Ref No. AN24-03

Key Scanning and LED Driving with the TMP86P202/203

Used Features:

- ✓ Key matrix scanning (rows and columns)
  OR
- ✓ Key detection by ADC input
- ✓ LED 7 segment driving
- ✓ Tone generation
- ✓ Time base timer
- ✓ Divider output

Introduction

This application note describes techniques for key detection using a minimal number of I/O lines with the TMP86P202/203. One approach scans the keys arranged as a matrix of rows and columns by multiplexing lines which are also used for display driving. The second alternative approach detects key presses via a single ADC input.

It features as an example, a simple minute-minder timer application which takes input from 12 keys and drives a 2 by 7 segment display. It also illustrates tone generation for key re-assurance and alarm tone generation.

Application Note Category

- ☐ Software Algorithm
- ☐ MCU specific
- ☑ **System Solution**
- ☐ Basic Design Technique

Toshiba 8-bit Series

- ☐ TLCS - 870
- ☑ **TLCS - 870/C**
- ☐ TLCS - 870/X

TMP86P202P/M
TMP86P203P/M

By:
Toshiba Electronics Europe GmbH
European LSI Design Eng. Center - **ELDEC**
Support-MCU@tee.toshiba.de

Hardware Schematic – Key Matrix Example

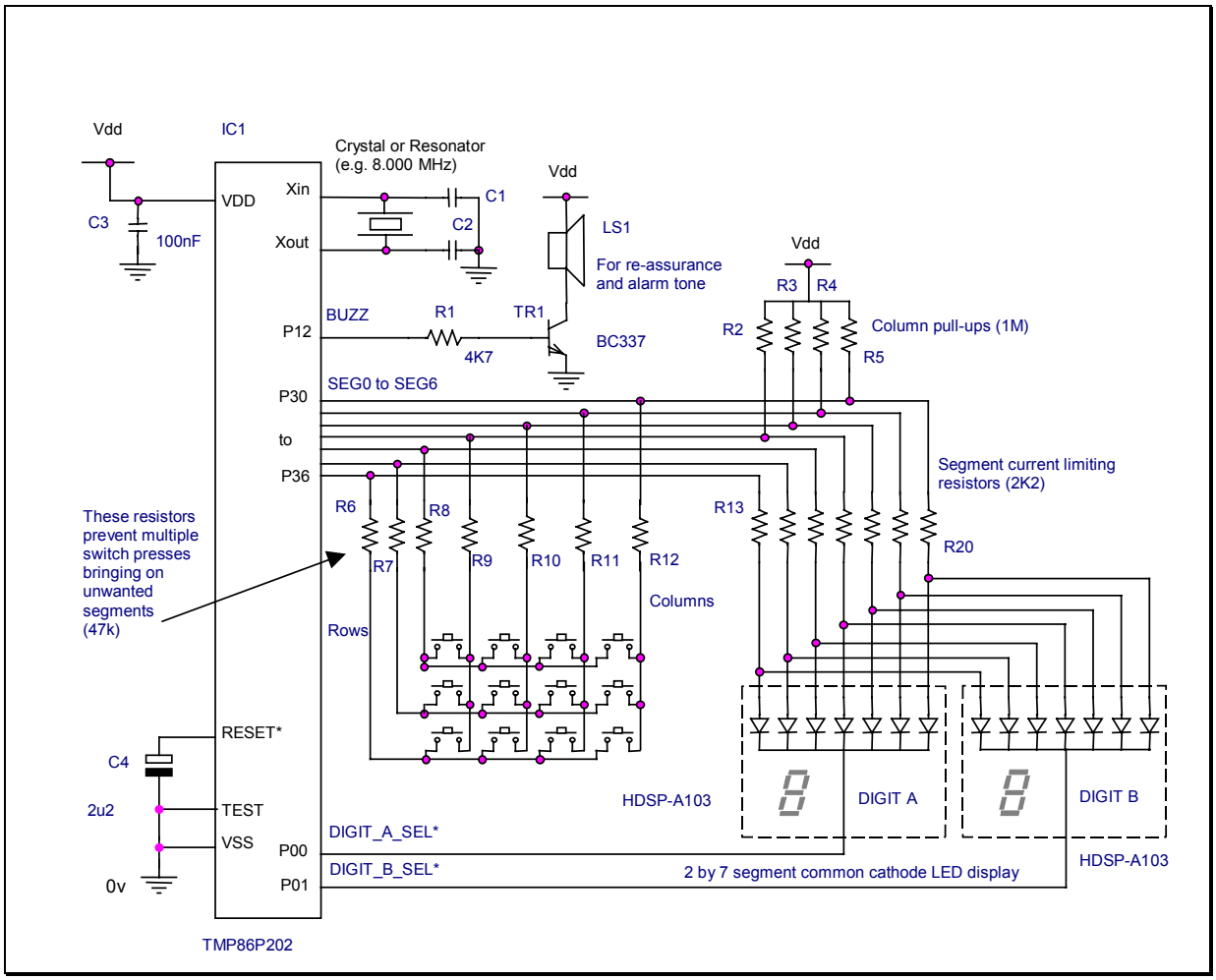


Figure 1 - The schematic of the sample timer using row and column scanning for key detection.

Hardware Description -– Row and Column Scanning Key Detection

PROCESSOR BASICS

The TMP86P202 processor is fed directly from the 5 volt supply and its internal oscillator circuitry is used with an 8 MHz crystal, giving a minimum processor instruction time of 0.5us. The internal reset Schmitt input with built in pull-up means that an external capacitor to ground is all that is required to generate a delayed reset signal at power up.

TONE SIGNALS

The DVO 'Divider Output' of the TMP86P202 can be configured to generate a 50 % square wave at an appropriate fraction of the main clock frequency and this is therefore used to generate drive signals for a buzzer to give a key re-assurance beep. The low cost TR1 common emitter stage provides the necessary simple current amplification to allow the logic signal generated at P12 to drive the sounder LS1.

DISPLAY MULTIPLEXING

The design uses time multiplexing to allow both the displays to be driven from a single set of segment drive signals. The individual digits are selected by the software in turn by driving their common cathode low. Pulling one set of cathodes low like this means that current provided by the segment

drivers can only flow down through that digit. This means we can share the segment drives for all the digits and this keeps the pin count down.

These digit select outputs have to sink all the current flowing through all the segments in their corresponding digit. Port 0 of the TMP86P202 has a high current capability which is sufficient to sink the sum of segment currents for a high efficiency such as the HDSP-A103. Lines P00 and P01 are therefore each used for direct digit strobes, saving the costs and board area of additional drivers. P30 to P36 are standard port pins and these are used to drive the individual segments directly via R13 to R20. The LEDs which form the individual segments of the displays have a forward voltage drop of 1.6 volts at a forward current of 1mA. With a five volt supply and 1.6 volt drop the maximum voltage across the segment limiting resistors will be 3.4 volts. Choosing 2k2 resistors here limits the maximum current to 1.55mA., below the quoted maximum 1.8 mA source current for TMP86P202 standard pins. Note that all the segment currents sum to form the current that the digit outputs must sink. When all 7 segments are on this gives a maximum current into the digit outputs of 1.55 * 7 or 10.8mA, well below the 30 mA quoted maximum for these high current pins. In practice small voltage drops in the segment and digit driving outputs will reduce these currents a little.

NOTE ON USE OF HIGHER CURRENT DISPLAYS

Higher current displays can be easily driven using simple transistor buffers to provide additional segment and digit currents. These can be high gain bipolar or FETs with low Vgs (threshold) to ensure that can be driven hard into conduction by the logic level outputs of the TMP86P202/203.

KEYPAD INTERFACING

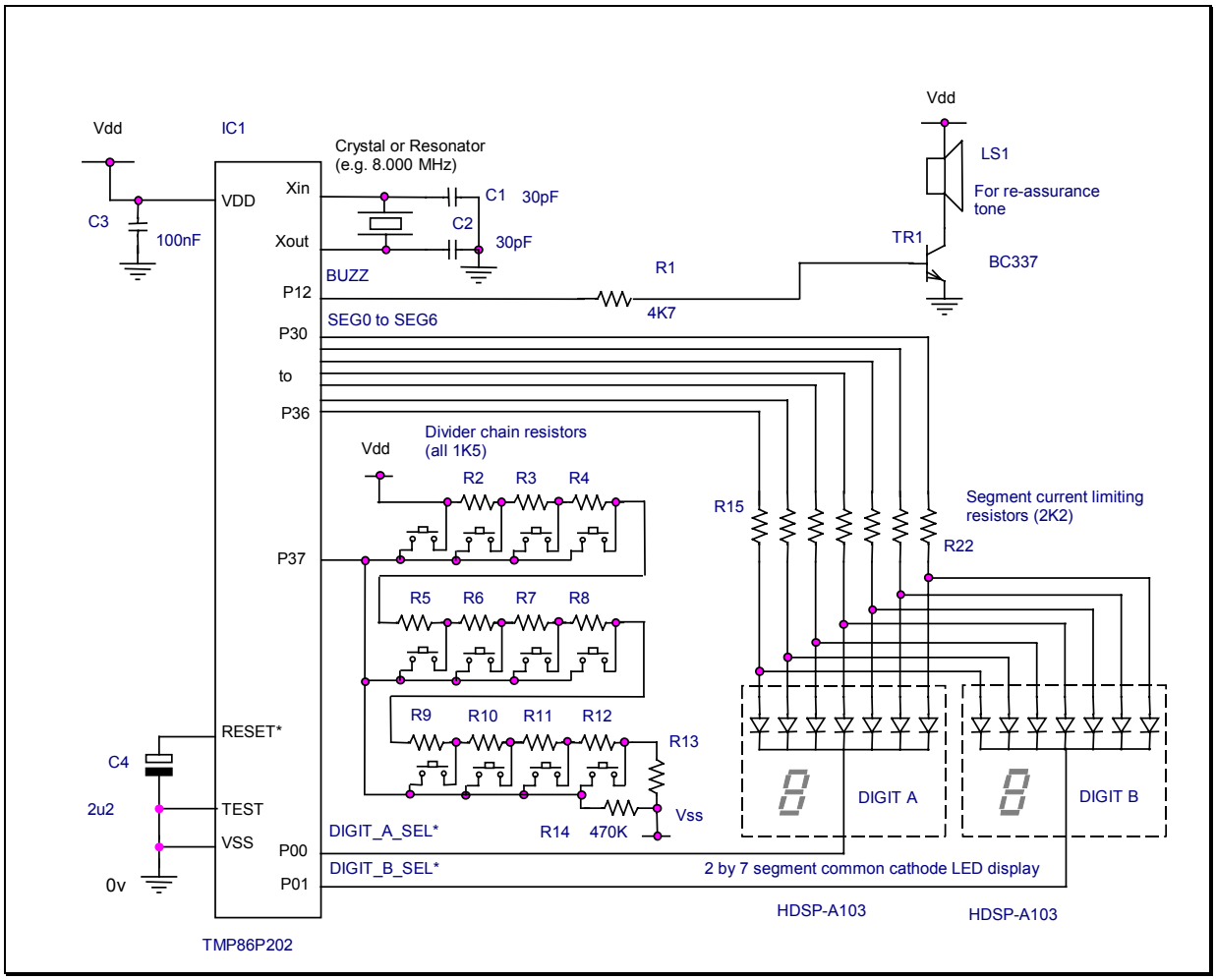
When keyboard scanning both the two digit drive outputs are set to the off state and consequently no current can flow through the display segments. In this condition the segment drive lines can therefore be re-used to scan the keyboard. The four port lines P30 to P33 are used as column inputs and weak pull-up resistors R2 to R5 bias these lines high. The three port lines P34 to P36 are used as row drives. The software drives these hard low in turn, reading the column inputs each time as it goes. When any one of the 12 switches is pressed it will connect one particular row to one particular column. It follows that where a switch is pressed the hard low on the row will overcome the weak pull-up causing that column input to read as low.

For keyboard scanning only, R6 to R12 are not strictly necessary. However because we are re-using display driving segment lines any made switches will have the effect of connecting one segment drive line to another, causing unwanted display segments to illuminate during display multiplexing.
The choice of value for these resistors is therefore a compromise – they need to be high enough to keep the unwanted segment currents when display driving acceptably low whilst still allowing enough current to flow when key scanning to overcome the weak pull-ups on the column lines.

EMC/ESD CONSIDERATIONS

These series resistors R6 to R12 also perform a valuable EMC function – commonly used membrane and other similar keypad arrangements provide a point of entry for static discharges, and these resistors therefore help limit the amount of energy from ESD discharges to the keypad that can reach the processor.

Hardware Schematic – ADC Based Key Detection



Hardware Description – ADC Based Key Detection

Apart from the key detection scheme, the remainder of the hardware is as for the key matrix implementation described previously.

Key detection via the ADC is made possible using a divider chain from Vdd to ground (R2 to R13). This chain is tapped at 12 points and operating any one key will therefore cause the potential at its associated tap to be presented to the ADC input (port 37). R14 ensures that the ADC input is held close to ground when no keys are pressed. It should be noted that this approach cannot readily be used to detect multiple simultaneous key presses.

The source impedance of the ADC input needs to be kept low to prevent voltage loss due to the charging of the internal sample and hold capacitance of the ADC. The worst case will be for the tap at the centre of the chain. This has 6 1k5 resistors to Vcc and 6 to ground. The source impedance will therefore be that of two 9k resistances in parallel or 4k5. This is lower than the 5k maximum source impedance recommended for the device.

Software Description

The source code for the minute minder example is given at the end of this application note. This includes a single '#define' compile time switch 'KEYS_VIA_ADC'. Defining this identifier will cause the code to be built for ADC based key detection, leaving it undefined will invoke the key matrix scanning instead.

OVERALL STRUCTURE

There is a simple conventional main loop, supported by a single fast heart beat interrupt.

MAIN LOOP

The main function calls an initialisation routines to set up the ports and the time base timer, enables the time base timer interrupt and then loops continually.

This main loop carries out three actions in turn :-

Action	Explanation/ Remarks
Check for key presses detected by the interrupt handler	Whenever the interrupt handler detects a new key press it sets gucIntKeyScanCode to a non zero value which represents the position of the key in the key matrix. When this is detected by the main line code the key scan code is converted via a table look-up process into a key 'value' which, for clarity is selected to be an 'ASCII' value. The converted value is then tested and the appropriate function carried out.
Handle elapsed time ticks recorded by the interrupt handler.	gucIntTicks is incremented in every time base timer interrupt. This section of main line code copies this to its own count of elapsed ticks and clears gucIntTicks. The elapsed tick count is then used to decrement timers (in this case there is only one used for tone duration timing). It is also used to account for the passage of real time with regard to the minute minder counts themselves. The scaling uses as accumulator based method described in more detail in a section below.
Translate the display digits ready for display whenever they may have changed.	A flag gucDisplayUpdate is used to indicate whenever the display contents may have altered. Setting this flag causes this code section to re-translate the digits held in the main minutes variables to corresponding segment patterns for display. Doing the translation here in the main line code keeps the interrupt code to a minimum which is always a sound objective. The flag means that the translation is only invoked when necessary, keeping main loop time short.

INTERRUPT HANDLER AND INTERRUPT SUB-JOBS

On each invocation the handler for the time base timer interrupt carries out one of 3 different sub-jobs. The first two such sub-jobs drive the display segments and the third looks for key presses.

DISPLAY DRIVING SUB JOBS

The display driving sub-jobs make the appropriate digit select line hard low and then apply the segment conditions for that digit. These conditions then remain until the next interrupt.

KEY SCANNING SUB JOB - ROW AND COLUMN SCANNING

For this approach, the key scan logic can be represented simply as follows :-

> *For each row in turn, drive the column hard low and set its neighbours as inputs, and read the consequent values on the 4 column port lines. If any of these are low a key switch must be in the 'pushed' state.*

As the row and column scanning proceeds a 'scan code' is incremented. This is effectively an index into the key matrix positions. When a key is detected a check is made to see if this a new condition or if it was also detected in the previous scan. This allows the logic to report only new key presses. This check simply checks the scan code detected against the stored scan code resulting from the previous scan. The key scanning logic is suppressed for a defined number of interrupts after a new press is detected in order to prevent key bounces causing spurious key presses. The scanning is also suppressed if a previously detected key press has yet to be processed by the main line code – the main line code clears gucIntKeyScanCode when it processes each key press.

KEY SCANNING SUB JOB – ADC BASED KEY DETECTION

This method relies on the individual switches each connecting the ADC input to a distinct tap on a potential divider chain. It follows that any one key press will set a particular input voltage at the ADC input. ADC values can therefore be converted to the equivalent key. Note that the software is coded so as to require three consecutive ADC reads to indicate the same key before it will be treated as valid. This is to prevent instantaneous values read during the on or off transitions from one key being erroneously treated as another.

SOFTWARE FEATURES

This section describes some of the software techniques that have been employed in the application that may be useful for designers of other or similar applications.

ACCUMULATOR BASED TIME ACCOUNTING AND SCALING

The minute minder requires an accurate 'seconds' event to be detected in order correctly to maintain the minutes variables which are displayed.

However, The time base timer period will in general not be an exact sub-multiple of a second, so scaling is required to detect the passage of seconds accurately. The technique that is used here is a general, and replies on simple arithmetic only, whilst still guaranteeing that no time is lost uring rounding etc.

The principle is to use a simple accumulator. Each interrupt tick detected by the main line code adds in an appropriate number of 'time units' to the accumulator and after each such addition, a test is done to see if the number of 'time units ' accrued in the accumulator has reached the equivalent of a second. When this level is reached, the equivalent of an exact seconds worth of units is subtracted from the accumulator, leaving any excess to form the start of the next second. The only critical part is to select an appropriate time unit whole numbers of which can represent the exact time corresponding to an interrupt period and a complete second.

In this application the main CPU clock is 8 MHz and the time base timer interrupt is set to divide by 2**14 so the period is 0.125 microseconds * 16384 or 2048 microseconds. If the accumulator were to be scaled in microseconds we could therefore add in 2048 for each detected interrupt tick and subtract 1000000 for each second. In fact in this application we are fortunate and we can divide by

the highest common factor of 2048 and 1000000 which is 64. We therefore add in 32 (i.e. 2048/64) on every interrupt and subtract 15625 (i.e. 1000000/64) every second. The has the added benefit that the accumulator need only be 16 bits in length as it needs only accommodate values slightly in excess of the 15625 seconds limit.

CONVERTING ADC READINGS TO KEY SCAN CODES

The nominal ADC reading for a key at tap 'n' will be given by :-

ADC reading = 255 * n/12

Thus for example, pressing the key at the bottom right in the schematic connecting the ADC input to the junction of R12 and R13 would give a nominal expected ADC reading of :-

ADC reading = 255 * 1/12 = 21.25 (rounded to 21)

For a 12 tap chain, the conversion to key position is effectively to divide by the number of ADC steps per tap (255/12) or 21.25. In practice it is faster to multiply than to divide, and we also need to avoid truncating values that are fractionally under the nominal positions.

The method used in the software is therefore based on :-

Key position (scan code) = ((ADC reading + 10) * 12)/256

This can be done quickly within simple 16 bit arithmetic by shifting and adding.

Note that R14 will reduce the nominal levels fractionally and the proportional reduction will be worst for the centre tap (because the effective source impedance is highest here). In fact the proportional loss at any tap 'n' can be approximated as :-

Resistance of one chain element (e.g R13)*(12n-n*n)/(12 *Pull down (i.e R14)

So for the centre element n=6, so assuming the chain elements are each 1k5 and the pull down resistor R14 is 470K we have a proportional loss of :-

Proportional loss = 1k5 *(72-36)/(12 * 470k)
= 0.0096 or 0.96 %

In practice this will be of the same order as the errors induced by tolerance errors in the resistances. The difference in expected ADC readings voltage between taps are around 1/12 or 8 % so these errors are acceptable. We can however compensate for this a little by simply increasing the constant used for rounding to 12 from 10, so that the final transformation used is :-

Key position (scan code) = ((ADC reading + 12) * 12)/256

MULTIPLEXING KEY SCANNING AND LED DRIVING

Multiplexing the two display digits helps keep pin count down as the segment lines can drive both digits. An interrupt driving strobing technique can provide a simple basis for this with alternate digits being driven during alternate interrupt periods. However if we add a third scan period to the basic LED signal frame but don't activate either of the digit drivers we can use the segment lines for key scanning without bringing on the LEDs. However any keys that are made will connect two of the segment lines together and this will have the consequence that when one of these connected segments is activated the drive current will flow through the switch to its partner, bringing on that possibly unwanted segment. This phenomenon can be removed by reducing the current that can flow through the switches by means of series resistors.

The resultant multiplexing of display and key scanning, used in this application is illustrated in Figure 2.

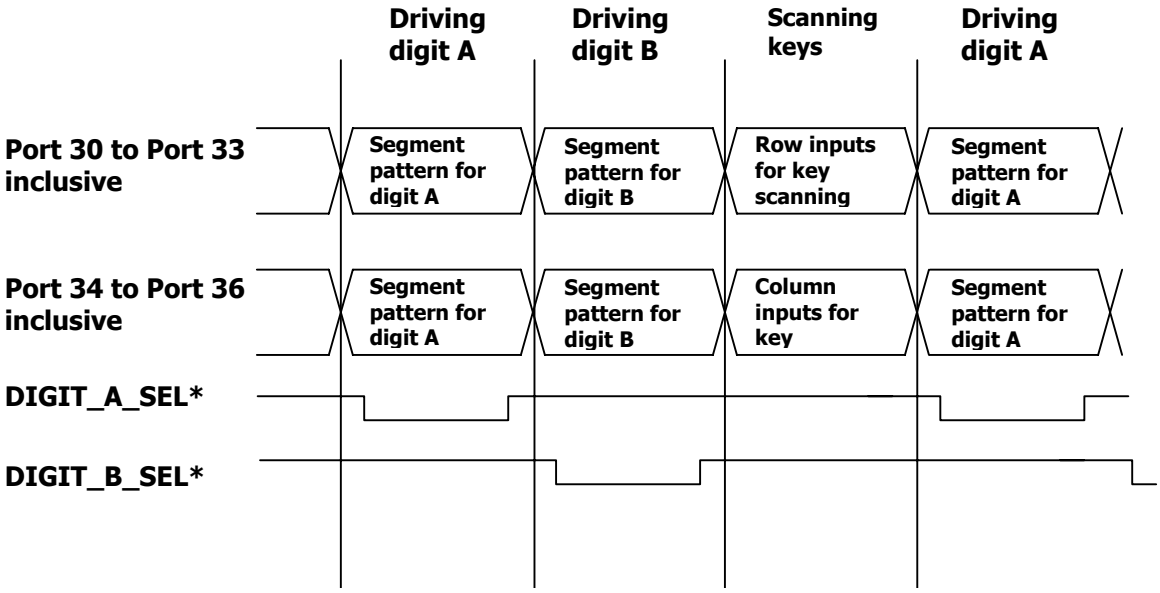


Figure 2 – Pin sharing for display driving and key scanning

Software Source Code

```
// ************************************************
// ************************************************
// ************************************************
// File        : MinuteMindermain.c
// Authors     : John Thorn
//              AND Technology Research Ltd.
//               4 Forest Drive,
//               Theydon Bois
//               Essex, CM16 7EY
//               Tel +44 (0) 1992 81 4655
//               Fax +44 (0) 1992 81 3362
//               e-mail : john.thorn@andtr.com
//               www : www.andtr.com
//
// This file contains the source code for the main function
// of the control for the Minute Minder Application Note
// for the TMP86P202/203 device.
//
// It is intended as valid input to the Toshiba
// C compiler/ tool chain.
//
// It is targeted at an TMP86P202
//
// ************************************************
// ************************************************
// TO DO LIST
// ----------
//
// ************************************************
// ************************************************
// Version History
// --------------------
// Version     Date           By
// ----------------------------------
// 1.00        11/02/2003     JT,ATR
//                                          Original coding completed.
// ----------------------------------
// ----------------------------------
// ************************************************
// FUNCTIONAL OVERVIEW
// -------------------
//
// This code is the sample code supporting an application
// note for the TMP86P202/203 family of devices.
// It illustrates the use of :-
// 7 segment display driving
// Key matrix scanning using I/O lines shared with
// the display driving
// Key scanning via ADC input and tapped divider chain
// Divider output for tone generation
// Timebase timer for timebase operation
//
//
// FILE/ MODULE STRUCTURE
// ----------------------
//
// There is a single main file only (this file).
// There is also cstartup.asm.
//
// BASIC SOFTWARE STRUCTURE
// ------------------------
//
// A simple main loop is used.
//
//
// INTERRUPT USAGE AND SYSTEM TIMING
// ---------------------------------
//
// A single timing interrupt using the time base timer to give a
// fast heartbeat timer at a nominal 10 milliseconds.
//
// INTERRUPT ENABLING AND DISABLING
// --------------------------------
//
// Apart from minor atomic sections global interrupts remain
```

```
// enabled the whole time.
//
// ADC USAGE
// ---------
//
// None in this application
//
// CLOCK DETAILING
// ---------------
//
// The CPU is assumed to run at 8 MHz.
//
// ************************************************
// MEMORY LAYOUT NOTES
// -------------------
//
// The processor is used in single chip mode only, so there
// are no external busses.
// ************************************************
// ************************************************
// ************************************************
// C O M P I L E R    D I R E C T I V E S
// ************************************************
// ************************************************
// Comment this out to get key scanning by rows and columns
// leave it uncommented to get ADC based key detection
//#define KEYS_VIA_ADC (1)
// This is used to select the key table for some
// prototype hardware
#define PROT_HARDWARE  (1)
// ************************************************
// ************************************************
// I N C L U D E    F I L E S
// ************************************************
// ************************************************
// These are the processor specific definitions etc.
#include "io86xx02.h"
// ************************************************
// ************************************************
// C O N S T A N T S    A N D   D E F I N E S
// ************************************************
// ************************************************
// ************************************************
// ************************************************
// G E N E R I C   D E F I N E S
// ************************************************
// ************************************************
#define BIT_0                   (0x01)
#define BIT_1                   (0x02)
#define BIT_2                   (0x04)
#define BIT_3                   (0x08)
#define BIT_4                   (0x10)
#define BIT_5                   (0x20)
#define BIT_6                   (0x40)
#define BIT_7                   (0x80)

#define HIGH                    1
#define LOW                             0

#define TRUE                    1
#define FALSE                   0

// ************************************************
// A P P L I C A T I O N     D E F I N E S
// ************************************************
// ************************************************
// These are hardware specific definitions for this application
// These two port bits are the
#define DIGIT_A_ENABLE _p0dr_.bit.b0
#define DIGIT_B_ENABLE _p0dr_.bit.b1

#ifndef KEYS_VIA_ADC
//************************************************
// Defines for the row and column scanning key detection scheme
//************************************************
#define COLUMN_PORT             _p3dr_.byte
#define C_COLUMN_MASK  (BIT_3|BIT_2|BIT_1|BIT_0)
```

```
#define C_COLUMN_0            (BIT_0)
#define ROW_PORT              _p3dr_.byte
#define C_ROW_MASK            (BIT_6|BIT_5|BIT_4)
#define C_ROW_0                      (BIT_4)
#define C_NUMBER_OF_ROWS      (3)
#define C_NUMBER_OF_COLUMNS   (4)
#define C_NUMBER_OF_KEYS      (C_NUMBER_OF_ROWS*C_NUMBER_OF_COLUMNS)

#define C_COLUMN_SETTLING_DELAY       (50)
#else
//*************************************************
// Defines for the row and column scanning key detection scheme
//*************************************************
#define C_NUMBER_OF_KEYS              (12)
// This actually 255/(2*number of keys) plus an adjustment to compensate
// the effect of the pull- down
#define C_ROUNDING_ADJUSTMENT (12)
// How many ADC reads of a given key before we see it as valid?
#define C_NUMBER_OF_CONS_ADC_READS    (4)
// How much to wait between ADC reads
#define C_DELAY_BETWEEN_ADC_READS     (20)
#define ADCCR_START_CONV_AIN5 (0xa5)
#endif

#define SEGMENT_PORT   _p3dr_.byte

#define C_DEBOUNCE_KEY_SCAN_PERIODS   (5)

#define C_TICKS_FOR_APPROX_5_SECONDS (2443)
#define C_TICKS_FOR_TONE_BEEP (25)

// for time base interrupt at 8000000/2**14
// interrupt period =0.125 * 16384
// = 2048 microseconds
// we need to add in equiv of 2048 microsecs every interrupt
// and then remove 10000000 every second
// to work in 16 bits and keep the numbers down
// we can cancel these to the simplest fraction
// 2048/1000000 - cancel 64s = 32/15625
// at 8000000
#define C_TICKS_PER_INTERRUPT (32)
#define C_TICKS_PER_SECOND    (15625)

// *************************************************
// *************************************************
// D A T A   D E F I N I T I O N S
//
// These include some associated constant defines
// also. Most of these are kept near the associated
// RAM data.
// *************************************************
// *************************************************
static volatile unsigned char __tiny gucIntKeyScanCode;
static volatile unsigned char __tiny gucIntTicks;


static unsigned char  __tiny gucInterruptSubJob;
static unsigned char  __tiny gucElapsedTicks;
static unsigned char  __tiny gucDebounceTimer;
static unsigned char  __tiny gucSeconds;
static unsigned char  __tiny gucMinutesMsb;
static unsigned char  __tiny gucMinutesLsb;
static unsigned char  __tiny gucDisplayMsbSegments;
static unsigned char  __tiny gucDisplayLsbSegments;
static unsigned char  __tiny gucDisplayUpdate;
static unsigned char  __tiny gucCounting;
static unsigned char  __tiny gucKeyValue;
static int                   __tiny gnTickAccumulator;
static int                   __tiny gnToneTimer;

#ifdef PROT_HARDWARE
const unsigned char KeyTransTable[C_NUMBER_OF_KEYS+1] =
       {  0,                                                  // dummy for scan code 0
         '4', '5', '6', '7',
         '0', '1', '2', '3',
         '8', '9', 'C', '+'};
#else
```

```
const unsigned char KeyTransTable[C_NUMBER_OF_KEYS+1] =
        {  0,                                                   // dummy for scan code 0
          '0', '1', '2',
          '3', '4', '5',
          '6', '7', '8',
          'C', '9', '+'};
#endif
/* define the character literals and the character translation table
for displaying - for this application we need only show
ordinary digits and a dash so we can set the values up so that
simple binary values can be used for the digits*/
enum {
        SHOW_0=0,
        SHOW_1,
        SHOW_2,
        SHOW_3,
        SHOW_4,
        SHOW_5,
        SHOW_6,
        SHOW_7,
        SHOW_8,
        SHOW_9,
        SHOW_DASH
};
#define SEGNONE 0x00
#define SEGA 0x01
#define SEGB 0x02
#define SEGC 0x04
#define SEGD 0x08
#define SEGE 0x10
#define SEGF 0x20
#define SEGG 0x40

#define C_SEGMENT_MASK (SEGA+SEGB+SEGC+SEGD+SEGE+SEGF+SEGG)

#define SIZE_OF_DISPLAY_CHAR_TABLE    (13)

const unsigned char DisplayCharToSegTable[C_NUMBER_OF_KEYS+1] =

{
        SEGA+SEGB+SEGC+SEGD+SEGE+SEGF,                  // 0x00 0
        SEGB+SEGC,                                                      // 0x01 1
        SEGA+SEGB+SEGD+SEGE+SEGG,                       // 0x02 2
        SEGA+SEGB+SEGC+SEGD+SEGG,                       // 0x03 3
        SEGB+SEGC+SEGF+SEGG,                            // 0x04 4
        SEGA+SEGC+SEGD+SEGF+SEGG,                       // 0x05 5
        SEGA+SEGC+SEGD+SEGE+SEGF+SEGG,                  // 0x06 6
        SEGA+SEGB+SEGC,                                                 // 0x07 7
        SEGA+SEGB+SEGC+SEGD+SEGE+SEGF+SEGG,   // 0x08 8
        SEGA+SEGB+SEGC+SEGD+SEGF+SEGG,                  // 0x09 9
        SEGG                                                            // 0x0a -
};

// ************************************************
// ************************************************
// F U N C T I O N   P R O T O T Y P E S
// ************************************************
// ************************************************

extern void startup(void);              // startup.asm

static void __interrupt IntDummy(void);
static void __interrupt_n IntDummyN(void);
static void __interrupt IntTBT(void);

static void Initialisation(void);

// ************************************************
// ************************************************
// C O D E
// ************************************************
// ************************************************

#pragma section code

/*********************************************************************
NAME:           static void SetTone(unsigned char ucOn)
```

```
FUNCTION:       Turns the tone (ie the DVO output) on or off.
PARAMETERS: ucOn - TRUE for tone on, FALSE for tone off
RETURN:         None
NOTES:          A simple wrapper to encapsulate hardware
                        details.
*****************************************************************/
static void SetTone(unsigned char ucOn)
{
        unsigned char ucNewTBTCR;
        ucNewTBTCR=TBTCR;
        if (ucOn)
        {
                ucNewTBTCR &=0x0f;
                ucNewTBTCR |=(BIT_7 | BIT_5); // enable DVO and set 1.953 KHz
                P1DR=BIT_2;
                P1CR=BIT_2;
        } else
        {
                ucNewTBTCR &=0x0f;
                P1DR=0;
                P1CR=0;
        }
        TBTCR=ucNewTBTCR;
}
/*****************************************************************
NAME:           static unsigned char GetKeyValue(unsigned char ucKeyScanCode)
FUNCTION:       Converts key scan codes to key values via a simple
                        look-up function.
PARAMETERS: The scan code
RETURN:         The key value
NOTES:          A simple wrapper to encapsulate hardware
                        details.
*****************************************************************/
static unsigned char GetKeyValue(unsigned char ucKeyScanCode)
{
        unsigned char ucReturn;
        ucReturn = KeyTransTable[ucKeyScanCode];
        return ucReturn;
}
/*****************************************************************
NAME:           unsigned char GetADCKeyScanCode()
FUNCTION:       Gets an ADC reading and converts it to a key scan code
PARAMETERS: None
RETURN:         The scan code
NOTES:          Zero if no keys pressed, else scan code value
                        from 1 to 12.
                        To be valid, same key code has to be seen
                        C_NUMBER_OF_CONSECUTIVE_ADC_READS times.
*****************************************************************/
#ifdef KEYS_VIA_ADC
unsigned char GetADCKeyScanCode()
{
        unsigned char volatile ucDelay;
        unsigned char ucCount;
        unsigned int unFirstADCScanCode;
        unsigned int unADCScanCode;

        ucCount=1;
        ADCCR1 = ADCCR_START_CONV_AIN5;       // start conversion
        while (!(ADCDR2 & 0x20));                     // then wait for it to finish
        unFirstADCScanCode=ADCDR1;
        unFirstADCScanCode+=C_ROUNDING_ADJUSTMENT;
        unFirstADCScanCode*=C_NUMBER_OF_KEYS; //  could  replace  this  by  add  and  shift  as
optimisation if necessary
        unFirstADCScanCode>>=8;                                                 // divide by 256
        while (ucCount<C_NUMBER_OF_CONS_ADC_READS)
        {
                // wait for voltage to move some more if it's moving
                for (ucDelay=0;ucDelay<C_DELAY_BETWEEN_ADC_READS;ucDelay++);
                // Now get new ADC value
                ADCCR1 = ADCCR_START_CONV_AIN5;   // start conversion
                while (!(ADCDR2 & 0x20));                   // then wait for it to finish
                unADCScanCode=ADCDR1;
                unADCScanCode+=C_ROUNDING_ADJUSTMENT;
                unADCScanCode*=C_NUMBER_OF_KEYS;            // could replace this by add and
shift as optimisation if necessary
                unADCScanCode>>=8;                                              // divide by 256
```

```
                if (unFirstADCScanCode!=unADCScanCode)
                {
                        return 0;
                }
                ucCount++;
        }
        // To get here we had a consecutiv number of valid reads
        return (unsigned char)(unFirstADCScanCode);
}
#endif
/*********************************************************************
NAME:           static unsigned char GetSegmentPattern(unsigned char ucByte)
FUNCTION:       Converts digits to the appropriate segment pattern for
                        the display, using a simple look-up table.
PARAMETERS: The digit value to convert
RETURN:         The segment pattern
NOTES:          A simple wrapper to encapsulate hardware
                        details.
                        Assumes that segment 'a' is bit 0,
                        segment 'b' is bit 1 etc.
*********************************************************************/
static unsigned char GetSegmentPattern(unsigned char ucByte)
{
        unsigned char ucReturn;
        ucReturn=DisplayCharToSegTable[ucByte];
        return ucReturn;
}
/*********************************************************************
NAME:           static void Initialisation(void)
FUNCTION:       Set up i/o control registers, intial state variables etc.
PARAMETERS: None
RETURN:         None
NOTES:          Only non-zero variable initialisation is required
                        as startup.asm zaps the whole RAM to zero.
*********************************************************************/
static void Initialisation(void)
{
        // Digit driving ports to initial state
        DIGIT_A_ENABLE = HIGH;
        DIGIT_B_ENABLE = HIGH;
        P0OUTCR                 = 0x03; // push-pull outputs

        //
        P3CR                    = 0x00; // start as all inputs
        P3DR                    = 0x00; // start as all low

        P1DR=0;
        P1CR=0;

        // Time base timer - initialise to close to 488.28 Hz
        // Ensure it's disabled
        TBTCR &=~(BIT_3);
        // Select clock and start the timer
        TBTCR=(BIT_3+3);        // fc/2**14 and go

#ifdef KEYS_VIA_ADC
        ADCCR1=0x25;    // software start mode, don't start yet, AIN5
        ADCCR2=0x16;    // ack= 156/fc= 19.5 us
#endif
        gucMinutesMsb = SHOW_DASH;
        gucMinutesLsb = SHOW_DASH;
        gucCounting = FALSE;
        gucDisplayUpdate = TRUE;
}
/*********************************************************************
NAME:           void main(void)
FUNCTION:       The main function.
                        This initialises the hardware before entering the main
                        loop, which is then repeated indefinitely.
                        The loop checks for new key presses and
                        updates timers derived from the time base timer
                        interrupt.
PARAMETERS: None
RETURN:         None
NOTES:
*********************************************************************/
void main(void)
```

```
{
	// Set up the hardware and any no-zero variable initialisation
	Initialisation();
	EIRL |= BIT_6; // now we've initialised we can safely enable the time base timer
interrupts

	while (1)		// so we don't exit out of main
	{
		// ******************************
		// STAGE 1 - KEY PROCESSING
		// Handle new key scan codes detected by the interrupt
		// handler
		// ******************************
		if (gucIntKeyScanCode)
		{
			// new key press detected, make a key beep
			SetTone(TRUE);
			gnToneTimer=C_TICKS_FOR_TONE_BEEP;
			// translate the raw scan code to the value of the key..
			gucKeyValue=GetKeyValue(gucIntKeyScanCode);
			// Tell the interrupt key scanning code we've seen this key press..
			gucIntKeyScanCode=0;
			// If he presses the 'Clear' key
			// we stop counting and force dashes into the display
			// ********************************
			// HANDLING OF 'C' CLEAR KEY
			// ********************************
			if (gucKeyValue=='C')
			{
				gucMinutesMsb=SHOW_DASH;
				gucMinutesLsb=SHOW_DASH;
				gucCounting=FALSE;
				gucDisplayUpdate=TRUE;
				// if he presses '+' we just inc the time to go
				// and start counting
			} else if (gucKeyValue=='+')
			// ********************************
			// HANDLING OF '+' INCREMENT KEY
			// ********************************
			{
				gucCounting=TRUE;
				gnTickAccumulator=0;
				if ((gucMinutesLsb!=SHOW_DASH) && (gucMinutesMsb!=SHOW_DASH))
				{
					gucMinutesLsb++;
					if (gucMinutesLsb>9)
					{
						gucMinutesLsb=0;
						if (gucMinutesMsb<9)
						{
						 gucMinutesMsb++;
						}
					}
				} else
				{
					gucMinutesLsb=1;
					gucMinutesMsb=0;
				}
				gucDisplayUpdate=TRUE;
			} else
			{
				// ********************************
				// HANDLING OF NUMBER KEYS
				// ********************************
				// Here he's entered a number - feed this through the
				// display starting with the ls digit
				if (gucMinutesLsb==SHOW_DASH)
				{
					gucMinutesMsb=0;
				} else
				{
					gucMinutesMsb=gucMinutesLsb;
				}
				gucMinutesLsb=gucKeyValue-'0';
				if (gucMinutesLsb | gucMinutesMsb) // only count if he's set a
non zero value
				{
```

```
                        gnTickAccumulator=0;
                        gucCounting=TRUE;
                } else
                {
                        gucCounting=FALSE;
                }
                gucDisplayUpdate=TRUE;
        }
}
// ******************************
// STAGE 2 - TIME ACCOUNTING
// Handle new elapsed time ticks detected by the interrupt
// handler
// ******************************
// Now handle the time detected by the interrupt handler
// fetch the elapsed time counted by the interrupt handler
__asm(" di");
        gucElapsedTicks=gucIntTicks;
        gucIntTicks=0;
__asm(" ei");
// Handle the elapsed time..
if (gucElapsedTicks)
{
        // Update all the timeout timers
        // (currently only the tone timer)
        // ********************************
        // HANDLING OF TIMEOUTS
        // ********************************
        if (gnToneTimer)
        {
                gnToneTimer--;
                if (gnToneTimer==0)
                {
                        SetTone(FALSE);
                }
        }
        // Now update the minute minder time
        // ********************************
        // HANDLING OF MINUTE MINDER TIME
        // ********************************
        while (gucElapsedTicks)       // count down for each tick..
        {
                gucElapsedTicks--;
                gnTickAccumulator+=C_TICKS_PER_INTERRUPT;
                while (gnTickAccumulator>=C_TICKS_PER_SECOND)
                {
                        gnTickAccumulator-=C_TICKS_PER_SECOND;
                        gucSeconds++;
                        // Handle seconds increment
                        if (gucSeconds>=60)
                        {
                                gucSeconds-=60;
                                // Handle minutes decrement...
                                // There will be a display update in all cases
                                gucDisplayUpdate=TRUE;
                                if ((gucMinutesLsb==1) && (gucMinutesMsb==0))
                                {
                                        gucMinutesLsb=0;
                                        // Timer has counted down to zero
                                        SetTone(TRUE);
                                        gnToneTimer=C_TICKS_FOR_APPROX_5_SECONDS;
                                        gucCounting=FALSE;
                                } else
                                {
                                        if (gucMinutesLsb)
                                        {
                                                gucMinutesLsb--;
                                        } else
                                        {
                                                gucMinutesLsb=9;
                                                gucMinutesMsb--;
                                        }
                                }
                        } // end of seconds wrap-over code
                }       // end of while  ie end of for each tick detected
        }
} // end of elapsed ticks
```

```
                // ******************************
                // STAGE 2 - HANDLING OF DISPLAY UPDATE
                // Translate any changed display values for
                // driving out by the interrupt handler.
                // ******************************
                if (gucDisplayUpdate)
                {
                        gucDisplayMsbSegments=GetSegmentPattern(gucMinutesMsb);
                        gucDisplayLsbSegments=GetSegmentPattern(gucMinutesLsb);
                        gucDisplayUpdate=FALSE;
                }
        }       //end of while 1 loop
}
// ************************************************
// ************************************************
// I N T E R R U P T   H A N D L E R S
// ************************************************
// ************************************************
/*********************************************************************
NAME:           static void __interrupt IntDummy(void)
FUNCTION:       Handles unwanted interrupts.
PARAMETERS: None
RETURN:         None
NOTES:          Defensive programming only - these interrupts should
                        never happen,
*********************************************************************/
static void __interrupt IntDummy(void)
{
        // do nothing except return from interrupt
}
/*********************************************************************
NAME:           static void __interrupt_n IntDummyN(void)
FUNCTION:       Handles unwanted NMI interrupts.
PARAMETERS: None
RETURN:         None
NOTES:          Defensive programming only - these interrupts should
                        never happen in this application,
*********************************************************************/
static void __interrupt_n IntDummyN(void)
{
        // do nothing except return from interrupt
}
/*********************************************************************
NAME:           static void __interrupt IntTBT(void)
FUNCTION:       Handles interrupts from the time base timer.
PARAMETERS: None
RETURN:         None
NOTES:          These are set to occur at 488.28 Hz.
                        Increments elapsed time counter for main line
                        timeouts.
                        Strobes the display and scans the keyboard.
                        Activities switch between three sub-jobs
                        on consecutive interrupts
*********************************************************************/
static void __interrupt IntTBT(void)
{
        unsigned char ucScanCode;
        static unsigned char ucLastScanCode=0;
#ifndef KEYS_VIA_ADC
        unsigned char ucColumns;
        unsigned char ucRow;
        unsigned char volatile ucDelay;
#endif
        // ********************************
        // Stage 1 - increment the interrupt tick
        // count which is used to feed the main line timer(s)
        // with elapsed time..
        // ********************************
        gucIntTicks++;
        // ********************************
        // Stage 2 - subjob despatching
        // ********************************
        switch (gucInterruptSubJob)
        {
                // ********************************
                // Stage 2A - subjob for drive digit A
                // ********************************
```

```
		default :
		case 0 :
			SEGMENT_PORT   = gucDisplayMsbSegments;
			P3CR=C_SEGMENT_MASK; // make all segments outputs
			DIGIT_A_ENABLE = LOW;
			// subjob housekeeping - next interrupt gets different job
			gucInterruptSubJob=1;
		break;
		// *********************************
		// Stage 2B - subjob for drive digit B
		// *********************************
		case 1 :
			DIGIT_A_ENABLE = HIGH;
			P3CR=C_SEGMENT_MASK; // make all segments outputs
			SEGMENT_PORT   = gucDisplayLsbSegments;
			DIGIT_B_ENABLE = LOW;
			// subjob housekeeping - next interrupt gets different job
			gucInterruptSubJob=2;
		break;
		// *********************************
		// Stage 2B - subjob for key scanning
		// *********************************
		case 2 :
			// subjob housekeeping - next interrupt gets different job
			gucInterruptSubJob=0;
			// Remove drive to digit B so we can re-use segments for scanning..
			DIGIT_B_ENABLE = HIGH;
			// After a valid keypress we hold off scanning to let things settle
			// for at least a defined debounce time...
			if (gucDebounceTimer)
			{
				gucDebounceTimer--;
				return;
			}
			// After a valid keypress we hold off scanning until the main line
			// code has 'absorbed' that key press.
			if (gucIntKeyScanCode)
			{
				return;
			}
			// Scan code zero is taken to mean no presses so we start scanning from
			// 1

#ifndef KEYS_VIA_ADC
			// *********************************
			// ROW AND COLUMN KEY-SCANNING SECTION
			// *********************************
			ucScanCode=1;
			// drive a low across all the rows...
			for (ucRow=C_ROW_0;(ucRow & C_ROW_MASK);ucRow<<=1)
			{
				P3CR=C_ROW_MASK; // make all columns inputs
				ROW_PORT=((ROW_PORT & ~C_ROW_MASK) | (~(ucRow) & C_ROW_MASK));
				// wait for lows from rows to propagate through switches
				// and series resistors etc.
				for (ucDelay=0;ucDelay<C_COLUMN_SETTLING_DELAY;ucDelay++);
				// now scan for a low in the columns..
				ucColumns=(COLUMN_PORT & C_COLUMN_MASK);
				// first of all- are there any lows at all??
				if (ucColumns!=C_COLUMN_MASK)
				{
					// to get here there must be at least one low..
					// so let's scan across the bits till we find one..
					while (ucColumns & C_COLUMN_0)
					{
						ucColumns>>=1;
						ucScanCode++;
					}
					// Here we've found the first low and we'll use that
					// but is this a change?
					if (ucScanCode!=ucLastScanCode)
					{
						// yes a new key press was detected - advise
						// the main line code and set the debounce timer
						// so we won't look again for a while.
						gucIntKeyScanCode=ucScanCode;
						gucDebounceTimer=C_DEBOUNCE_KEY_SCAN_PERIODS;
```

```
                                        ucLastScanCode=ucScanCode;
                                }
                                return;
                        }
                        ucScanCode+=C_NUMBER_OF_COLUMNS;
                }
                ucLastScanCode=0;
#else
                // **********************************
                // ADC KEY DETECTION SECTION
                // **********************************
                ucScanCode=GetADCKeyScanCode();
                // Here we've found the first low and we'll use that
                // but is this a change?
                if (ucScanCode!=ucLastScanCode)
                {
                        // yes a new key press was detected - advise
                        // the main line code and set the debounce timer
                        // so we won't look again for a while.
                        gucIntKeyScanCode=ucScanCode;
                        gucDebounceTimer=C_DEBOUNCE_KEY_SCAN_PERIODS;
                }
                ucLastScanCode=ucScanCode;
#endif
                break;
        }
}
// ************************************************
// ************************************************
// V E C T O R   T A B L E
// ************************************************
// ************************************************

#pragma section const VECTORS 0xffe0

static const void *const _IntTbl[] = {
        IntDummy,                               // int5 (external interrupt 5)
        IntDummy,                               // reserved
        IntDummy,                               // reserved
        IntDummy,                               // intadc (adc interrupt)
        IntDummy,                               // inttc4 (timer/counter 4 interrupt)
        IntDummy,                               // inttc3 (timer/counter 3 interrupt)
        IntDummy,                               // reserved
        IntDummy,                               // reserved
        IntDummy,                               // reserved
        IntTBT,                                 // inttbt (time base timer interrupt)
        IntDummy,                               // int1 (external interrupt 1)
        IntDummy,                               // int0 (external interrupt 0)
        IntDummyN,                              // intwdt (watchdog timer interrupt)
        IntDummyN,                              // intatrap (address trap interrupt)
        IntDummyN,                              //  intswi/intundef  (software  interrupt/undefined
instruction interrupt)
        startup,                                // (reset)
};
/*****************************************************************************
End of MinuteMinder.c
*****************************************************************************

/
```

Disclaimer

- TOSHIBA is continually working to improve the quality and reliability of its products. Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the buyer, when utilizing TOSHIBA products, to comply with the standards of safety in making a safe design for the entire system, and to avoid situations in which a malfunction or failure of such TOSHIBA products could cause loss of human life, bodily injury or damage to property. In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent TOSHIBA products specifications. Also, please keep in mind the precautions and conditions set forth in the "Handling Guide for Semiconductor Devices," or "TOSHIBA Semiconductor Reliability Handbook" etc..
- The TOSHIBA products listed in this document are intended for usage in general electronics applications (computer, personal equipment, office equipment, measuring equipment, industrial robotics, domestic appliances, etc.). These TOSHIBA products are neither intended nor warranted for usage in equipment that requires extraordinarily high quality and/or reliability or a malfunction or failure of which may cause loss of human life or bodily injury ("Unintended Usage"). Unintended Usage include atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, medical instruments, all types of safety devices, etc.. Unintended Usage of TOSHIBA products listed in this document shall be made at the customer's own risk.
- The information contained herein is presented only as a guide for the applications of our products. No responsibility is assumed by TOSHIBA CORPORATION for any infringements of intellectual property or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any intellectual property or other rights of TOSHIBA CORPORATION or others.

The information contained herein is subject to change without notice.

OVERSEAS SUBSIDIARIES AND AFFILIATES

Toshiba Electronics Europe GmbH

Düsseldorf Head Office
Hansaallee 181, D-40549 Düsseldorf
Germany
Tel: (0211)5296-0 Fax: (0211)5296-400

München Office
Büro München Hofmannstrasse 52,
D-81378, München, Germany
Tel: (089)748595-0 Fax: (089)748595-42

Toshiba Electronics France SARL
Immeuble Robert Schumann 3 Rue de Rome,
F-93561, Rosny-Sous-Bois, Cedex, France
Tel: (1)48-12-48-12 Fax: (1)48-94-51-15

Toshiba Electronics Italiana S.R.L.
Centro Direzionale Colleoni
Palazzo Perseo Ingr. 2-Piano 6,
Via Paracelso n.12, 1-20041 Agrate Brianza
Milan, Italy
Tel: (039)68701 Fax:(039)6870205

Toshiba Electronics España, S.A
Parque Empresarial San Fernando Edificio Europa,
1a Planta, ES-28831 Madrid, Spain
Tel: (91)660-6700 Fax:(91)660-6799

Toshiba Electronics(UK) Limited
Riverside Way, Camberley Surrey,
GU15 3YA, U.K.
Tel: (01276)69-4600 Fax: (01276)69-4800

Toshiba Electronics Scandinavia AB
Gustavslundsvägen 12, 2nd Floor
S-161 15 Bromma, Sweden
Tel: (08)704-0900 Fax: (08)80-8459

Toshiba Electronics Asia

(Singapore) Pte. Ltd.
Singapore Head Office
438B Alexandra Road, #06-08/12 Alexandra Technopark, Singapore 119968
Tel: (278)5252 Fax: (271)5155

Bangkok Office
135 Moo 5 Bangkadi Industrial Park,
Tivanon Rd.,Bangkadi Amphur Muang
Pathumthani, Bangkok, 12000, Thailand
Tel: (02)501-1635 Fax: (02)501-1638

Toshiba Electronics Trading
(Malaysia)Sdn. Bhd.
Kuala Lumpur Head Office
Suite W1203, Wisma Consplant, No.2,
Jalan SS 16/4, Subang Jaya, 47500 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: (3)731-6311 Fax: (3)731-6307

Penang Office
Suite 13-1, 13th Floor, Menard Penang Garden,
26th Floor, Citibank Tower, Valero Street,
Makati, Manila, Philippines
Tel: (02)750-5510 Fax: (02)750-5511

Toshiba Electronics Philippines, Inc.
26th Floor, Citibank Tower, Valero Street,
Makati, Manila, Philippines
Tel: (02)750-5510 Fax: (02)750-5511

Toshiba America Electronic Components, Inc.

Headquarters-Irvine, CA
9775 Toledo Way, Irvine, CA 92618, U.S.A.
Tel: (949)455-2000 Fax: (949)859-3963

Boulder, CO
3100 Arapahoe Avenue, Ste. 500,
Boulder, CO 80303, U.S.A.
Tel: (303)442-3801 Fax: (303)442-7216

Boynton Beach, FL(Orlando)
11924 W. Forest Hill Blvd., Ste. 22-337,
Boynton Beach, FL 33414, U.S.A.
Tel: (561)374-6193 Fax: (561)374-6194

Deerfield, IL(Chicago)
One Pkwy., North, Suite 500, Deerfield,
IL 60015-2547, U.S.A.
Tel: (847)945-1500 Fax: (847)945-1044

Duluth, GA(Atlanta)
3700 Crestwood Parkway, Ste. 460,
Duluth, GA 30096, U.S.A.
Tel: (770)931-3363 Fax: (770)931-7602

Edison, NJ
2035 Lincoln Hwy. Ste. #3000, Edison
NJ 08817, U.S.A.
Tel: (732)248-8070 Fax: (732)248-8030

Orange County, CA
2 Venture Plaza, #500 Irvine, CA 92618,
U.S.A.
Tel: (949)453-0224 Fax: (949)453-0125

Portland, OR
1700 NW 167th Place, #240,
Beaverton, OR 97006, U.S.A.
Tel: (503)629-0818 Fax: (503)629-0827

Richardson, TX(Dallas)
777 East Campbell Rd., Suite 650,
Richardson,
TX 75081, U.S.A.
Tel: (972)480-0470 Fax: (972)235-4114

San Jose Engineering Center, CA
1060 Rincon Circle, San Jose, CA 95131,
U.S.A.
Tel: (408)526-2400 Fax:(408)526-2410

Wakefield, MA(Boston)
401 Edgewater Place, Suite #360, Wakefield,
MA 01880-6229, U.S.A.
Tel: (781)224-0074 Fax: (781)224-1095

Toshiba Do Brasil S.A.

Electronic Components Div.
Estrada Dos Alvarengas, 5. 500
09850-550-Sao Bernardo do campo - SP
Tel: (011)7689-7171 Fax: (011)7689-7189

Toshiba Electronics Asia, Ltd.

Hong Kong Head Office
Level 11, Top Glory Insurance Building,
Grand Century
Place, No.193, Prince Edward Road West,
Mong Kok, Kowloon, Hong Kong
Tel: 2375-6111 Fax: 2375-0969

Beijing Office
Rm 714, Beijing Fortune Building,
No.5 Dong San Huan Bei-Lu, Chao Yang District, Beijing, 100004, China
Tel: (010)6590-8795 Fax: (010)6590-8791

Chengdu Office
Unit F, 18th Floor, New Times Plaza, 42
Wenwu Road, Xinhua Avenue, Chengdu,
610017, China
Tel: (028)675-1773 Fax: (028)675-1065

Shenzhen Office
Rm 3010-3012, Office Tower Shun Hing Square, Di Wang Commercial Centre, 333 ShenNan East Road, Shenzhen, 518008, China
Tel: (0755)246-1582 Fax: (0755)246-1581

Toshiba Electronics Korea Corporation
Seoul Head Office
14/F, KEC B/D, 257-7 Yangjae-Dong,
Seocho-ku, Seoul, Korea
Tel: (02)589-4334 Fax: (02)589-4302

Gumi Office
6/F, Ssangyong Investment Securities B/D,
56 Songjung-Dong, Gumi City
Kyeongbuk, Korea
Tel: (82)54-456-7613 Fax: (82)54-456-7617

Toshiba Technology Development (Shanghai) Co., Ltd.
23F, Shanghai Senmao International Building, 101 Yin Cheng East Road, Pudong New Area, Shanghai, 200120, China
Tel: (021)6841-0666 Fax: (021)6841-5002

Tsurong Xiamen Xiangyu Trading Co., Ltd.
8N, Xiamen SEZ Bonded Goods Market Building, Xiamen, Fujian, 361006, China
Tel: (0592)562-3798 Fax: (0592)562-3799

Toshiba Electronics Taiwan Corporation
Taipei Head Office
17F, Union Enterprise Plaza Bldg. 109
Min Sheng East Rd., Section 3, 0446 Taipei, Taiwan
Tel: (02)514-9988 Fax: (02)514-7892

Kaohsiung Office
16F-A, Chung-Cheng Bldg., Chung-Cheng 3Rd., 80027, Kaohsiung, Taiwan
Tel: (07)222-0826 Fax: (07)223-0046

TOSHIBA Semiconductor Websites

Europe:	www.toshiba-components.com
Japan:	www.semicon.toshiba.co.jp/eng/index.html
America:	www.toshiba.com/taec/