OVERVIEW
This application note describes a KEELOQ code hopping decoder implemented on a Microchip Mid-range Enhanced FLASH MCU (PIC16F872). The software has been designed as a group of independent modules (standard C source files "C").
For clarity and ease of maintenance, each module covers a single function. Each module can be modified to accommodate a different behavior, support a different MCU, and/or a different set of peripherals (memories, timers, etc.).

KEY FEATURES
The set of modules presented in this application note implement the following features:
• Source compatible with HITECH and CCS C compilers
• Pin out compatible with PICDEM-2 board
• Normal Learn mode
• Learn up to 8 transmitters, using the internal EEPROM memory of PIC16F872
• Interrupt driven Radio Receive (PWM) routine
• Compatible with all existing KEELOQ hopping code encoders with PWM transmission format selected, operating in "slow mode" (Te = 400 µs)
• Automatic synchronization during receive, using a 4 MHz RC oscillator

Notice:
This is a non-restricted version of Application Note AN745 which is available under the KEELOQ License Agreement. The license agreement can be ordered from the Microchip Literature Center as DS40149.
DESIGN OBJECTIVES

Each module has been designed for maximum simplic-ity and maintainability. Whenever possible, we favored clarity of design over efficiency in order to show the basic concepts of the design of a KEELoQ decoder without the constraints of previous PIC16CSX implementa-tions such as limited RAM, STACK, or other resources.

To achieve maximum ease in maintenance, we adopted “modern” C language programming tech-niques, specifically:

- All pin assignments are mapped through #define directives. This results in almost complete code independence from the specific pin out chosen.
- Drivers to peripherals that are specific to a given processor type (such as PIC16F872) have been encapsulated in more generic modules.
- Function input and output values are documented.
- Pseudo-graphical representation of the data structures used and program flow is commented whenever possible.

Although the code can be compiled in a set of independent object files and then linked together to build the actual application, we kept all the modules included in line with the main module to retain compatibility with compilers that have no linker such as CCS PIC C.

MODULES OVERVIEW

The code presented in this application note is composed of the following basic modules:

RXI.C interrupt driven receiver
KEYGEN.C KEELoQ key generation routines implementing Normal Mode
FASTDEC.C KEELoQ decrypt routine
MEM-87X.C PIC16F87X EEPROM driver
TABLE.C transmitters table memory management (linear list)
MAIN.C the actual initialization and main loop

FIGURE 2: MODULES OVERVIEW
The receiver module has been developed around a fast and independent Interrupt Service Routine (ISR). The whole receiving routine is implemented as a simple state machine that operates on a fixed time base. This can be used to produce a number of virtual timers. The operation of this routine is completely transparent to the main program and similar to a UART. In fact, the interrupt routine consumes only 30% of the computational power of the MCU working in the background.

After a complete code-word of 66 bits has been properly received and stored in a 9 bytes buffer, a status flag (RF_FULL) is set and the receiver becomes idle.

It is the responsibility of the main program to make use of the data in the buffer and to clear the flag to enable the receiving of a new code-word.

In order to be compatible with all KEEL7 encoders, with or without oscillator tuning capabilities, the receiver routine constantly attempts to resynchronize with the first rising edge of every bit in the incoming code-word. This allows the decoder to operate from an inexpensive (uncalibrated) RC clock. In doing so, the last rising edge/bit of every code-word is lost (resulting in an effective receive buffer capacity of 65-bit).

For HCS20X and HCS30X encoders this implies that the REPEAT bit (being the 66th) cannot be captured. While for Advanced Encoders like the HCS36X or HCS4XX, the reader can easily modify the definition of the constant BIT_NUM to 68 to receive all bits transmitted with exception of the last queue bit Q1 (being the 69th), again rarely used.

The only resource/peripheral used by this routine is Timer0 and the associated Overflow Interrupt. This is available on every mid-range PICmicro microcontroller. Timer0 is reloaded on overflow, creating a time base (of about 1/3 Te = 138 µs). The same interrupt service routine also provides a virtual 16-bit timer, derived from the same base period, called XTMR.

**FIGURE 3: CODE-WORD TRANSMISSION FORMAT**

---

**FIGURE 4: CODE-WORD ORGANIZATION**
Since the radio input is polled (for 1 µs) on multiples of the base period (138 µs), the chance of a glitch (short noise pulse) disturbing the receiver is reduced.

Further, since the time base produced is constant, the same interrupt service routine could easily be extended to implement a second UART as a separate state machine for full duplex asynchronous communication up to 1,200 baud at 4 MHz.

Note: This would also require the main oscillator to be crystal based.

Other implementations of the same receiver module can be obtained using other peripherals and detection techniques. These include:

- Using the INT pin and selectable edge interrupt source
- Using the Timer1 and CCP module in capture mode
- Using comparator inputs interrupt

All of these techniques pose different constraints on the pin out, or the PICmicro MCU, that can be used. This would lead to different performances in terms of achievable immunity from noise and or CPU overhead, etc.

**FAST DECRYPTION MODULE**

This module contains an implementation of the KEELOQ decryption algorithm that has been optimized for speed on a mid-range PICmicro microcontroller. It allows fast decryption times for maximum responsiveness of the system even at 4 MHz clock.

The decryption function is also used in all learning schemes and represents the fundamental building block of all KEELOQ decoders.

**KEY GENERATION MODULE**

This module shows a simple and linear implementation of the Normal Learn Key Generation.

This module uses the KEELOQ Decrypt routine from the Fast Decryption module to generate the key at every received code-word instead of generating it during the learn phase and storing it in memory. The advantage is a smaller Transmitter Record of 8 bytes instead of 16 bytes (see Table 2). This translates in a double number of transmitters that can be learned using the 64 byte internal EEPROM available inside the PIC16F872. This space reduction comes at the expense of more computational power required to process every code-word. When a new code-word is received, the key generation algorithm is applied (Normal Learn) and the resulting Description key is placed in the array DKEY[0..7]. During a continuous transmission (the user is holding the button on the transmitter), the key generation is not repeated, to save time, the last computed Decryption Key value is used safely instead (the serial number being the same).

Due to double buffering of the receiver and the PICmicro MCU execution speed and efficiency (even running at 4 MHz only), it is possible to receive and decrypt, at the same time, each and every incoming code-word.

For an overview of some of the different security levels that can be obtained through the use of different key generation/management schemes, refer to the "Secure Data Products Handbook" [DS40168] (Section 1, KEELOQ Comparison Chart, Security Level Summary).

A detailed description of the Normal Learn key generation scheme can be found in Technical Brief TB003 "An Introduction To KEELOQ Code Hopping" [DS91002].

More advanced Key Generation Schemes can be implemented replacing this module with the techniques described in Technical Brief TB001 "Secure Learning RKE Systems Using KEELOQ Encoders" [DS91000].

**TABLE MODULE**

One of the major tasks of a decoder is to properly maintain a database that contains all the unique ID’s (serial numbers) of the learned transmitters. In most cases, the database can be as simple as a single table, which associates those serial numbers to the synchronization counters (that are at the heart of the hopping code technology).

This module implements the easiest of all methods, a simple "linear list" of records.

Each transmitter learned is assigned a record of 8 bytes (shown in Table 2), where all the relevant information is stored and regularly updated.

**TABLE 2: TRANSMITTER RECORD**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>FCODE</td>
<td>Function code (4 bits) and upper 4 Serial Number bits [24..28]</td>
</tr>
<tr>
<td>+1</td>
<td>IDLo</td>
<td>Serial Number bits [0..7]</td>
</tr>
<tr>
<td>+2</td>
<td>IDHi</td>
<td>Serial Number bits [8..15]</td>
</tr>
<tr>
<td>+3</td>
<td>IDM1</td>
<td>Serial Number bits [16..23]</td>
</tr>
<tr>
<td>+4</td>
<td>SYNCH</td>
<td>Sync Counter 8 MSB</td>
</tr>
<tr>
<td>+5</td>
<td>SYNCL</td>
<td>Sync Counter 8 LSB</td>
</tr>
<tr>
<td>+6</td>
<td>SYNCH2</td>
<td>Second copy of SYNCH</td>
</tr>
<tr>
<td>+7</td>
<td>SYNCL2</td>
<td>Second copy of SYNCL</td>
</tr>
</tbody>
</table>
The 16-bit synchronization counter value is stored in memory twice because it is the most valuable piece of information in this record. It is continuously updated at every button press on the remote. When reading the two stored synchronous values, the decoder should verify that the two copies match. If not, it can adopt any safe resync or disable technique required depending on the desired system security level.

The current implementation limits the maximum number of transmitters that can be learned to eight. This is due to the size of the internal EEPROM of the PIC16F872.

This number can be changed to accommodate different PICmicro models and memory sizes by modifying the value of the constant MAX_USER.

The simple "linear list" method employed can be scaled up to some tens of users. But due to its simplicity, the time required to recognize a learned transmitter grows linearly with the length of the table.

It is possible to reach table sizes of thousands of transmitters by replacing this module with another module that implements a more sophisticated data structure like a "Hash Table" or other indexing algorithms.

Again due to the simplicity of the current solution, it is not possible to selectively delete a transmitter from memory. The only delete function available is a Bulk Erase (complete erase of all the memory contents) that happens when the user presses the Learn button for up to 10 seconds. (The LED will switch off. At the release of the button, it will flash once to acknowledge the delete command). To allow for selective transmitter removal from memory, more sophisticated techniques will be analyzed in future application notes, by simply replacing/updating this module.

**MEM-87X MODULE**

This module is optimized to drive the internal EEPROM of the PIC16F87X device.

The module makes the memory generically accessible by means of two routines RDword and WRword that respectively read and write a 16-bit value out of an even address specified in parameter IND.

Replacing this module with the appropriate drivers, (and adapting the pin out) make possible the use of any kind of nonvolatile memory. This includes internal and external serial EEPROMs (Microwire®, SPI™ or I²C™ bus) of any size up to 64 Kbytes.

**THE MAIN PROGRAM**

The main program is reduced to a few pages of code. The behavior is designed to mimic the basic behavior of the HCS512 integrated decoder, although just the parallel output is provided (no serial interface).

Most of the time, the main loop goes idle waiting for the receiver to complete reception a full code-word.

Double buffering of the receiver is done in RAM, in order to immediately re-enable the reception of new codes and increase responsiveness and perceived range.

**CONCLUSION**

The C language source increases the readability of the program structure and eases the maintenance. This benefit has come at the cost of the program size. That in terms of memory words, has considerably increased over the equivalent code written in assembly (more than 30% larger).

Selecting a FLASH PICmicro microcontroller from the mid-range family as the target MCU allows us to make the code simpler and cleaner. It also provides larger RAM memory space and a deeper hardware stack. Interrupts have been used to "virtualize" the receiving routine as a software peripheral and to free the design of the hard real time constraint that it usually poses.

Still, many of the resources available on the PIC16F872 are left unused and available to the designer. These include:

- Timer1, a 16-bit timer
- Timer1 oscillator, a low power oscillator for real time clock
- CCP module, capable of capture, compare and PWM generation
- Timer2, an 8-bit timer, with auto reload
- 10-bit A/D converter with a 5 channel input multiplexer

We resisted introducing extra features and optimizations in favor of clarity. For example:

- Speed optimizations and code compacting
- More complex key generation schemes
- Multiple manufacturer codes
- Co-processor functionality
- Advanced user entry and deletion commands
- Large memory tables (up to 8,000 users)
- Serial interface to PDAs and/or terminals for memory management and logging

These are left as exercises to the advanced reader/designer or as suggestions for further application notes.
MEMORY USAGE FUNCTION
HEADERS

Compiling with HITECH 7.86r3

Memory Usage Map:

<table>
<thead>
<tr>
<th>Program ROM</th>
<th>$0000 - $00A8</th>
<th>$00A9</th>
<th>169 words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program ROM</td>
<td>$04A0 - $07FF</td>
<td>$0351</td>
<td>849 words</td>
</tr>
<tr>
<td>Program ROM</td>
<td>$2000 - $2005</td>
<td>$0006</td>
<td>6 words</td>
</tr>
<tr>
<td>Program ROM</td>
<td>$2007 - $2007</td>
<td>$0001</td>
<td>1 word</td>
</tr>
<tr>
<td></td>
<td>$0401</td>
<td></td>
<td>1025 words total Program ROM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bank 0 RAM</th>
<th>$0021 - $006D</th>
<th>$004D</th>
<th>77 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank 0 RAM</td>
<td>$0070 - $0074</td>
<td>$0005</td>
<td>5 bytes</td>
</tr>
<tr>
<td></td>
<td>$0052</td>
<td></td>
<td>82 bytes total Bank 0 RAM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bank 0 Bits</th>
<th>$0100 - $0105</th>
<th>$0006</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>total Bank 0 bits</td>
</tr>
</tbody>
</table>

CCS PCW C Compiler, Version 2.535, 4511

Filename: D:\WORK\SMAD\AN\DECC\MAIN.LST

ROM used: 1155 (28%) 1155 (28%) including unused fragments

RAM used: 71 (37%) at main () level 84 (44%) worst case

Stack: 4 worst case (3 in main +1 for interrupts)

REFERENCES

KEELOQ Code Hopping Decoder on a PIC16C56 AN642 DS00642
Converting NTQ105/106 to Designs to HCS200/300s AN644 DS00644
Code Hopping Security System on a PIC16C57 AN645 DS00645
Secure Learn Code Hopping Decoder on a PIC16C56 AN652 DS00652
KEELOQ Simple Code Hopping Decoder AN659 DS00659
KEELOQ Code Hopping Decoder on a PIC16C56 AN661 DS00661
(public version)
Secure Learn Code Hopping Decoder on a PIC16C56 AN662 DS00662
(public version)
KEELOQ Simple Code Hopping Decoder (public version) AN663 DS00663
Using KEELOQ to Generate Hopping Passwords AN665 DS00665
PICmicro Mid-Range MCU Code Hopping Decoder AN662 DS00672
HCS410 Transponder Decoder using a PIC16C56 AN675 DS00675
Modular PICmicro Mid-Range MCU Code Hopping Decoder AN742 DS00742
Secure Learning RKE Systems Using KEELOQ Encoders TB001 DS91000
An Introduction to KEELOQ Code Hopping TB003 DS91002
A Guide to Designing for EuroHomelink Compatibility TB021 DS91021
KEELOQ Decryption & IFF Algorithms TB030 DS91030
KEELOQ Decryption Routines in C TB041 DS91041
Interfacing a KEELOQ Encoder to a PLL Circuit TB042 DS91042
KEELOQ CRC Verification Routines TB043 DS91043

DS00744A-page 6 Preliminary © 2001 Microchip Technology Inc.
APPENDIX A:  DECHIT H SOURCE CODE

// Module DECHIT.h

// include this file when using the HiTech C compiler

#define HITECH

#include <pic.h>
#include <string.h>

typedef unsigned char byte;
#define TRUE 1

typedef signed char sbyte;
#define FALSE 0

typedef signed int word;
#define ON 1
#define OFF 0

#define BIT_TEST(x, y) ((x) & (1<<(y))) != 0

#define UNPROTECT | (FOSC1 | FOSC0) | BODEN;

void __IDLOC(0x1234);                    // define ID locations

__CONFIG( UNPROTECT | (FOSC1 | FOSC0) | BODEN); // set config word

__IDLOC(0x1234); // define ID locations
APPENDIX B: DECCCS H SOURCE CODE

// Module DECCCS.h
//
// include this file when using the CCS C compiler
//
#define CCS

#define DEVICE PIC16C63

typedef short bit;         // one bit
typedef unsigned int byte; // one byte unsigned
typedef signed   int sbyte; // one byte signed
typedef signed  long word; // one word signed

// un-supported directives
#define static
#define volatile
#define interrupt

#define TRUE    1
#define FALSE   0
#define ON      1
#define OFF     0

// F872 special function registers

#define TMR0 = 0x01       // Timer 0
#define T0IF = 0x0B.2     // Timer 0 interrupt flag
#define T0IE = 0x0B.5     // Timer 0 interrupt enable
#define GIE  = 0x0B.7     // Global Interrupt Enable

#define OPTION = 0x81     // prescaler timer0 control
#define ADCON1 = 0x9f     // A/D converter control

#define TRISA = 0x85      // PORT A
#define PORTA = 0x05      // PORT A
#define RA0 = 0x05.0
#define RA1 = 0x05.1
#define RA2 = 0x05.2
#define RA3 = 0x05.3
#define RA4 = 0x05.4
#define RA5 = 0x05.5

#define TRISB = 0x86      // PORT B
#define PORTB = 0x06      // PORT B
#define RB0 = 0x06.0
#define RB1 = 0x06.1
#define RB2 = 0x06.2
#define RB3 = 0x06.3
#define RB4 = 0x06.4
#define RB5 = 0x06.5
#define RB6 = 0x06.6
#define RB7 = 0x06.7

#define TRISC = 0x87      // PORT C
#define PORTC = 0x07      // PORT C

// internal EEPROM access
#define BRADR = 0x10d
#define BREADR = 0x10c
#define BEREAD1 = 0x18c
#define BEREAD2 = 0x18d
#define WR = 0x18c.1
#bit RD = 0x18c.0
#bit WREN = 0x18c.2
#bit EEPGD = 0x18c.7

// macro versions of EEPROM write and read
#define EEPROM_WRITE(addr, value) while(WR)continue;EEADR=(addr);EEDATA=(value);EEPGD=0;GIE=0;WREN=1;
    EECON2=0x55;EECON2=0xAA;WR=1;WREN=0
#define EEPROM_READ(addr) ((EEADR=(addr)),(EEPGD=0),(RD=1),EEDATA)

// configuration and ID locations
#define FUSES RC, NOWDT, NOPROTECT, BROWNOUT
#define ID 0x1234
APPENDIX C: MAIN C SOURCE CODE

﻿// *********************************************************************
//  Filename:   MAIN.c
// *********************************************************************
//  Author:     Lucio Di Jasio
//  Company:    Microchip Technology
//  Revision:   Rev 1.00
//  Date:       08/07/00
//  Keeloq Normal Learn Decoder on a mid range PIC
//  full source in C
//  Compiled using HITECH PIC C compiler v.7.93
//  Compiled using CCS PIC C compiler v. 2.535
// ********************************************************************

#include "decccs.h"  // uncomment for CCS compiler
#include "dechit.h" // uncomment for HiTech compiler

// I/O definitions for PIC16F872
// compatible with PICDEM-2 demo board

#define RFIn   RA1           // i radio signal input
#define Learn  RA4           // i learn button
#define Out0   RB0           // o S0 output
#define Out1   RB1           // o S1 output
#define Out2   RB2           // o S2 output
#define Out3   RB3           // o S3 output
#define Led    RB4           // o LearnOut Led
#define Vlow   RB5           // o low battery
#define MASKPA  0xff         // port A I/O config (all input)
#define MASKPB  0xc0         // port B I/O config (6 outputs)
#define MASKPC  0xff         // port C I/O config (NU)

byte Buffer[9];  // receive buffer
// keeloq receive buffer map
//
// | Plain text                                | Encrypted
// RV000000.KKKKIII.IIIIIIIIII.IIIIIIIIIIIIIII.KKKKO000.DDDDDDD.SSSSSSSS.SS
// |                                           |
// //
// // I=S/N  -> SERIAL NUMBER       (28 BIT)
// // K=KEY   -> buttons encoding     (4 BIT)
// // S=Sync  -> Sync counter         (16 BIT)
// // D=Disc  -> Discrimination bits  (10 BIT)
// // R=Rept  -> Repeat/first         (1 BIT)
// // V=Vlow  -> Low battery          (1 BIT)
// //
////-- alias -------------------------------------------------------------
//
#define     HopLo   Buffer[0] // sync counter
#define     HopHi   Buffer[1] //
#define     DisLo   Buffer[2] // discrimination bits LSB
#define     DOK     Buffer[3] // Disc. MSB + Ovf + Key
#define     IDLo    Buffer[4] // S/N LSB
#define     IDMi    Buffer[5] // S/N
#define     IDHi    Buffer[6] // S/N MSB
#define S0  5   // Buffer[3] function codes
#define S1  6   // Buffer[3] function codes
#define S2  7   // Buffer[3] function codes
#define S3  4   // Buffer[3] function codes
#define VFlag  7 // Buffer[8] low battery flag

// timings
//
#define TOUT    5           // 5 * 71ms = 350ms output delay
#define TFLASH  2           // 4 * 71ms = 280ms half period
#define TLEARN  255         // 255 * 71ms = 18s learn timeout

// byte Flags;                 // various flags
byte CLearn, CTLearn;       // learn timers and counter
byte CFlash, CTFlash;       // led flashing timer and counter
byte COut;                  // output timer
byte FCode;                 // function codes and upper nibble of serial number

word Dato;                  // temp storage for read and write to mem.
word Ind;                   // address pointer to record in mem.
word Hop;                   // hopping code sync counter
word EHop;                  // last value of sync counter (from EEPROM)
word ETemp;                 // second copy of sync counter

//
// interrupt receiver
//
#include "rxim.c"

//
// external modules
//
#include "mem-87x.c"  // EEPROM I2C routines
#include "table.c"     // TABLE management
#include "keygen.c"     // Keeloq decrypt and normal keygen

// prototypes
void Remote( void);

// MAIN
// Main program loop, I/O polling and timing
void main ()
{
  // init
  ADCON1 = 0x7;        // disable analog inputs
  TRISA = MASKPA;      // set i/o config.
  TRISB = MASKPB;
  TRISC = MASKPC;
  PORTA = 0;          // init all outputs
  PORTB = 0;
  PORTC = 0;
  OPTION = 0x8f;      // prescaler assigned to WDT, TMR0 clock/4, no pull ups
  
  CTLearn = 0;        // Learn debounce
  CLearn = 0;         // Learn timer
  COut = 0;           // output timer
  CFlash = 0;         // flash counter
  CTFlash = 0;        // flash timer
  FLearn = FALSE;     // start in normal mode
  F2Chance = FALSE;   // no resynchronization required
  InitReceiver();     // enable and init the receiver state machine

  // main loop
  while ( TRUE)
  {
    if ( RFFull)       // buffer contains a message
      Remote();

    // loop waiting 512* period = 72ms
    if ( XTMR < 512)
      continue;       // main loop

    // once every 72ms
    XTMR = 0;
    
    // re-init fundamental registers
    ADCON1 = 0x7;        // disable analog inputs
    TRISA = MASKPA;      // set i/o config.
    TRISB = MASKPB;
    TRISC = MASKPC;
    OPTION = 0x0f;      // prescaler assigned to WDT, TMR0 clock/4, pull up
    T0IE = 1;
    GIE = 1;

    // poll learn
    if ( !Learn)       // low -> button pressed
    {
      CLearn++;
    
      if ( !Learn)       // low -> button pressed
      {
      
      }
// pressing Learn button for more than 10s -> ERASE ALL
if (CLearn == 128) // 128 * 72 ms = 10s
{
    Led = OFF; // switch off Learn Led
    while( !Learn); // wait for button release
    Led = ON; // signal Led on
    ClearMem(); // erase all command!
    COut = TOUT; // single long flash pulse time
    // timer will switch off Led
    CLearn = 0; // reset learn debounce
    FLearn = FALSE; // exit learn mode
}

// normal Learn button debounce
if (CLearn == 4) // 250ms debounce
{
    FLearn = TRUE; // enter learn mode command!
    CTLearn = TLEARN; // load timeout value
    Led = ON; // turn Led on
}
else  CLearn=0; // reset counter

// outputs timing
if (COut > 0) // if timer running
{
    COut--; if (COut == 0) // when it reach 0
    {
        Led = OFF; // all outputs off
        Out0 = OFF;
        Out1 = OFF;
        Out2 = OFF;
        Out3 = OFF;
        Vlow = OFF;
    }
}

// Learn Mode timeout after 18s (TLEARN * 72ms)
if (CTLearn > 0)
{
    CTLearn--; // count down
    if (CTLearn == 0) // if timed out
    {
        Led = OFF; // exit Learn mode
        FLearn = FALSE;
    }
}

// Led Flashing
if (CFlash > 0)
{
    CTFlash--; // count down
    if (CTFlash == 0) // if timed out
    {
        CTFlash = TFLASH; // reload timer
        CFlash--; // count one flash
        Led = OFF; // toggle Led
        if (CFlash & 1)
            Led = ON;
    }
}

} // main loop
} // main
// Remote Routine
// Decrypts and interprets receive codes
// Does Normal Operation and Learn Mode
// INPUT: Buffer contains the received code word
// OUTPUT: S0..S3 and LearnOut
// void Remote()
{
    // a frame was received and is stored in the receive buffer
    // move it to decryption Buffer, and restart receiving
    memcpy( Buffer, B, 9); // ready to receive a new frame

    // decoding
    NormalKeyGen(); // compute the decryption key
    Decrypt(); // decrypt the hopping code portion

    if ( DecCHK() == FALSE) // decryption failed
        return;

    if ( FLearn)
    {
        // Learn Mode
        if ( Find()== FALSE) // could not find the Serial Number in memory
            { // look for new space
                if ( !Insert()) // fail if no memory available
                    return;
            }

        // ASSERT Ind is pointing to a valid memory location
        IDWrite(); // write Serial Number in memory
        FHopOK = TRUE; // enable updating Hopping Code
        HopUpdate(); // Write Hopping code in memory

        CFlash = 32; // request Led flashing
        CTFlash = TFLASH; // load period timer
        Led = TRUE; // start with Led on
        FLearn = FALSE; // terminate successfully Learn
    } // Learn

    else // Normal Mode of operation
    {
        if ( Find()== FALSE)
            return;

        if ( !HopCHK()) // check Hopping code integrity
            return;

        if ( FSame) // identified same code as last memorized
            { // if output is still active
                if ( Cout >0)
                    Cout = TOUT; // reload timer to keep active
                else
                    return; // else discard
            }

        else // hopping code incrementing properly
        {
HopUpdate(); // update memory

// set outputs according to function code
if (BIT_TEST(Buffer[3], S0))
    Out0 = ON;
if (BIT_TEST(Buffer[3], S1))
    Out1 = ON;
if (BIT_TEST(Buffer[3], S2))
    Out2 = ON;
if (BIT_TEST(Buffer[3], S3))
    Out3 = ON;

// set low battery flag if necessary
if (BIT_TEST(Buffer[8], VFlag))
    Vlow = ON;

// check against learned function code
if (((Buffer[7] ^ FCode) & 0xf0) == 0)
    Led = ON;

// init output timer
COut = TOUT;
#endif // recognized
} // normal mode

} // remote
// **********************************************************************
//  Filename:   RXI.c
// *********************************************************************
//  Author:     Lucio Di Jasio
//  Company:    Microchip Technology
//  Revision:   Rev 1.00
//  Date:       08/07/00
//
//  Interrupt based receive routine
//
//  Compiled using HiTech PIC C compiler v.7.93
//  Compiled using CCS    PIC C compiler v.2.535
// ********************************************************************
#define CLOCK           4       // MHz
#define TE            400       // us
#define OVERSAMPLING    3
#define PERIOD          TE/OVERSAMPLING*4/CLOCK
#define NBIT            65      // number of bit to receive -1

byte B[9];                      // receive buffer
static byte  RFstate;           // receiver state
static sbyte RFcount;           // timer counter
static byte  Bptr;              // receive buffer pointer
static byte  BitCount;          // received bits counter
word   XTMR;                    // 16 bit extended timer
volatile bit RFFull;            // buffer full
volatile bit RFBit;             // sampled RF signal
#define TRFreset    0
#define TRFSYNC     1
#define TRFUNO      2
#define TRFZERO     3
#define HIGH_TO     -10         // longest high Te
#define LOW_TO       10         // longest low  Te
#define SHORT_HEAD   20         // shortest Thead accepted 2,7ms
#define LONG_HEAD    45         // longest Thead accepted 6,2ms

#pragma int_rtcc   // install as interrupt handler (comment for HiTech!)
interrupt
rxi()
{
    // this routine gets called every time TMR0 overflows
    RFBit = RFIn;               // sampling RF pin verify!!!
    TMR0 -= PERIOD;             // reload
    TOIF = 0;
    XTMR++;                     // extended 16 long timer update
    if (RFFull)                 // avoid overrun
        return;
    switch( RFstate)            // state machine main switch
    {
        case TRFUNO:
            if ( RFBit == 0)
                { // falling edge detected ----+
                    // |
                    // |
RFstate= TRFZERO;
}
else
{ // while high
RFcount--;                         // reset if too long
if ( RFcount < HIGH_TO)
RFstate = TRFreset;
}
break;

case TRFZERO:
if ( RFBit)
{ // rising edge detected
//                          +----
//                          |     |
//                          +----+
RFstate= TRFUNO;
B[Bptr] >>= 1;                    // rotate
if ( RFcount >= 0)
{
    B[Bptr] += 0x80;            // shift in bit
}
RFcount = 0;                      // reset length counter
if ( ( ++BitCount & 7) == 0)
Bptr++;                           // advance one byte
if (BitCount == NBIT)
{
    RFstate = TRFreset;      // finished receiving
    RFFull = TRUE;
}
}
else
{ // still low
RFcount++;                       // reset points, while keep counting on
if ( RFcount >= LOW_TO)         // too long low
{
    RFstate = TRFSYNC;        // fall back into RFSYNC state
    Bptr = 0;
    BitCount = 0;
}
}
break;

case TRFSYNC:
if ( RFBit)
{ // rising edge detected
//                          +----
//                          |     |
//                          +----+
if ( RFcount < SHORT_HEAD) || ( RFcount >= LONG_HEAD)
{
    RFstate = TRFreset;
    break;                   // too short/long, no header
}
else
{
    RFcount =0;              // restart counter
    RFstate= TRFUNO;
}
}
else
{ // still low
    RFcount++;              // reset if too long
}
break;
case TRFrset:
    default:
        RFstate = TRFSYNC; // reset state machine in all other cases
        RFcount = 0;
        Bptr = 0;
        BitCount = 0;
        break;
    } // switch

} // rxi

void InitReceiver()
{
    T0IF = 0;
    T0IE = 1; // TMR0 overflow interrupt
    GIR = 1; // enable interrupts
    RFstate = TRFreset; // reset state machine in all other cases
    RFFull = 0; // start with buffer empty
    XTMR = 0; // start extended timer
    
}
APPENDIX E: TABLE C SOURCE CODE

// *********************************************************************
//  Filename:   TABLE.c
// *********************************************************************
//  Author:     Lucio Di Jasio
//  Company:    Microchip Technology
//  Revision:   Rev 1.00
//  Date:       08/07/00
//
//  EEPROM TABLE Management routines
//     simple "linear list" management method
//
// Compiled using HiTech C compiler v.7.93
// Compiled using CCS    PIC C compiler v. 2.535
// ********************************************************************/
#define MAX_USER 8         // max number of TX that can be learned
#define EL_SIZE   8         // single record size in bytes

// Table structure definition:
//
// the EEPROM is filled with an array of MAX_USER user records
// starting at address 0000
// each record is EL_SIZE byte large and contains the following fields:
// EEPROM access is in 16 bit words for efficiency
//
// DatoHi  DatoLo  offset
// +-------+-------+          XP contains the function codes (buttons) used during learning
// | FCode | IDLo  | 0     and the top 4 bit of Serial Number
// +-------+-------+          IDHi IDMi IDLo contain the 24 lsb of the Serial Number
// | IDHi  | IDMi  | +2    sync counter
// +-------+-------+          HopHi HopLo | +4
// | HopHi | HopLo | +4    second copy of sync counter for integrity checking
// +-------+-------+          HopHi2 HopLo2 | +6
// +-------+-------+
//
// NOTE a function code of 0f0 (seed transmission) is considered
// invalid during learning and is used here to a mark location free
//
// FIND Routine
//
// search through the whole table the given a record whose ID match
//
// INPUT:
//  IDHi, IDMi, IDLo, serial number to search
//
// OUTPUT:
//  Ind address of record (if found)
//  EHop sync counter value
//  ETemp second copy of sync counter
// RETURN:               TRUE if matching record found
//
byte Find()
{
  byte Found;
  Found = FALSE;       // init to not found

  for (Ind=0; Ind < (EL_SIZE * MAX_USER); Ind+=EL_SIZE)
  {
    RDword( Ind);       // read first Word
    FCode = (Dato>>8);
  }

  // more code here...
}
// check if 1111xxxx
if ((FCode & 0xf0) == 0xf0)
    continue;  // empty

if (IDLo != (Dato & 0xff))
    continue;  // fails match

RDnext();  // read next word
if ( ( (Dato & 0xff) == IDMi) && ( (Dato>>8) == IDHi))
{
    Found = TRUE;   // match
    break;
}
}  // for

if (Found == TRUE)
{
    RDnext();  // read HopHi/Lo
    EHop = Dato;
    RDnext();  // read HopHi2/Lo2
    ETemp= Dato;
}

return Found;

// INSERT Routine

//search through the whole table for an empty space

//INPUT:
//  IDHi, IDMi, IDLo,   serial number to insert

//OUTPUT:
//  Ind                 address of empty record

//RETURN:
//  FALSE if no empty space found

byte Insert()
{
    for (Ind=0; Ind < (EL_SIZE * MAX_USER); Ind+=EL_SIZE)
    {
        RDword(Ind);  // read first Word
        FCode = (Dato>>8);  // check if 1111xxxx
        if ( (FCode & 0xf0) == 0xf0)
            return TRUE;  // insert point found
    }  // for

    return FALSE;  // could not find any empty slot
}  // Insert

// Function IDWrite

// store IDHi,Mi,Lo + XF at current address Ind

//INPUT:
//  Ind                 point to record + offset 0
//  IDHi, IDMi, IDLo    Serial Number
//  XF                  function code

byte IDWrite()
{
    if (!FLearn)
return FALSE;           // Guard statement: check if Learn ON

Dato = Buffer[7];
Dato = (Dato<<8) + IDLo;
WRword(Ind);                // write first word

Dato = IDHi;
Dato = (Dato<<8) + IDMi;
WRword(Ind+2);              // write second word

return TRUE;
} // IDWrite

//------------------------------------------------------------
//Function HopUpdate
//  update sync counter of user record at current location
//INPUT:
//  Ind     record + offset 0
//  Hop     current sync counter
//OUTPUT:
//  none
//byte HopUpdate()
{
  if (!FHopOK)
    return FALSE;           // Guard statement: check if Hop update

  Hop = ((word)HopHi<<8) + HopLo;
  Dato = Hop;
  WRword(Ind+4);              // write at offset +4
  Dato = Hop;
  WRword(Ind+6);              // back up copy at offset +6
  FHopOK = FALSE;             // for safety disable updating hopping code

  return TRUE;
} // HopUpdate

//------------------------------------------------------------
//Function ClearMem
//  mark all records free
//INPUT:
//OUTPUT:
//USES:
//byte ClearMem()
{
  for (Ind=0; Ind < (EL_SIZE * MAX_USER); Ind+=EL_SIZE)
    {
      Dato = 0xffff;
      WRword( Ind);
    }

  return TRUE;
} // ClearMem
APPENDIX F: MEM-87X C SOURCE CODE

// *********************************************************************
//  Filename:   mem-87x.c
// *********************************************************************
//  Author:     Lucio Di Jasio
//  Company:    Microchip Technology
//  Revision:   Rev 1.00
//  Date:       08/11/00
//  Internal EEPROM routines for PIC16F87X
//  Compiled using HiTech PIC C compiler v.7.93
//  Compiled using CCS    PIC C compiler v. 2.535
// *********************************************************************

void RDword(word Ind)
{
    Dato = EEPROM_READ( Ind);
    Dato += (word) EEPROM_READ( Ind+1) <<8;
}

void RDnext()
{
    // continue reading
    EEADR++;       // NOTE generate no carry
    Dato = ((RD=1), EEDATA);
    EEADR++;       
    Dato += ((RD=1), EEDATA)<<8;
}

void WRword(word Ind)
{
    EEPROM_WRITE( Ind, Dato); GIE = 1; // write and re-enable interrupt
    EEPROM_WRITE( Ind+1, Dato>>8); GIE = 1;
}
APPENDIX G: KEY GENERATION SOURCE CODE

// ------------------------------------------------------------------------
// LEGAL NOTICE
//
// The information contained in this document is proprietary and
// confidential information of Microchip Technology Inc. Therefore all
// parties are required to sign a non-disclosure agreement before
// receiving this document.
// ------------------------------------------------------------------------
//
// Keeloq Normal Key Generation and Decryption
// Compiled using CCS PIC C compiler v. 2.535
// Compiled using HITECH PIC C compiler v. 7.93
//
// version 1.00 08/07/2000 Lucio Di Jasio
//
// ------------------------------------------------------------------------

byte    DKEY[8];   // Decryption key
byte    SEED[4];   // seed value = serial number
word    NextHop;   // resync value for 2 Chance

#ifdef HITECH
#include "fastdech.c"   // for HITECH optimized version
#else
#include "fastdecc.c"   // for CCS optimized version
#endif

//void LoadManufCode();
{    
DKEY[0]=0xef;   // DKEY=0123456789ABCDEF
DKEY[1]=0xcd;
DKEY[2]=0xAB;
DKEY[3]=0x89;
DKEY[4]=0x67;
DKEY[5]=0x45;
DKEY[6]=0x23;
DKEY[7]=0x01;
}

//----------------------------------------------------------------------

// Key Generation routine
//
// Normal Learn algorithm
//
// INPUT: Serial Number (Buffer[4..7])
//         Manufacturer code
// OUTPUT: DKEY[0..7] computed decryption key
//
void NormalKeyGen()
{
byte    HOPtemp[4];    // HOP temp buffer
byte    SKEYtemp[4];   // temp decryption key

    // check if same Serial Number as last time while output active
    // it was stored in Seed
if (( SEED[0] != Buffer[4]) ||
    ( SEED[1] != Buffer[5]) ||
    ( SEED[2] != Buffer[6]) ||
    ( SEED[3] != (Buffer[7] & 0x0f)) ||

    }
(COut == 0))
{
    // no new KeyGen is needed
    memcpy(HOPtemp, Buffer, 4); // save hopping code to temp
    memcpy(SEED, &Buffer[4], 4); // make seed = Serial Number
    SEED[3] &= 0x0f;           // mask out function codes

    // compute LSB of decryption key first
    memcpy(Buffer, SEED, 4);  // get SEED in
    Buffer[3] |= 0x20;        // add constant 0x20
    LoadManufCode();
    Decrypt();
    memcpy(SKEYtemp, Buffer, 4); // save result for later

    // compute MSb of decryption key
    memcpy(Buffer, SEED, 4);  // get SEED in
    Buffer[3] |= 0x60;        // add constant 0x60
    LoadManufCode();
    Decrypt();
    memcpy(&DKEY[4], Buffer, 4); // move it into DKEY MSb
    memcpy(DKEY, SKEYtemp, 4); // add LSB

    // ready for Decrypt
    memcpy(Buffer, HOPtemp, 4); // restore hopping code
}

else // same Serial Number as last time...
{
    // just keep on using same Decryption Key
}

} // Normal KeyGen

//------------------------------------------------------------------------------
//
// Valid Decryption Check
//
// INPUT:  Serial Number (Buffer[4..7])
//         Hopping Code  (Buffer[0..3])
// OUTPUT: TRUE if discrimination bits == lsb Serial Number
//          and decrypted function code == plain text function code
// byte DecCHK()
{
    // verify discrimination bits
    if ( DisLo != IDLo) // compare low 8bit of Serial Number
        return FALSE;

    if ( ((Buffer[3] ^ IDMi) & 0x3) != 0) // compare 9th and 10th bit of SN
        return FALSE;

    // verify function code
    if ( ((Buffer[3] ^ Buffer[7]) & 0xf0) != 0)
        return FALSE;

    return TRUE;
} // DecCHK

//------------------------------------------------------------------------------
//
// Hopping Code Verification
//
// INPUT:  Hopping Code  (Buffer[0..3])
//         and previous value stored in EEPROM EHop
// OUTPUT: TRUE if hopping code is incrementing and inside a safe window (16)
//
byte ReqResync()
{
    F2Chance = TRUE;  // flag that a second (sequential) transmission
    NextHop = Hop+1;  // is needed to resynchronize receiver
    return FALSE;    // cannot accept for now
}

byte HopCHK()
{
    FHopOK = FALSE;  // Hopping Code is not verified yet
    FSame = FALSE;   // Hopping Code is not the same as previous

    // make it a 16 bit signed integer
    Hop = ((word)HopHi << 8) + HopLo;

    if ( F2Chance)
        if ( NextHop == Hop)
            {F2Chance = FALSE;       // resync success
             FHopOK = TRUE;
             return TRUE;
            }

    // verify EEPROM integrity
    if ( EHop != ETemp)  // memory corrupted need a resync
        return ReqResync();

    // main comparison
    ETemp = Hop - EHop;  // subtract last value from new one

    if ( ETemp < 0)   // locked region
        return FALSE;   // fail
    else if ( ETemp > 16)   // resync region
        return ReqResync();
    else  // 0>= ETemp >16 ; open window
        {
            if ( ETemp == 0)  // same code (ETemp == 0)
                FSame = TRUE;  // rise a flag

            FHopOK = TRUE;
            return TRUE;
        }
}  // HopCHK
“All rights reserved. Copyright © 2001, Microchip Technology Incorporated, USA. Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip’s products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights. The Microchip logo and name are registered trademarks of Microchip Technology Inc. in the U.S.A. and other countries. All rights reserved. All other trademarks mentioned herein are property of their respective companies. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.”

**Trademarks**

The Microchip name, logo, PIC, PICmicro, PICMASTER, PICSTART, PRO MATE, KEELoQ, SEEVAL, MPLAB and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

Total Endurance, ICSP, In-Circuit Serial Programming, FilterLab, MXDEV, microID, FlexROM, fuzzyLAB, MPASM, MPLINK, MPLIB, PICDEM, ICEPIC, Migratable Memory, FanSense, ECONOMONITOR, SelectMode and microPort are trademarks of Microchip Technology Incorporated in the U.S.A.

Serialized Quick Term Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2001, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999. The Company's quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, Keeloc® code hopping devices, Serial EEPROMs and microperipheral products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001 certified.
**Worldwide Sales and Service**

**AMERICAS**

Corporate Office
2335 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Rocky Mountain
2335 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-7456

Atlanta
500 Sugar Mill Road, Suite 200
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Austin
Analog Product Sales
8033 MoPac Expressway North
Suite A-201
Austin, TX 78759
Tel: 512-345-2030 Fax: 512-345-6085

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Boston
Analog Product Sales
Unit A-8-1 Milbrook Tarry Condominium
97 Lowell Road
Concord, MA 01742
Tel: 978-371-6400 Fax: 978-371-0050

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Dayton
Two Prestige Place, Suite 130
Miamisburg, OH 43342
Tel: 937-291-1654 Fax: 937-291-9175

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-283-1836 Fax: 949-263-1338

Mountain View
Analog Product Sales
1300 Terra Bella Avenue
Mountain View, CA 94043-1836
Tel: 650-966-8241 Fax: 650-967-1590

New York
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 631-273-5305 Fax: 631-273-5355

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

**ASIA/PACIFIC**

**Australia**

Microchip Technology Australia Pty Ltd
Suite 22, 41 Lawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

**China - Beijing**

Microchip Technology Beijing Office
Unit Unit 915
New China Hong Kong Manhattan Bldg.,
No. 6 Chaoyangmen Baidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Shanghai

Microchip Technology Shanghai Office
Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6273-5700 Fax: 86-21-6273-5060

Hong Kong

Microchip Asia Pacific
RM 2101, Tower 2, Metroplaza
222 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200 Fax: 852-2401-3431

India

Microchip Technology India Ltd.
India Liaison Office
Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnesssey Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax 91-80-2290062

Japan

Microchip Technology Intl. Inc.
Benes B-1 EF
3-18-20, Shinanokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471-6166 Fax: 81-45-471-6122

**ASIA/PACIFIC (continued)**

Korea

Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore

Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 189980
Tel: 65-334-8870 Fax: 65-334-8850

Taiwan

Microchip Technology Taiwan
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

**EUROPE**

Denmark

Microchip Technology Denmark ApS
Regus Business Centre
Lautrup høj T-3
Ballenrun DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France

Arizona Microchip Technology SARL
Parc d’Activite du Moulin de Massy
43 Rue du Saule Trapu
Bdiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany

Arizona Microchip Technology GmbH
Gustav-Heinemann Ring 125
D-81739 Munich, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Greece

Analogy Product Sales
Locshamer Strasse 13
D-82152 Martinsried, Germany
Tel: 49-89-955650-0 Fax: 49-89-955650-22

Italy

Arizona Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899883

United Kingdom

Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44 118 921 5820

01/30/01

Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, except as maybe explicitly expressed herein, under any intellectual property rights. The Microchip logo and name are registered trademarks of Microchip Technology Inc. in the U.S.A. and other countries. All rights reserved. All other trademarks mentioned herein are the property of their respective companies.