the CheckSum packet to the FLEX Alphanumeric Chip IC. If FLEX One-Way Stack does not seem to get out of the interrupt handler (i.e., the FLEX IC keeps on asserting interrupt to the host processor), it is reasonable to suspect that the FLEX IC has not received the correct (expected) Checksum packet due to a hardware problem. The

FLEX Alphanumeric Chip is still in Reset or Disable mode. In this case, the product engineer should check the configuration of the SPI communication module on the host. Common faults include the SPI module not being properly enabled or not transmitting properly, the SPI baud-rate being too high, or the reset line connected to the FLEX Alphanumeric Chip being held low.The very first packet sent from the host processor to the FLEX Alphanumeric Chip is a null packet (i.e., a packet of value 0). If this packet is a nonzero packet, the FLEX One-Way Stack may not calculate the checksum packet correctly, resulting in a locked-up situation where the chip will never come out of Reset state. Therefore, the host software may need to initialize the gSecurity variable to 0 before running FLEX One-Way Stack.

3.1.4   FLEX Alphanumeric Chip and FLEX One-Way Stack Configurations

The FLEX One-Way Stack software on the host MPU is responsible for configuring the FLEX Alphanumeric Chip IC by sending to it (via SPI) a series of configuration packets. The chip needs information such as receiver control, frame assignments, and user address enabling, so that it behaves properly for a particular paging application. FLEX One-Way Stack software performs this task automatically after the chip is enabled via the checksum feature as described above. However, product engineers need to specify how they would like to configure the chip, and translate those configurations into 32-bit packets of data that FLEX One-Way Stack will send to the FLEX IC. These configuration packets are stored in the initialization buffer.

When incorporating FLEX One-Way Stack software into FLEX paging products, it is important that product engineers properly set up the initialization buffer. The initialization buffer contains not only configuration data to be sent to FLEX Alphanumeric Chip but also memory reservations and all necessary setups so that FLEX One-Way Stack can properly manage and filter paging messages. The initialization buffer is often in EEPROM, or nonvolatile memory. The initialization data is segmented by functionality and is divided into four segments: Driver Initialization, Notification, Message Manager, Filter and FLEX Alphanumeric Chip Initialization segments. Each segment contains a header block and a data block. The header block in all segments has the same format as the INIT_SECTION structure defined in *init.h*. The data blocks for each segment are defined in *struct.h*. A sample of the function that sets up the initialization buffer is provided in file *idata.c* in FLEX One-Way Stack software.

3.1.5　Receiving and Processing Paging Messages

FLEX One-Way Stack consists of interoperable modules that work together through a set of external and intermodule APIs. The following four modules can be found in FLEX One-Way Stack:

- FLEX Driver—Directly manages the FLEX Alphanumeric Chip IC and builds raw message data from received data stream.

- FLEX Message Filter—Formats raw message data into character format, such as ASCII, binary data, etc.

- FLEX Message Manager—Stores and manages completed messages.

- FLEX Application Interface—A lightweight wrapper of APIs that exposes a high-level interface to host software.
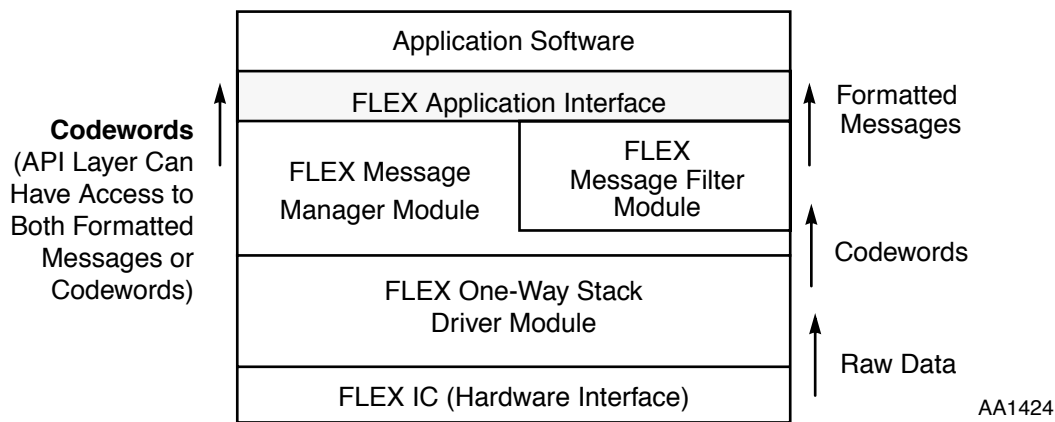
Figure 3-4　Flow of Data through FLEX One-Way Stack

The flow of message data between software layers of FLEX One-Way Stack is summarized in **Figure 3-4**. The process of receiving and processing paging messages occurs in two stages.

In the first stage, the interrupt service routine receives raw data from the FLEX IC. Every time the FLEX Alphanumeric Chip has data to send over to the host, it asserts an interrupt to the host processor. The host acknowledges the interrupt with the interrupt handler. The interrupt handler receives raw data from the FLEX Alphanumeric Chip via SPI communication and stores the data in a circular queue as described in the following paragraphs.

In the second stage, the host software periodically calls the FLEX One-Way Stack function to read data out of the circular queue. FLEX One-Way Stack then analyzes and assembles the original FLEX page and returns it to the host software.

Two pointers (index variables) are used to manage the circular queue, *gPackWrite* and *gPackRead*. The *gPackWrite* variable points to the next available byte in the queue for data storage. The pointer *gPackWrite* advances for every byte of data that the FLEX Alphanumeric Chip SPI interrupt handler receives. The *gPackRead* variable points to the next byte of data in the queue to be read and processed. The pointer *gPackRead* advances every time FLEX One-Way Stack reads a byte out of the queue for processing. Product developers can manage the size of the circular queue. However, the size of the queue (in bytes) must be a multiple of four to avoid splitting of packets (1 packet = 32 bits = 4 bytes). The circular queue is defined as shown in **Example 3-1**

Example 3-1 Circular Queue Definition

unsigned char gPacketQ[PAKQ_SIZE];
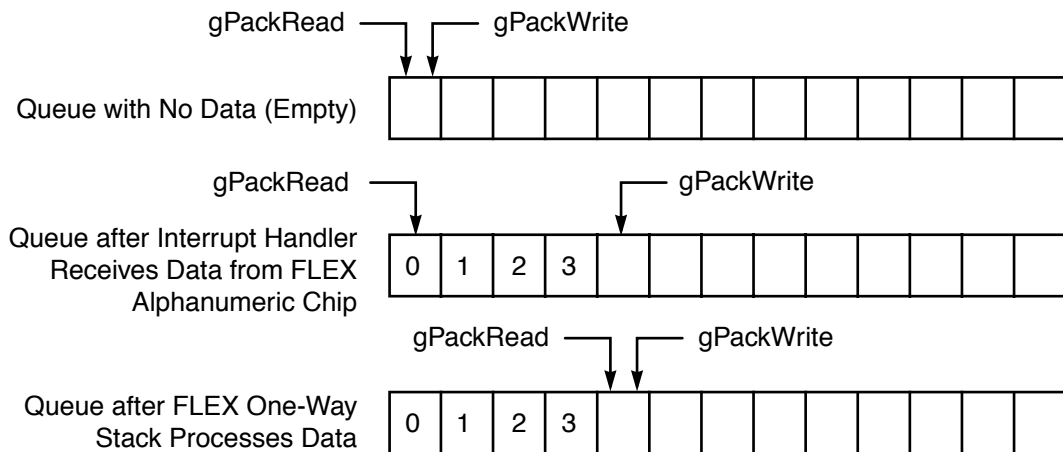
Figure 3-5 Circular Queue for Data Storage

The host software can either poll the two pointers or use a timer interrupt to check them for new data in the circular queue, shown in **Figure 3-5**. If the FLEX One-Way Stack finishes assembling the original page (meaning that no other fragment of the page is expected), FLEX One-Way Stack notifies the host software by calling the function *FStkNotifyNewMsg()*. Product engineers can implement *FStkNotifyNewMsg ()* in accordance with the individual features of each product, such as alerting the user or retrieving and displaying the message on the LCD.

Figure 3-6 shows how paging messages are received and handled by FLEX One-Way Stack.
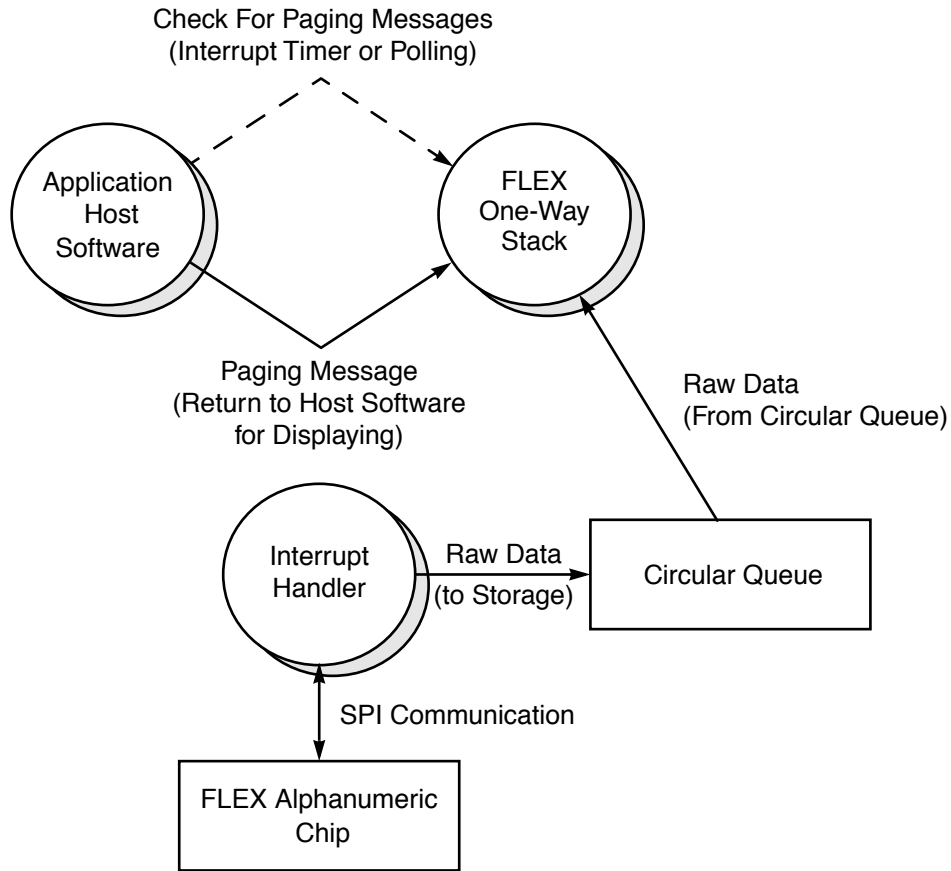
Figure 3-6 Paging Message Received/Handled by FLEX One-Way Stack

3.2   PORTING FLEX ONE-WAY STACK TO THE MC68328 MPU

The FLEX One-Way Stack software is written in the ANSI C language and is therefore highly portable. However, in order to run FLEX One-Way Stack on a specific microprocessor, some work is required. After getting the FLEX One-Way Stack software from Motorola website, product engineers may have to make a few modifications to enable the FLEX One-Way Stack software to run on the MC68328. This section describes the recommended steps to assist product engineer to port FLEX One-Way Stack software to the MC68328 (Dragonball) microprocessor.

Note: The term "FLEXchip" in code comments refers to the FLEX Alphanumeric Chip MC68175.

3.2.1 Creating the SPI Driver

The SPI interrupt service routines are hardware dependent and require the most porting effort. The recommended implementation code can be obtained from Motorola Semiconductor Products Sector. Product engineers should attempt to understand this code and reconfigure it according to their unique product specifications.

The recommended implementation code includes four functions, as shown in **Examples 3-2 through 3-6**. These functions are called *FLEX IC Handler*, *storeData*, *FlexSPITransfer*, and *waitForTransfer()*.

Example 3-2 Function *FLEX IC Handler*

```
/************************************************************************
 * FUNCTION NAME: FlexICHandler()                                      *
 * Date created:  07/26/96                                             *
 *                                                                     *
 * DESCRIPTIONS:                                                       *
 * This function is the interrupt service routine for FLEXchip         *
 * IC.  Every time the FLEXchip IC would like to communicate            *
 * with the host microprocessor, it will assert an interrupt to        *
 * the host.  This interrupt service routine is responsible for        *
 * handling all communications between the host and the FLEXchip.      *
 *                                                                     *
 * INPUT: None                                                         *
 * OUTPUT: None                                                        *
 ************************************************************************/

void FlexICHandler()
{
    volatile UVAR16 data1, data2;
    volatile UVAR8 *ptr;

    DISABLE_INT;       /* Disable further interrupts */

    /*
     * First, we assert the FLEXchip chip-select to indicate that the
     * host processor is ready to communicate with FLEXchip IC.
     */
    portj_dat = (UVAR8 *) PJDATA;
    *portj_dat &= ASSERT_SS;

    /*
     * Next, we prepare the 2 16-bit variables data1 and data2 to
     * contain the 32-bit package that the host will send to the
     * FLEXchip IC.
     */
```

Example 3-2 Function *FLEX IC Handler* (Continued)

```
    if (!(gFlag1 & INITCHIP)){        /* Normal mode */
        if (gFlag1 & CMDOUT){         /* Send 4 bytes from the gCommand */
          ptr = (UVAR8*) &data1;
          *ptr++ = gCmdBuffer.byte3;
          *ptr = gCmdBuffer.byte2;
          ptr = (UVAR8*) &data2;
          *ptr++ = gCmdBuffer.byte1;
          *ptr = gCmdBuffer.byte0;
          }
        else{                         /* Send 4 bytes from gSecurity */
          ptr = (UVAR8*) &data1;
          *ptr++ = gSecurity.byte3;
          *ptr = gSecurity.byte2;
          ptr = (UVAR8*) &data2;
          *ptr++ = gSecurity.byte1;
          *ptr = gSecurity.byte0;
        }
    }
    else{    /* This is when FlexIC is initialized the first time */
        ptr = (UVAR8*) &data1;
        *ptr++ = gInitData[gPointer++];
        *ptr = gInitData[gPointer++];
        ptr = (UVAR8*) &data2;
        *ptr++ = gInitData[gPointer++];
        *ptr = gInitData[gPointer++];
    }

    /*
     * FLEXchip and Host processor exchange data
     */
    FlexSPITransfer(&data1, &data2);

    /* Store receiving data from FLEXchip in the queue */
    storeData(data1, data2);
```

Example 3-2 Function *FLEX IC Handler* (Continued)

```
/*
      * This is when FLEXstack sends packets to configure FLEXchip IC.
      * gInitCount keeps track of the number of configuration packets.
      * Data received from FLEXchip during the configuration process
      * is ignored
      */
     if (gFlag1 & INITCHIP) {
           gInitCount--;
           if (gInitCount == 0) {
               gFlag1 &= CLEAR_INITCHIP;
               gPackWrite = gPackRead;
           }
     }
     /*
      * If this is the part ID packet, call FLEXstack function to process
      * the part ID packet to come up with the correct Checksum packet.
      */
     if (!(BTST(gFlag1,DISPART_FIRST_f1)))
           FLEXstack();

     gFlag1 &= CLEAR_CMDOUT;
     gFlag1 |= XFERDONE;

     /* At the end of the interrupt service routine, we deselect
        the FLEXchip IC */
     portj_dat = (UVAR8 *) PJDATA;
     *portj_dat |= DESELECT_SS;
}
```

Example 3-3 Function *storeData*

```
/************************************************************************
 * FUNCTION NAME: storeData()                                          *
 * Date created:  07/26/96                                             *
 *                                                                     *
 * DESCRIPTIONS:                                                       *
 * This function stores packets of information in the circular         *
 * queue for later processing.  These Packets are received             *
 * from the FLEXchip IC via the interrupt service routine              *
 *                                                                     *
 * INPUT: 32-bit of data in 2 16-bit variables                         *
 *        data1: First 16-bit data                                     *
 *        data2:Second 16-bit data                                     *
 * OUTPUT: None                                                        *
 * GLOBAL VARIABLES:                                                   *
 * gPacketQ[]:Array used as a circular queue for                       *
 *    data storage                                                     *
 * gPackWrite:index in the circular queue to keep                      *
 *    keep track of the next available storage                         *
 *    location in the queue.                                           *
 ************************************************************************/
void storeData(UVAR16 data1, UVAR16 data2)
{
    volatile UCHAR *ptr;

    /* Store the first word (16-bit data) in the queue */
    ptr = (UCHAR*) &data1;
    gPacketQ[gPackWrite++] = (UCHAR) *ptr++;
    gPacketQ[gPackWrite++] = (UCHAR) *ptr;

    /* Store the second word (16-bit data) in the queue */
    ptr = (UCHAR*) &data2;
    gPacketQ[gPackWrite++] = (UCHAR) *ptr++;
    gPacketQ[gPackWrite++] = (UCHAR) *ptr;

    /* If the index gPackWrite points to the end of the queue,
       "circulate" it to the beginning of the queue */
    gPackWrite &= (PAKQ_SIZE - 1);/* Assuming PAKQ_SIZE is 128 */
}
```

Example 3-4 Function *FlexSPITransfer*

```
/************************************************************************
 * FUNCTION NAME: FlexSPITransfer()                                     *
 * Date created:  07/26/96                                              *
 *                                                                      *
 * DESCRIPTIONS:                                                        *
 * This function performs data communication via the                    *
 * Serial Peripheral Interface (SPI).  Given the Dragonball as          *
 * the 16-bit data bus microprocessor, two transmissions are            *
 * needed, with 16 bits of data for each transmission.                  *
 *                                                                      *
 * INPUT: 32-bit of data to be sent to the FLEXchip IC                  *
 * data1: Pointer to the first 16-bit data                              *
 * data2: Pointer to the second 16-bit data                             *
 * OUTPUT: data1 and data2 contain data received from the               *
 * FLEXchip IC.                                                         *
 ************************************************************************/

void FlexSPITransfer(UVAR16 *data1, UVAR16 *data2)
{

    *spmode &= XCH_MASK;        /* Make sure no other transmission */
    *spmode |= SPMEN_BIT;       /* Enable SPI */

    /* First 16-bit data transmission */
    *spbd = *data1;
    *spmode |= XCH_BIT;
    waitForTransfer();
    *data1 = *spbd;             /* End of the first 16-bit transfer - data1
                                   contains data from the FLEXchip IC */

    /* Second 16-bit data transmission */
    *spbd = *data2;
    *spmode |= XCH_BIT;
    waitForTransfer();
    *data2 = *spbd;             /* End of the second 16-bit transfer - data2
                                   contains data from the FLEXchip IC */
    return;
}
```

Example 3-5 Function *waitForTransfer()*

```
/**********************************************************************
 * FUNCTION NAME: waitForTransfer()                                   *
 * Date created:  07/26/96                                            *
 *                                                                    *
 * DESCRIPTIONS:                                                      *
 * This function checks the SPIM_IRQ bit (in the SPI status register),*
 * after data has been moved to the SPI data register to be transmitted *
 * to the FLEXchip.  When the SPI finishes the transmission, it sets  *
 * the SPIM_IRQ bit.  By checking this bit, we can monitor the        *
 * SPI communication.                                                 *
 *                                                                    *
 * INPUT: None                                                        *
 * OUTPUT: None                                                       *
 **********************************************************************/

void waitForTransfer()
{
    while (!(*spmode & SPIM_IRQ_BIT));

    /* Reset SPIM */
    *spmode &= SPIM_IRQ_MASK;
    *spmode &= XCH_MASK;

    return;
}
```

3.2.2 Configuring PORT.H

The PORT.H file contains most of the options and configurable items, depending on the processor type. In PORT.H, product engineers should add the definitions shown in **Example 3-6** when porting FLEX One-Way Stack software to the Dragonball processor:

Example 3-6 PORT.H Definitions

```
#define DRAGONBALL

#ifdef DRAGONBALL

typedef UVAR8 *ADDRESS;             /* UVAR8 is an unsigned 8-bit value */
typedef UVAR8 *HANDLE;
typedef short VAR16;                /* Type short is a 16-bit value */
typedef int VAR32;                  /* Type int is a 32-bit value */
typedef unsigned short UVAR16;
typedef unsigned int UVAR32;
#define PTR_SIZE 4                  /* machine pointer size in bytes */
#define printf                      /* override printf */
#define ENABLE_INT asm(" ANDI.W   #$F8FF,SR")    /* Enable interrupts */
#define DISABLE_INT asm(" ORI.W   #$0700,SR")    /* Disable interrupts */
#define ATTRIB_BUF 0                        /* No temp attrib buffer needed */

#endif    /* DRAGONBALL */
```

3.2.3 Completing PORT.C

Some routines in PORT.C are product dependent and must be completed for FLEX One-Way Stack to perform properly for a particular application. We describe some important functionalities in this file. Recommended code is also included. However, the product engineer should examine this code carefully, and use it basically as reference to design applicable products.

To complete PORT.C, three functions are called upon: *FStkNotifyNewMsg()* , *Send_4_bytes()* , and *FLEXstack()*. Code samples of the three functions are shown in **Examples 3-7** through **3-9.**

FStkNotifyNewMsg()

The *FStkNotifyNewMsg()* function is called each time FLEX One-Way Stack software receives a new and completed FLEX message from the FLEX Alphanumeric Chip IC. The message ID is passed to this function. Product engineers should implement this function according to the product specifications. For example, *FStkNotifyNewMsg()* can alert the user (with sounds or vibration) and/or retrieve the message from FLEX One-Way Stack and display it on the screen or LCD. In **Example 3-7**, the new message ID is simply stored in an array, and function *GetPage()* is called to display the message on the LCD.

Example 3-7   Function *FStkNotifyNewMsg()* Sample

```
void FStkNotifyNewMsg(MSG msg)
{
      UVAR8 i;

      /*
       * If the new message overflows the array (storage of messages),
       * pop the oldest message off the array for storage.
       */
      if (pageCount >= MSGTABLESZ)
         MMMakeRoom();

      /* Store message in the message table */
      msgtable[pageCount] = msg->msgId;

      GetPage(msg->msgId);                   /* Retrieve message for display */
      pageCount++;                           /* Increase the number of pages */
}
```

Send_4_bytes()

The *Send_4_Bytes()* function sends 4-byte packets to the FLEX Alphanumeric Chip using the SPI. FLEX One-Way Stack uses this function to initiate the configuration process by sending the FLEX Alphanumeric Chip a series of packets from the initialization buffer (described earlier). This is the communication initiated by the host, so that the host processor must first drive the FLEX Alphanumeric Chip chip-select signal low to start the communication process. The actual SPI communication happens through the interrupt service routine.

Send_4_bytes() supervises the interrupt process and will terminate after the transfer is completed.

Example 3-8 Function *Send_4_bytes()* Sample

```
void Send_4_bytes (void)
{
      BCLR(gFlag1, XFERDONE_f1);

      /* Select the FLEXchip IC to initiate the communication */
      portj_dat = (UVAR8*) PJDATA;
      *portj_dat &= ASSERT_SS;

      /* Wait until the data transfer is complete */
      while(!BTST(gFlag1, XFERDONE_f1));
}
```

FLEXstack()

The *FLEXstack* function checks the circular packet queue for new data (by checking *gPackWrite* and *gPackRead* indexes, as described in earlier section). If new data are in the queue, it calls *FStkPacketProcessing* to process one packet.

A loop should wrap around *FStkPacketProcessing* to process all new information in the circular queue.

Example 3-9 Function *FStkPacketProcessing* Sample

```
void FLEXstack()
{
     while (gPackWrite != gPackRead)
        FStkPacketProcessing(GetPacket());
}
```

3.2.4   Setting up Initialization Buffer

The host software must provide initialization data to the FLEX One-Way Stack software in a predefined format for the software for configure the IC after enabling it. The initialization buffer is set up by function *BuildInitBuffer()*, as illustrated in **Example 3-10**. This function is part of FLEX One-Way Stack software and can be found in *data.c*.

Please note that data should usually be placed directly in a reserved EPROM section for code optimization. However, to better illustrate how the buffer is set up, we define structures and hard-code the initialization data in function *BuildInitBuffer()*. The host software must call this function right after enabling the FLEX Alphanumeric Chip.

Example 3-10   Function *BuildInitBuffer()*

```
/*************************************************************************
 *   FUNCTION NAME: BuildInitBuffer()                                   *
 *   Date created:   07/26/96                                           *
 *                                                                      *
 *   Description:                                                       *
 *   This function dynamically stores data in the initialization        *
 *   buffer.  Note that this function is designed for software testing, *
 *   and better illustration of how the initialization buffer is       *
 *   constructed.  For users' end-products, initialization data can be  *
 *   stored in EEPROM for better performance and reducing code size.   *
 *                                                                      *
 *   INPUT: None                                                        *
 *   OUTPUT: None                                                       *
 *                                                                      *
 *   GLOBAL VARIABLES:                                                  *
 * init_buffer: Pointer to the beginning of the                        *
 * initialization buffer.                                              *
 *************************************************************************/

 void BuildInitBuffer()
 {
    volatile FS_DRIVER_STRUCT *ptr1;
    volatile E_NOTIFY_STRUCT *ptr2;
    volatile MMLITE_STRUCT *ptr3;
    volatile FILTER_STRUCT *ptr4;
    volatile FLEXCHIP_STRUCT *ptr5;
```

Example 3-10 Function *BuildInitBuffer()* (Continued)

```
   /* Create FLEXstack Driver Initialization module */
   ptr1 = (FS_DRIVER_STRUCT *)
_malloc((UVAR32)sizeof(FS_DRIVER_STRUCT));
   ptr1->fs_driver_hdr.size = sizeof(FS_DRIVER_STRUCT);
   ptr1->fs_driver_hdr.flag = 0;
   ptr1->fs_driver_hdr.type = 1;
   ptr1->fs_driver_data.msgStorSize = STATUS_TABLE_SIZE;
   ptr1->fs_driver_data.msgBldSize = MESSAGE_BLD_SIZE;
            ptr1->fs_driver_data.msgstraddr = (void*)
_malloc((UVAR32)STATUS_TABLE_SIZE * sizeof(UCHAR));
   ptr1->fs_driver_data.msgbldaddr = (void*)
_malloc((UVAR32)MESSAGE_BLD_SIZE * sizeof(UCHAR));

   /* The pointer to this module is the beginning of the
      initialization buffer */
   init_buffer = (UVAR8*) ptr1;

   /* Create Event Notification module */
   ptr2 = (E_NOTIFY_STRUCT *)
_malloc((UVAR32)sizeof(E_NOTIFY_STRUCT));
   ptr1->fs_driver_hdr.next = (UVAR8*) ptr2;

   ptr2->e_notify_hdr.size = sizeof(E_NOTIFY_STRUCT);
   ptr2->e_notify_hdr.flag = 0;
   ptr2->e_notify_hdr.type = 2;
   ptr2->e_notify_data.FChipMask = 0;
   ptr2->e_notify_data.FStackMask = 0;
   ptr2->e_notify_data.BIWMask = 0;

   /* create Message Manager Lite module */
   ptr3 = (MMLITE_STRUCT *) _malloc((UVAR32)sizeof(MMLITE_STRUCT));
   ptr2->e_notify_hdr.next = (UVAR8*) ptr3;

   ptr3->mmlite_hdr.size = sizeof(MMLITE_STRUCT);
   ptr3->mmlite_hdr.flag = 0;
   ptr3->mmlite_hdr.type = 5;
   ptr3->mmlite_data.numNodes = 0x009E;
   ptr3->mmlite_data.nodeSize = 0x0020;
   ptr3->mmlite_data.poolStartAddress = (UVAR8*)
_malloc((UVAR32)NUM_NODES * NODE_SIZE * sizeof(UCHAR));
   ptr3->mmlite_data.maxHdls = 0x10;
   ptr3->mmlite_data.hdlPoolAddress = (UVAR8*)
_malloc((UVAR32)MAX_HANDLES * NUM_MSGS * sizeof(UCHAR));
```

Example 3-10 Function *BuildInitBuffer()* (Continued)

```
    /* Create Message Filter Module */
    ptr4 = (FILTER_STRUCT *) _malloc((UVAR32)sizeof(FILTER_STRUCT));
    ptr3->mmlite_hdr.next = (UVAR8*) ptr4;

    ptr4->filter_hdr.size = sizeof(FILTER_STRUCT);
    ptr4->filter_hdr.flag = 0;
    ptr4->filter_hdr.type = 4;
    ptr4->filter_data.hdlPoolAddress = (void*)
_malloc((UVAR32)16 * sizeof(UCHAR));
    ptr4->filter_data.maxHdls = 1;
    ptr4->filter_data.filterOptions = 1;
    ptr4->filter_data.numSpare = 33;

    /* Create FLEXchip Initialization Module */
    ptr5 = (FLEXCHIP_STRUCT *)
_malloc((UVAR32)sizeof(FLEXCHIP_STRUCT));
    ptr4->filter_hdr.next = (UVAR8*) ptr5;

    ptr5->flexchip_hdr.size = sizeof(FLEXCHIP_STRUCT);
    ptr5->flexchip_hdr.flag = 1;
    ptr5->flexchip_hdr.type = 3;
    ptr5->flexchip_hdr.next = NULL;
    ptr5->flexchip_data.Major = 0x30;
    ptr5->flexchip_data.Minor = 0x30;
    ptr5->flexchip_data.reserved = 0;
    ptr5->flexchip_data.cmapSize = 0x2B;

    /*
     * These are packets of information to be sent out to configure
     * the FLEXchip IC.  Please refer to FLEXchip User's Manual on
     * how to construct these parameters.
     */

    ptr5->cmds[0] = 0x010100D8;/* Configuration packet */
    ptr5->cmds[1] = 0x03400000;/* All frame mode */
    ptr5->cmds[2] = 0x0400FFFF;/* Reserved */
    ptr5->cmds[3] = 0x05000000;/* Reserved */
    ptr5->cmds[4] = 0x0F000700;/* Receiver line control */
    ptr5->cmds[5] = 0x10000132;/* Off settings */
    ptr5->cmds[6] = 0x11000132;/* Warm-up 1 setting */
    ptr5->cmds[7] = 0x12000132;/* Warm-up 2 setting */
    ptr5->cmds[8] = 0x13000132;/* Warm-up 3 setting */
    ptr5->cmds[9] = 0x14000132;/* Warm-up 4 setting */
```

Example 3-10 Function *BuildInitBuffer()* (Continued)

```
    ptr5->cmds[10] = 0x15000132;/* Warm-up 5 setting */
    ptr5->cmds[11] = 0x16007102;/* 3200 sync. configuration */
    ptr5->cmds[12] = 0x17006100;/* 1600 sync. configuration */
    ptr5->cmds[13] = 0x18003100;/* 3200 data configuration */
    ptr5->cmds[14] = 0x19002100;/* 1600 data configuration */
    ptr5->cmds[15] = 0x1A000000;/* Shut-down 1 config. */
    ptr5->cmds[16] = 0x1B000000;/* Shut-down 2 config. */
    ptr5->cmds[17] = 0x2000FFFF;/* Frames assignment 112-127*/
    ptr5->cmds[18] = 0x2100FFFF;/* Frames assignment 96-111 */
    ptr5->cmds[19] = 0x2200FFFF;/* Frames assignment 80-95 */

    ptr5->cmds[20] = 0x2300FFFF;/* Frames assignment 64-79 */
    ptr5->cmds[21] = 0x2400FFFF;/* Frames assignment 48-63 */
    ptr5->cmds[22] = 0x2500FFFF;/* Frames assignment 32-47 */
    ptr5->cmds[23] = 0x2600FFFF;/* Frames assignment 16-31 */
    ptr5->cmds[24] = 0x2700FFFF;/* Frames assignment 0-15 */
    ptr5->cmds[25] = 0x7800000F;/* User address enable */
    ptr5->cmds[26] = 0x801F0063;/* User address 0 */
    ptr5->cmds[27] = 0x811F11E9;/* User address 1 */
    ptr5->cmds[28] = 0x821F2700;/* User address 2 */
    ptr5->cmds[29] = 0x831F2696;/* User address 3 */

    ptr5->cmds[30] = 0x84090000;/* User address 4 */
    ptr5->cmds[31] = 0x85088765;/* User address 5 */
    ptr5->cmds[32] = 0x86100005;/* User address 6 */
    ptr5->cmds[33] = 0x871C0CCE;/* User address 7 */
    ptr5->cmds[34] = 0x881F2701;/* User address 8 */
    ptr5->cmds[35] = 0x891F2702;/* User address 9 */
    ptr5->cmds[36] = 0x8A400001;/* User long address 10 */
    ptr5->cmds[37] = 0x8B5F8000;/* User long address 11 */
    ptr5->cmds[38] = 0x8C400002;/* User long address 12 */
    ptr5->cmds[39] = 0x8D5F9000;/* User long address 13 */

    ptr5->cmds[40] = 0x8E0EEEEE;/* User address 14 */
    ptr5->cmds[41] = 0x8F0FFFFF;/* User address 15 */
    ptr5->cmds[42] = 0x02000001;/* Control packet */

    return;
}
```

3.2.5   Retrieving Paging Messages from FLEX One-Way Stack

This section illustrates sample code acting as the host software running on the MCU.
When FLEX One-Way Stack software has a new paging message in the buffer, the *main ()*
and *GetPage* functions retrieve the message and display it to the user. These functions
also illustrate the API provided in the software. Function *main ()* code is shown in
Example 3-11. Function *Getpage* code is shown in **Example 3-12**.

Example 3-11   Function *main()*

```
/**********************************************************************
 * FUNCTION NAME: main()                                             *
 * Date created: 07/26/96                                            *
 *                                                                   *
 * Description:                                                      *
 * This is the main module that controls the FLEXstack software      *
 * to communicate with the FLEXchip IC.                              *
 *                                                                   *
 * INPUT: None                                                       *
 * OUTPUT: None                                                      *
 **********************************************************************/

main()
{
        UVAR8 i;

        /* Initialize the SPI communication module to enable FLEXchip */
        _FlexICInit();

        /*
         * Initializes some global variables for storage of
         * FLEX messages.  The msgtable stores up to a certain number
         * of message IDs, allowing the users to retrieve previously
         * received messages.
         */
        packet = (PACKET*) _Lmalloc((UVAR32)sizeof(PACKET));
        for (i = 0; i < MSGTABLESZ; i++);
          msgtable[i] = 0;
```

Example 3-11 Function *main()* (Continued)

```
        /* Prepare the initialization buffer */
       BuildInitBuffer();
      /*
       * Initialize FLEXstack - init_buffer is the pointer to the
       * beginning of the initialization buffer.
       */
      FStkInit(init_buffer);

      /*
       * Poll FLEXstack for new messages.  Programmer can also uses
       * timer interrupt to call FLEXstack function periodically
       * for new message.
       */
      while(1)
      {
          delay(10000);
          FLEXstack();
      }
}
```

Example 3-12 Function *GetPage*

```
/***********************************************************
 * FUNCTION NAME: GetPage(MSGID)                           *
 * Date created:  07/26/96                                 *
 *                                                         *
 * Description:                                            *
 * This function retrieves a message from FLEXstack        *
 * and calls function displayMessage to display the message *
 * on the LCD.                                             *
 *                                                         *
 * INPUT: msgid: MSGID-type variable - a number to         *
 * identify the message to be retrieved                    *
 * OUTPUT: None                                            *
 *                                                         *
 * GLOBAL VARIABLES                                        *
 *                                                         *
 ***********************************************************/

void GetPage(MSGID msgid)
{
   HEADER* header;
   UVAR8* handle;
   char pageData[64];
   int i = 0;
   FILTERDATA filterData;
```

Example 3-12 Function *GetPage* (Continued)

```c
   /* Read the first message specified in the table */
   handle = (UVAR8*) FStkOpen(msgid, (UVAR8)FILTERED);

   /* Get the message attributes, including message length */
          header = (HEADER *)
FStkGetAttrib(msgid,(UVAR8)(sizeof(HEADER)), ATTRIB_BUF);

   /*
    * In the case that we cannot open the message, return
             to the caller
    */
   if (!header) {
      (void)FStkClose(handle);
      return;
   }

   if (BTST(header->e.msg_attrib, HDR_TONE_ONLY)) {
      pageData[i] = '\0';
   }
   else {

      /* Retrieve message and store it in pageData variable */
      for (i = 0; i < header->e.msg_char_size; i++) {
         (void)FStkRead(handle, (UVAR8*) &filterData);
         if ((filterData.data >= 0x20) && (filterData.data
                       < 0x7F))
        pageData[i] = (UVAR8) filterData.data;
         else
        pageData[i] = 0x2E;
      }
      pageData[i] = '\0';
   }
   (void)FStkClose(handle);

   /* Display the paging message on LCD */
   displayMessage(pageData);
}
```

dsp