RENESAS CONTEST 2003

BY LINDSAY MEEK and ILARIO DIMASI

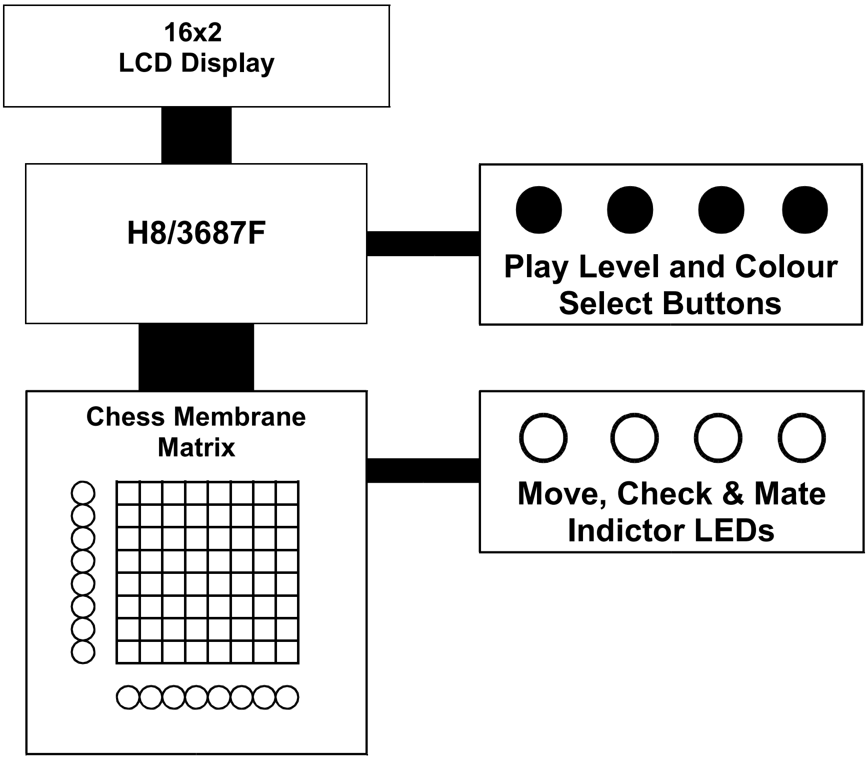
CHESS CHALLENGER

1. ABSTRACT

This project utilises the Renesas H8/3687F as the central processing unit in an electronic chessboard. The chessboard itself consists of a matrix of pressure sensors with LEDs along the rows and columns. The H8/3687F interfaces to the chessboard matrix, and scans it to detect the movement of pieces from a human opponent.

A chess algorithm then devises counter moves, which are indicated using the row and column LEDs. The chess algorithm is capable of playing at four difficulty levels, corresponding to the number of moves ahead it is processing.

2. BLOCK DIAGRAM


16x2
LCD Display
H8/3687F
Play Level and Colour Select Buttons
Chess Membrane Matrix
Move, Check & Mate Indictor LEDs

3. DESCRIPTION

1 Hardware

The hardware components for the chess challenger consist of the H8/3687F evaluation board piggy-backed onto an expansion board using a 28-pin header. The expansion board consists of some LEDs, a buzzer, and a pressure-sensitive 8x8 chessboard matrix with LEDs along the rows and columns.

The four buttons and the 16x2 LCD display on the evaluation board are used during game initialisation to pick black or white, select the level of play and indicate the last move made.

The LEDs on the expansion board are used to indicate the current move; black or white, and whether or not a king is check, and also mate.

The 8x8 chessboard matrix operates in a similar fashion to a standard keyboard. The interface to the matrix consists of 8 row bits and 8 column bits. Scanning the chessboard for a ‘pressed’ piece consists of energising each column in sequence, and reading the contents of the row outputs. If a row bit is asserted for a given column, then the corresponding X,Y coordinate is ‘pressed’. This keyboard scanning method is used as the primary input when sensing the movement of pieces on the board; the source square is pressed first by holding down the piece and then it is moved to the destination square and pressed again.

The buzzer is used as an audible feedback to the movement of pieces, as does the row/column LEDs provide visual feedback as to the X,Y coordinate. The row/column LEDs are also used to demonstrate the computer move to the human opponent by showing the source piece and its destination location on the chessboard.

To reduce the I/O requirements, the row and column signals are multiplexed with the LED outputs. The LEDs are pulsed rapidly when active and appear to be steadily lit due to the persistence-of-vision effect.

2 Software

The chess game software is written in C, and has a simple main loop that can be summarised using the following pseudo code:

```
Get Computer Difficulty Level (1-4)
Get Human Side (Black or White)
Set Up Board
While Not Mate
    Display Game Status
    Get A Valid Human Move
    Update Chessboard
    Determine Computer Move
    If Computer Move Made Then
        Update Chessboard
        If Human King is in Check And King Cannot Move Then
            Mate
        End If
    Else
        Mate
        Display Game Status
    End If
End While
Restart
```

The chessboard is represented within a data array consisting of 32 bytes, where pieces are mapped into nibbles thus resulting in 64 squares. Each square can consist of a value representing a Pawn, Rook, Knight, Bishop, King, Queen or Empty and whether the piece belongs to Black or White.

The data structure in the chess game contains the variables used for the algorithm to determine the computer's move.

3.1.1 Computer Opponent Chess Algorithm

The algorithm used to generate the computer's move operates by searching the chessboard for the best possible move.

The search operates by scanning the chessboard array for the squares containing pieces of a given colour. If a chess piece is encountered for the given side's move, it is then moved in all the directions that constitute legal moves.

For each legal move of the chess piece, a movement score is derived which gives an indication of the value of the move. The move with the highest score is selected as the preferred move.

The movement score is derived from the value of the piece that is being taken and whether the side's King is being threatened. The values of the Pawn, Rook, Knight, Bishop, Queen and King are fixed at {2,6,6,10,18,40}. Therefore, the best-scored move for the side will tend to be the capturing of a high valued piece.

Depending on whether the computer is playing White or Black depends on how the movement score is affected. If making a move for Black, the movement score is to be minimised and maximised if making a move for White, to be considered as the best score for the moving side. It continues down the move branch by moving each side in turn until the maximum search depth has been reached. Once the maximum depth has been reached, any moves made are taken back and another made thus exploring all possible moves. When all possible moves have been explored, the best move based on the best score is made. This method is the min-max algorithm utilising a brute force searching.

The search algorithm is implemented using a recursive structure, where the stack is used to record changes made to the main chessboard array. This allows the algorithm to 'reverse' modifications made to the array, without requiring vast amounts of memory. Each chessboard search represents another level of recursion, and two levels of recursion correspond to a ply.

A 'ply' refers to the 'moves ahead' that the algorithm is able to search. One ply only searches a single move ahead. In the chess challenger, this ply level directly corresponds to the difficulty level, which is set by the user when the game starts.

For two-ply, the search is extended such that for each counter move response to a computer move, the chessboard is rescanned for corresponding valid computer moves as was done initially. The process repeats for the counter move phase as described above. The three and four-ply level extend the search further by repeating the search process to one and two additional levels. Four ply algorithms are difficult to beat by most humans (knowing the algorithm's weaknesses helps, however).

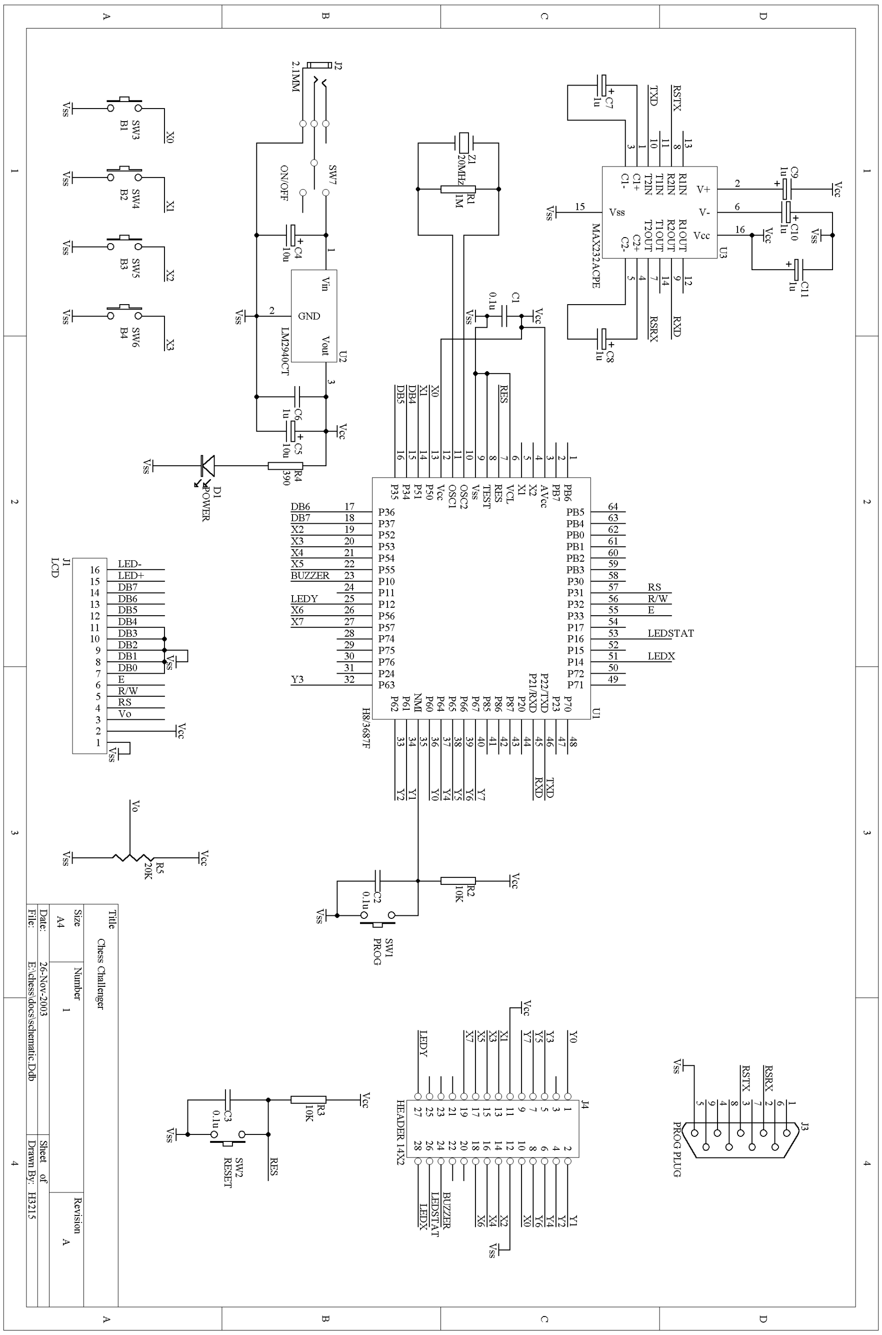
3.1.2 Enhancements

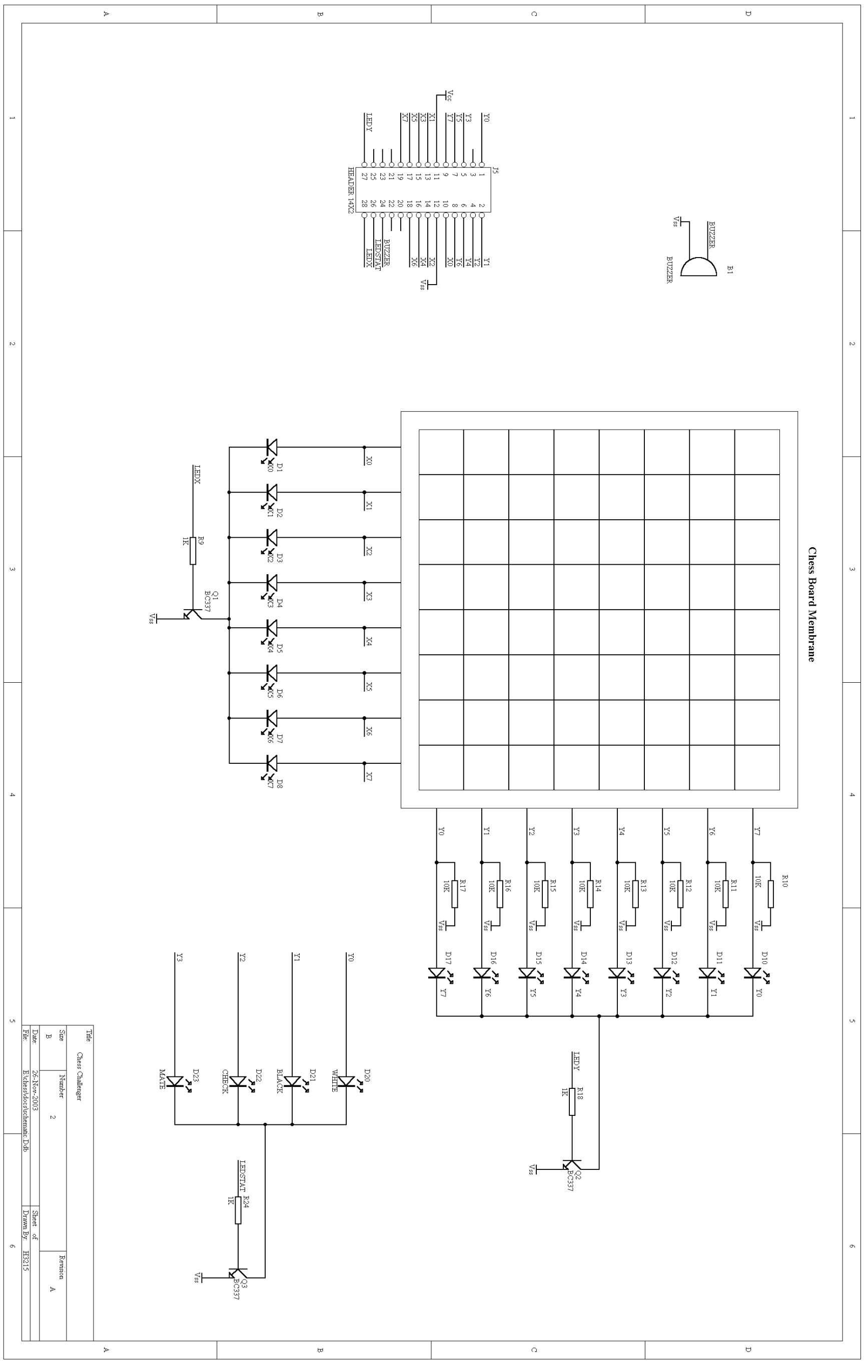
Some enhancements have been made to the computer chess algorithm to improve its performance.

The first enhancement is an alteration to the chessboard search pattern, so that moves near the centre of the X-axis are favoured. This serves to discourage the computer from playing 'into a corner ' which can reduce the number of possible moves. The search is altered so that chessboard X coordinates are scanned 'centre out' in the sequence {4,3,5,2,6,1,7,0} instead of left-to-right {0,1,2,3,4,5,6,7}.

The other enhancement to the algorithm limits the search complexity at higher difficulty levels by ‘pruning’ the recursive searches for counter moves. The criterion for abandoning a recursive search at a given depth is if the score of the current move is better than best score encountered at the previous search depth.

4. SCHEMATICS




Chess Board Membrane
J5
HEADER 14X2
Vcc
Vss
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
X0
X1
X2
X3
X4
X5
X6
X7
LEDX
LEDY
LEDSTAT
BUZZER
B1
D1
D2
D3
D4
D5
D6
D7
D8
D10
D11
D12
D13
D14
D15
D16
D17
D20
D21
D22
D23
WHITE
BLACK
CHECK
MATE
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R24
1K
10K
Q1
Q2
Q3
BC337
Title
Chess Challenger
Size
B
Number
2
Revision
A
Date:
26-Nov-2003
File:
E:\chess\docs\schematic.Ddb
Sheet of
Drawn By:
H3215

5. PICTURE

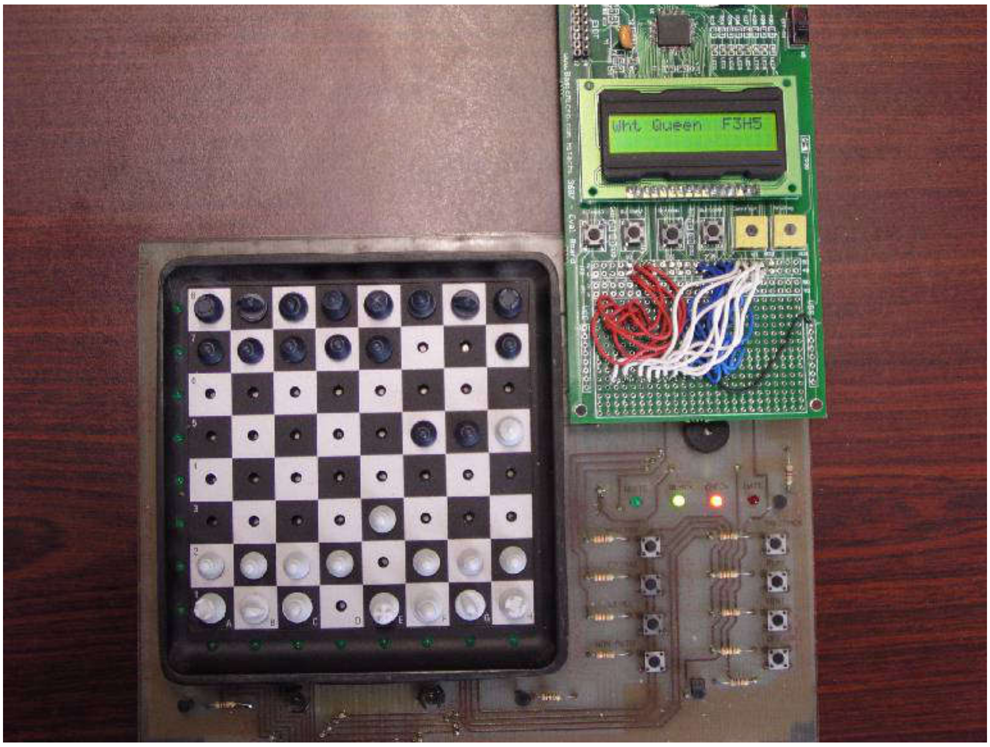


Figure 1. Chess challenger showing black in check

6. SOURCE CODE

3 CHESSMAIN.C

```
/*
Program:       Main Chess Game Algorithm
Author:        H3215
*/

/* Hardware interface drivers */
#include "drivers.h"

/* Datatypes */
typedef unsigned char  Byte;
typedef unsigned char  BOOL;

typedef enum
{
        FALSE,
        TRUE
};

typedef BOOL          TSide;

typedef struct
{
    Byte    x1;
    Byte    x2;
    Byte    y1;
    Byte    y2;
    Byte    depth;
    TSide   side;
    BOOL    ok;
} TChess;

/* Piece & Colour definitions for populated chess board array */
#define PAWN          1
#define KNIGHT        2
#define BISHOP        3
#define ROOK          4
#define QUEEN         5
#define KING          6

#define NO_PIECE      0
#define WHITE         1
#define BLACK         0

#define  W_PAWN       PAWN + (WHITE<<3)
#define W_ROOK        ROOK + (WHITE<<3)
#define W_KNIGHT      KNIGHT + (WHITE<<3)
#define W_BISHOP      BISHOP + (WHITE<<3)
#define W_QUEEN       QUEEN + (WHITE<<3)
#define W_KING        KING + (WHITE<<3)
#define B_PAWN        PAWN + (BLACK<<3)
#define B_ROOK        ROOK + (BLACK<<3)
#define B_KNIGHT      KNIGHT + (BLACK<<3)
#define B_BISHOP      BISHOP + (BLACK<<3)
#define B_QUEEN       QUEEN + (BLACK<<3)
#define B_KING        KING + (BLACK<<3)

/* Unoccupied square on the board */
#define SQUARE_NONE   255

#define WHTLED      0x01    // white status led
#define BLKLED      0x02    // black status led
#define CHKLED      0x04    // check status led
#define MTELED      0x08    // mate status led

#define MAXLK       6       // Maximum search depth
#define MAX                   (MAXLK+1)
```

```
/*
        Within each nibble of the board memory, the highest bit in a nibble
        represents the piece colour as defined above.
*/

/* Board definition at the start of the game */
static const    Byte   PieceSet[64] =
{
        W_ROOK,W_KNIGHT,W_BISHOP,W_QUEEN,W_KING,W_BISHOP,W_KNIGHT,W_ROOK,
        W_PAWN,W_PAWN,W_PAWN,W_PAWN,W_PAWN,W_PAWN,W_PAWN,W_PAWN,
        NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,
         NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,
        NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,
        NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,NO_PIECE,
        B_PAWN,B_PAWN,B_PAWN,B_PAWN,B_PAWN,B_PAWN,B_PAWN,B_PAWN,
        B_ROOK,B_KNIGHT,B_BISHOP,B_QUEEN,B_KING,B_BISHOP,B_KNIGHT,B_ROOK
};

/* Value assigned to piece based on its class */
static  const   Byte    PieceValue[7] = { 0,2,6,6,10,18,40 };

//--------------------------------------------------------------------------
// General Set and Look Up piece functions
static  void    SetUpBoard(void);
static  void    SetPiece(Byte square,Byte piece);
static  Byte    GetPiece(Byte square);
static  Byte    PieceDefn(Byte square);
static  Byte    Col(Byte square);
static  Byte    Row(Byte square);
static  Byte    AbsDiff(Byte x,Byte y);
// Chess algorithm functions
static  Byte    BdLkup(Byte col,Byte row);
static  void    MoveGen(Byte depth,Byte piece);
static Byte     Bounds(Byte depth,Byte pawnf);
static Byte     Move(Byte depth,Byte msave);
static void     MoveF(Byte depth,Byte msave,Byte mxy,Byte mpq);
static void     MoveB(Byte depth,Byte msave,Byte mxy,Byte mpq);
static void     Bishop(Byte depth);
static void     Knight(Byte depth);
static void     Rook(Byte depth);
static void     King(Byte depth);
static void     Pawn(Byte depth);
static Byte     FindKing(TSide side);
static BOOL     IsKingInCheck(TSide side);
static void     CheckMove(TChess* pChess);
static Byte     MinMax(Byte depth);
static void     DoChess(TChess* pChess);
static void     CheckPromotePawn(Byte to,TSide side);
static void     CheckPerformCastle(Byte from,Byte to,TSide side);
static void     ShowMoveLCD(Byte from,Byte to,Byte piece);

/*
 Algorithm working arrays
*/

static  Byte    ChessBoard[32];
static  Byte    BestScore[MAX];
static  Byte    BestX[MAX];
static  Byte    BestY[MAX];
static  Byte    BestP[MAX];
static  Byte    BestQ[MAX];
static  Byte    MoveOk[MAX];
static  TSide   Side[MAX];

static  Byte    t1,t2,t3,t4,function,pieceValue,x,y,p,q,kingF,ok,castle;
static  Byte    x1,x2,y1,y2,forcePlay,mySide,mate,status,PlayLevel,PruneTree;
static  TSide   side = WHITE;

#define STATUS  (((side==WHITE)?CHESS_WHITE:CHESS_BLACK)+(IsKingInCheck(side)?
CHESS_CHECK:0)+mate)

/* Mainline */

void    ChessMain(void)
{
        Byte    from,to,capPiece,sq;
        TChess  chess;
```

```
SetUpBoard();
while (1)
{
        // Display Status
        status = STATUS;
        Chess_LedStat(status);
        // Test for Game end?
        if (mate==CHESS_MATE)
                continue;
        // Is this my Opponents Move?
        if (side==mySide)
        {
                // Get Opponents move
                from = Chess_ScanBoard(-1,STATUS);
                to = Chess_ScanBoard(from,STATUS);
                //Check for user correction
                ShowMoveLCD(from,to,GetPiece(from));
                if (from!=to)
                {
                        chess.x1 = Col(from);
                        chess.y1 = Row(from);
                        chess.x2 = Col(to);
                        chess.y2 = Row(to);
                        chess.depth = 0;
                        DoChess(&chess);
                        if (chess.ok)
                        {
                                // Move Valid...well update Chessboard
                                capPiece = GetPiece(to);
                                SetPiece(to,GetPiece(from));
                                SetPiece(from,NO_PIECE);
                                if (IsKingInCheck(side))
                                {
                                        // Invalid...can't move into check
                                        SetPiece(from,GetPiece(to));
                                        SetPiece(to,capPiece);
                                        // InValid Move
                                        // tell user to put back piece
                                        do {
                                                Chess_Beep();
                                                Chess_Beep();
                                                to = Chess_ScanBoard(from,STATUS);
                                        } while (to!=from);
                                }
                                else
                                {
                                        CheckPromotePawn(to,side);
                                        CheckPerformCastle(from,to,side);
                                        side ^= 1;      // Swap sides
                                }
                        }
                        else
                        {
                                // InValid Move....tell user to put back piece
                                do {
                                        Chess_Beep();
                                        Chess_Beep();
                                        to = Chess_ScanBoard(from,STATUS);
                                } while (to!=from);
                        }
                }
        }
        // It is the Computers Move.
        else
        {
                chess.depth = PlayLevel;
                chess.side = side;
                DoChess(&chess);
                from = ((chess.y1*8)+chess.x1);
                to = ((chess.y2*8)+chess.x2);
                if (MoveOk[chess.depth])
                {
                        // Show Computers Move
                        ShowMoveLCD(from,to,GetPiece(from));
                        mate = 0;
                        // Do Computers Move
```

```
                                Chess_Beep();
                                for (sq=Chess_ScanBoard(from,STATUS);sq!=from;)
                                {
                                        Chess_Beep();
                                        Chess_Beep();
                                        sq = Chess_ScanBoard(from,STATUS);
                                }
                                for (sq=Chess_ScanBoard(to,STATUS);sq!=to;)
                                {
                                        Chess_Beep();
                                        Chess_Beep();
                                        sq = Chess_ScanBoard(to,STATUS);
                                }
                                // Make the Move on the ChessBoard
                                SetPiece(to,GetPiece(from));
                                SetPiece(from,NO_PIECE);
                                // Test for PAWN promotion and castling
                                CheckPromotePawn(to,side);
                                CheckPerformCastle(from,to,side);
                                side ^= 1;      //swap sides
                                // Is opponents KING in check ... assume NO
                                kingF = FALSE;
                                if (IsKingInCheck(side))
                                {
                                        // KING is in check...indicate it.
                                        kingF = TRUE;
                                        // Can the opponents KING move?
                                        from = ((chess.y2*8)+chess.x2);
                                        // Assume it can't
                                        mate = CHESS_MATE;
                                        for (to=0; to<64; to++)
                                        {
                                                chess.x1 = Col(from);
                                                chess.y1 = Row(from);
                                                chess.x2 = Col(to);
                                                chess.y2 = Row(to);
                                                chess.depth = 0;
                                                DoChess(&chess);
                                                if (chess.ok)
                                                {
                                                        capPiece = GetPiece(to);
                                                        SetPiece(to,GetPiece(from));
                                                        SetPiece(from,NO_PIECE);
                                                        if (!IsKingInCheck(side))
                                                                mate = 0;
                                                                // KING can Move!
                                                        SetPiece(from,GetPiece(to));
                                                        SetPiece(to,capPiece);
                                                }
                                                if (!mate)
                                                        break;
                                        }
                                }
                        }
                        else
                                mate = CHESS_MATE;
                }
        }
}

/*
        Initialisation Function
*/
static  void    SetUpBoard(void)
{
        Byte    i,x;

        // Initialise some globals
        mate = 0;
        castle = 0xFF;
        PlayLevel = 1;
        PruneTree = 0;
        // Initialise the Chess Board
        for (i=0,x=0; i<64; i+=2,x++)
                ChessBoard[x] = (PieceSet[i]<<4)+PieceSet[i+1];
        // Select PlayLevel
        LCD_clear();
        LCD_at(0,0);
        LCD_str("Play Level  ");
        LCD_at(12,0);
        LCD_char(PlayLevel+'0');
        LCD_at(0,1);
        LCD_str(" OK   -   +");
        x=1;
        i=0;
        while(x)
        {
                if (ScanButton(i))
                {
                        switch (i)
                        {
                        case 0:
                                if (PlayLevel<MAXLK)
                                        PlayLevel++;
                                break;
                        case 1:
                                if (PlayLevel>1)
                                        PlayLevel--;
                                break;
                        case 2:
                                x = 0;
                                break;
                        }
                        while (ScanButton(i));
                        LCD_at(12,0);
                        LCD_char(PlayLevel+'0');
                }
                if (++i==4)
                        i = 0;
        }
        while(ScanButton(0) || ScanButton(1) || ScanButton(2) || ScanButton(3));
        // Select Play Side
        LCD_clear();
        LCD_at(0,0);
        LCD_str("Play White = B3");
        LCD_at(0,1);
        LCD_str("Play Black = B0");
        while(!ScanButton(3) && !ScanButton(0));
        LCD_clear();
        mySide = (ScanButton(3))?WHITE:BLACK;
        // Shall we Prune the tree
        while(ScanButton(0) || ScanButton(1) || ScanButton(2) || ScanButton(3));
        PruneTree = (PlayLevel>4)?TRUE:FALSE;
}

/*
        Access main chess board
*/
static  void    SetPiece(Byte square,Byte piece)
{
        Byte*   pPos;

        pPos = &ChessBoard[square/2];
        if ((square & 0x01)==0)
                *pPos = (*pPos&0x0F)|(piece<<4);
        else
                *pPos = (*pPos&0xF0)|piece;
}

static  Byte    GetPiece(Byte square)
{
        Byte*   pPos;

        pPos = &ChessBoard[square/2];
        if ((square & 0x01)==0)
                return  ((*pPos>>4)&0x0F);
        return  (*pPos&0x0F);
}
//-----------------------------------------------------------------------------
static  Byte    PieceDefn(Byte square)
{
        return  (GetPiece(square)&0x07);
}
//-----------------------------------------------------------------------------
```

```
static  Byte    Col(Byte square)
{
        return  (square%8);
}
//-------------------------------------------------------------------------------
static  Byte    Row(Byte square)
{
        return  (square/8);
}
//-------------------------------------------------------------------------------
static  Byte    AbsDiff(Byte x,Byte y)
{
        if (x>=y)
                return  (x-y);
        return  (y-x);
}
//-------------------------------------------------------------------------------
static  Byte    BdLkup(Byte col,Byte row)
{
        return  GetPiece((row*8)+col);
}
//-------------------------------------------------------------------------------
static  void    MoveGen(Byte depth,Byte piece)
{
    switch(piece & 0x07)
    {
    case PAWN:       Pawn(depth);     break;
    case BISHOP:     Bishop(depth);   break;
    case KNIGHT:     Knight(depth);   break;
    case ROOK:       Rook(depth);     break;
    case QUEEN:      Rook(depth);     Bishop(depth);   break;
    case KING:       King(depth);     break;
    }
}
//-------------------------------------------------------------------------------
static  Byte    Bounds(Byte depth,Byte pawnf)
{
    Byte    piece,side;

    // Check if off board
    if (p>7 || q>7)
        return  2;
    // Get a piece and it's side
    piece = BdLkup(p,q);
    side = (piece>>3);
    piece = (piece & 0x07);
    // Return normal stops
    if (pawnf==2)
    {
        if (piece==NO_PIECE)    return  0;
        if (side==Side[depth])  return  2;
        return  1;
    }
    else if (pawnf>2)   return  pawnf;
    //We have a pawn!
    if (pawnf==0 && piece==NO_PIECE)    return  0;
    if (pawnf==1 && side!=Side[depth] && piece!=NO_PIECE)   return  1;
    return  2;
}
//-------------------------------------------------------------------------------
static  Byte    Move(Byte depth,Byte msave)
{
    Byte    mxy,mpq;

    msave = Bounds(depth,msave);    // msave doubles as pawn flag
    if (msave==2)   return  2;      // stop
    msave = msave<<4;               // save stop in upper nibble
    switch(function)
    {
    case 0: // Just checking if a move is valid
        if (p==x2 && q==y2) ok = TRUE;
        break;
    default:    // Move a piece
        msave = msave | BdLkup(p,q);
        mxy = ((y*8)+x);
        mpq = ((q*8)+p);
        MoveF(depth,msave,mxy,mpq);
        // Don't move if in check
//        if (!IsKingInCheck(Side[depth]))
            t1 = MinMax(depth-1);       // Recursive call
//        else if (!t1)
//            t1 = (Side[depth]==WHITE)?1:255;
        MoveB(depth,msave,mxy,mpq);
        x = Col(mxy);
        y = Row(mxy);
        p = Col(mpq);
        q = Row(mpq);
        // Detect best move
        t2 = 0;
        t3 = BestScore[depth];
        t4 = Side[depth];
        if (t4==WHITE && t1>t3) t2 = 1; // Best for WHITE
        if (t4==BLACK && t1<t3) t2 = 1; // Best for BLACK
        if (t2)// && t1!=255 && t1!=1)
        {
            BestScore[depth] = t1;
            BestX[depth] = x;
            BestY[depth] = y;
            BestP[depth] = p;
            BestQ[depth] = q;
            MoveOk[depth] = TRUE;
        }
        break;
    }
    return  (msave>>4);
}
//-----------------------------------------------------------------------------
static  void    MoveF(Byte depth,Byte msave,Byte mxy,Byte mpq)
{
    t2 = msave & 0x0F;      // piece taken
    t3 = (t2>>3);
    t4 = PieceValue[t2 & 0x07];
    if ((t2 & 0x07)==KING)  kingF = TRUE;
    // Was this a Castle?
    msave >>= 4;
    if (msave>=4)
    {
        SetPiece(((msave==4)?(mpq-1):(mpq+1)),GetPiece((msave==4)?(mpq+1):(mpq-2)));
        SetPiece(((msave==4)?(mpq+1):(mpq-2)),NO_PIECE);
    }
    // Subs zero for empty spaces
    if (t3)
        pieceValue -= t4;
    else
        pieceValue += t4;
    // Make the move
    msave = GetPiece(mxy);
        SetPiece(mpq,msave);
        SetPiece(mxy,NO_PIECE);
    // Prevent King from moving
    if ((msave & KING)==KING)
    {
        if ((msave>>3)==WHITE)
        {
            castle &= 0x7F;     // Indicate WHITE KING moved
            pieceValue -= 1;
        }
        else
        {
            castle &= 0xF7;     // Indicate BLACK KING moved
            pieceValue += 1;
        }
    }
}
//-----------------------------------------------------------------------------
static  void    MoveB(Byte depth,Byte msave,Byte mxy,Byte mpq)
{
    t2 = msave & 0x0F;      // piece taken
    t3 = (t2>>3);
    t4 = PieceValue[t2 & 0x07];
    if ((t2 & 0x07)==KING)  kingF = FALSE;
    // Was this a Castle?
    msave >>= 4;
    if (msave>=4)
    {
        SetPiece(((msave==4)?(mpq+1):(mpq-2)),GetPiece((msave==4)?(mpq-1):(mpq+1)));
        SetPiece(((msave==4)?(mpq-1):(mpq+1)),NO_PIECE);
    }
    // Subs zero for empty spaces
    if (t3)
        pieceValue += t4;
    else
        pieceValue -= t4;
    // Make the move
    msave = GetPiece(mpq);
       SetPiece(mxy,msave);
       SetPiece(mpq,t2);
    // Prevent King from moving
    if ((msave & KING)==KING)
    {
        if ((msave>>3)==WHITE)
        {
            castle |= 0x80;
            pieceValue += 1;
        }
        else
        {
            castle |= 0x08;
            pieceValue -= 1;
        }
    }
}
//----------------------------------------------------------------------------
static  void    Bishop(Byte depth)
{
    Byte    stop;

    // Test NE direction
    stop = 0; p = x; q = y;
    while (stop==0)
    {
        p++; q++;
        stop = Move(depth,2);
    }
    // Test SE direction
    stop = 0; p = x; q = y;
    while (stop==0)
    {
        p++; q--;
        stop = Move(depth,2);
    }
    // Test NW direction
    stop = 0; p = x; q = y;
    while (stop==0)
    {
        p--; q++;
        stop = Move(depth,2);
    }
    // Test SW direction
    stop = 0; p = x; q = y;
    while (stop==0)
    {
        p--; q--;
        stop = Move(depth,2);
    }
}
//----------------------------------------------------------------------------
static  void    Knight(Byte depth)
{
    Byte    dy,inc;

    p = x-3; dy = 0; inc = 1;
    while (p!=(x+2))
    {
        dy += inc;
        p++;
        q = y + dy;
        if (p==x)
        {
            inc = 255;
            continue;
        }
        Move(depth,2);
        q = y - dy;
        Move(depth,2);
    }
}
//---------------------------------------------------------------------------
static  void    Rook(Byte depth)
{
    Byte    stop;

    // Test E direction
    stop = 0; p = x; q = y;
    while (stop==0)
    {
        p++;
        stop = Move(depth,2);
    }
    // Test W direction
    stop = 0; p = x; q = y;
    while (stop==0)
    {
        p--;
        stop = Move(depth,2);
    }
    // Test N direction
    stop = 0; p = x; q = y;
    while (stop==0)
    {
        q++;
        stop = Move(depth,2);
    }
    // Test S direction
    stop = 0; p = x; q = y;
    while (stop==0)
    {
        q--;
        stop = Move(depth,2);
    }
}
//---------------------------------------------------------------------------
static  void    King(Byte depth)
{
    Byte    sp,sq;

    sp = p;
    sq = q;
    // Test for castle ... ignore if done already
    if (x==4 && (y==0 || y==7) && kingF==0)
    {
        if (function==0)    // Is this a user move?
            p = x2;         // Yes
        p = ((p<x)?0x02:0x04);
        p = ((Side[depth]==WHITE)?(0x80|(p<<4)):(0x08|p));
        if ((castle & p)==p)
        {
            // Make sure squares are empty
            q = y;
            if ((BdLkup((x-1),q)==NO_PIECE) &&
                (BdLkup((x-2),q)==NO_PIECE) &&
                (BdLkup((x-3),q)==NO_PIECE))
            {
                // Test Queen side
                p = (x-2);
                Move(depth,8);
            }
            if (BdLkup((x+1),q)==NO_PIECE &&
                BdLkup((x+2),q)==NO_PIECE)
            {
                // Test King side
                p = (x+2);
                Move(depth,4);
            }
        }
    }
    sp = x+2;
    sq = y+2;
    // Test for normal move
    for (p=(x-1); p!=sp; p++)
    {
        for (q=(y-1); q!=sq; q++)
            Move(depth,2);
    }
}
//-----------------------------------------------------------------------------
static  void    Pawn(Byte depth)
{
    Byte    pawnF,stop;

    stop = 0;
    pawnF = 1;  // Test captures first
    p = x+1; q = y;
    if (Side[depth]==WHITE)
    {
        q++;
        Move(depth,pawnF);
        p -= 2;
        Move(depth,pawnF);
        p = x; pawnF = 0;
        stop = Move(depth,pawnF);
        if (stop==0 && y==1)
        {
            q++;
            Move(depth,pawnF);
        }
    }
    else
    {
        q--;
        Move(depth,pawnF);
        p -= 2;
        Move(depth,pawnF);
        p = x; pawnF = 0;
        stop = Move(depth,pawnF);
        if (stop==0 && y==6)
        {
            q--;
            Move(depth,pawnF);
        }
    }
}
//-----------------------------------------------------------------------------
static  Byte    FindKing(TSide side)
{
    Byte    square,piece;

    for (square=0; square<64; square++)
    {
        piece = BdLkup(Col(square),Row(square));
        if ((piece>>3)==side && (piece&7)==KING)
            break;
    }
    return  square;
}
//-----------------------------------------------------------------------------
static  BOOL    IsKingInCheck(TSide side)
{
    ok = FALSE;
    // Make sure Side's King is NOT in check
    x2 = FindKing(side);
    // Make sure NOT in check
    function = 0;
    y2 = Row(x2);  x2 = Col(x2);
    for (x1=0; x1<64; x1++)
    {
        y1 = BdLkup(Col(x1),Row(x1));
        if (y1 && (y1>>3)!=side)
        {
            x = Col(x1);
            y = Row(x1);
            Side[0] = (side==WHITE)?BLACK:WHITE;
            MoveGen(0,y1);
            if (ok) // King in check then break!
                break;
        }
    }
    function = 2;
    return  (ok!=FALSE);
}
//---------------------------------------------------------------------------
static  void    CheckMove(TChess* pChess)
{
    Byte    piece;

    x = pChess->x1;
    y = pChess->y1;
    Side[pChess->depth] = (((piece=BdLkup(x,y))>>3)&WHITE);
    MoveGen(pChess->depth,piece);
}
//---------------------------------------------------------------------------
static  Byte    MinMax(Byte depth)
{
    Byte    i,piece;

    // Is this as deep we want to go OR has a king been checked?
       if (depth==0 || kingF)
        return  pieceValue;
    // Shall we Prune the search tree
    if (PruneTree)
    {
        if ((Side[depth]==WHITE) && (pieceValue>=BestScore[depth+1]))
            return  pieceValue;
        if ((Side[depth]==BLACK) && (pieceValue<=BestScore[depth+1]))
            return  pieceValue;
    }
    // Does the user wish to force the computer to finish the current move?
    if (forcePlay && (BestScore[depth]!=(Side[depth]==WHITE)?1:255))
        return  pieceValue;
       Chess_LedStat(status);
    // Initialise best node's score
    BestScore[depth] = (Side[depth]==WHITE)?1:255;
    // Find each piece and generate moves
    x = 4;
    for (i=0; i<8; i++)
    {
        x = (i & 1)?(x-i):(x+i);    // gen 43526170 seq, encourage center moves
        for (y=0; y<8; y++)
        {
            piece = BdLkup(x,y);
            if (piece!=NO_PIECE && (piece>>3)==Side[depth])
                MoveGen(depth,piece);
        }
    }
    return  BestScore[depth];
}
//---------------------------------------------------------------------------
static  void    DoChess(TChess* pChess)
{
    x2 = pChess->x2;
    y2 = pChess->y2;
    ok = FALSE;
    // Are we just checking a move?
    if (pChess->depth==0)
    {
        function = 0;
        CheckMove(pChess);
    }
    else
    {
        // Do computers move
        BestScore[pChess->depth+1] = (pChess->side==WHITE)?255:1;
        for (t2=pChess->depth; ; --t2)
        {
            Side[t2] = pChess->side;
            MoveOk[t2] = FALSE;
            // Initialise best node score
            BestScore[t2] = (pChess->side==WHITE)?1:255;
            if (!t2)
                break;
            pChess->side = (pChess->side ^ WHITE);
        }
        function = 2;
        pieceValue = 128;
        t1 = 0;
        kingF = FALSE;
        forcePlay = 0;
        MinMax(pChess->depth);
        pChess->x1 = BestX[pChess->depth];
        pChess->y1 = BestY[pChess->depth];
        pChess->x2 = BestP[pChess->depth];
        pChess->y2 = BestQ[pChess->depth];
    }
    pChess->ok = ok;
}
//------------------------------------------------------------------------------
static  void   CheckPromotePawn(Byte to,TSide side)
{
        // Promote PAWNs to QUEEN if other side reached
        if (PieceDefn(to)==PAWN)
        {
                if (to<8 && side==BLACK)
                        SetPiece(to,B_QUEEN);
                if (to>55 && side==WHITE)
                        SetPiece(to,W_QUEEN);
        }
}
//------------------------------------------------------------------------------
static  void   CheckPerformCastle(Byte from,Byte to,TSide side)
{
        Byte    sq;

        // ONLY can castle one...adjusts the castle flag
        switch (PieceDefn(to))
        {
        case KING:
                castle &= (side==WHITE)?0x0F:0xF0;
                if (AbsDiff(from,to)==2)
                {
                        // AutoMove ROOK into place
                        if (to>from)
                        {
                                SetPiece((to-1),GetPiece(to+1));
                                SetPiece((to+1),NO_PIECE);
                                from = to+1;
                                to--;
                        }
                        else
                        {
                                SetPiece((to+1),GetPiece(to-2));
                                SetPiece((to-2),NO_PIECE);
                                from = to-2;
                                to++;
                        }
                        Chess_Beep();
                        for (sq=Chess_ScanBoard(from,STATUS);sq!=from;)
                        {
                                Chess_Beep();
                                Chess_Beep();
                                sq = Chess_ScanBoard(from,STATUS);
                        }
                        for (sq=Chess_ScanBoard(to,STATUS);sq!=to;)
                        {
                                Chess_Beep();
                                Chess_Beep();
                                sq = Chess_ScanBoard(to,STATUS);
                        }
                }
                break;
        case ROOK:
                if (Col(from)==0)
                        castle &= (side==WHITE)?0xDF:0xFD;
                if (Col(from)==7)
                        castle &= (side==WHITE)?0xBF:0xFB;
                break;
        }
}

/*
```

```
        Display move on LCD
*/

static  void    ShowMoveLCD(Byte from,Byte to,Byte piece)
{
        static  const   char    PieceName[7][8] =       {

                {"       "},

                {"Pawn   "},

                {"Knight "},

                {"Bishop "},

                {"Rook   "},

                {"Queen  "},

                {"King   "}

        };

        char    move[30];
        char*   pMove = move;
        char*   pChar;

        LCD_clear();
        pChar = ((piece>>3)==WHITE)?"Wht ":"Blk ";
        while (*pChar)
                *pMove++ = *pChar++;
        pChar = PieceName[piece&7];
        while (*pChar)
                *pMove++ = *pChar++;
        *pMove++ = ('A'+RANK(from));
        *pMove++ = ('1'+FILE(from));
        *pMove++ = ('A'+RANK(to));
        *pMove++ = ('1'+FILE(to));
        *pMove++ = 0;
        LCD_str(move);
}
```

4 DRIVERS.C

```
/*
Program:        Hardware interface drivers
Author:         H3215
*/

#include <machine.h>
#include "iodefine.h"
#include "edk3687def.h"

#include "drivers.h"

#define LCD_RS 1
#define LCD_RW 2
#define LCD_E  3

/*
        Software timing loops
*/
void delay5ms(void)
{
        long ulDelay;

          for(ulDelay=0; ulDelay<400000; ulDelay++);
}

void delay1us(void)
{
        int i;

        for(i=0;i<1000;i++);
```

```
}

/*
        Wait for the LCD busy flag to be clear
*/
void LCD_wait_BF(void)
{
        unsigned char x;

        P_PORT.PCR3.BYTE = 0xF;       // Data bus is input

        P_PORT.PDR3.BYTE = 1<<LCD_RW; // Select R/W=1, RS = 0, E = 0

        delay1us();

        do
        {

        P_PORT.PDR3.BYTE = (1<<LCD_RW)|(1<<LCD_E);

        delay1us();

        x=P_PORT.PDR3.BYTE ;

        P_PORT.PDR3.BYTE = 1<<LCD_RW; // Select R/W=1, RS = 0, E = 0

        delay1us();

        } while(x & 128);

}

/*
        Write nibble to LCD
*/
void LCD_write(unsigned char RS, unsigned char x)
{

        P_PORT.PCR3.BYTE = 0xFF;              // Data bus is output

        // R/W=0, load data, RS
        P_PORT.PDR3.BYTE = (RS<<LCD_RS) | ((x << 4) & 0xF0);

        delay1us();

        P_PORT.PDR3.BYTE |= 1<<LCD_E;

        delay1us();

        P_PORT.PDR3.BYTE &= ~(1<<LCD_E);

        delay1us();
}

/*
        Write byte to LCD
*/
void LCD_write8(unsigned char RS,unsigned char x)
{

        LCD_write(RS,x >> 4);
        LCD_write(RS,x & 15);

        LCD_wait_BF();

}

/*
        Initialise the LCD
*/
void LCD_init(void)
{

        delay5ms();
        delay5ms();
        delay5ms();
        LCD_write(0,3);

        delay5ms();

        LCD_write(0,3);

        delay5ms();

        LCD_write(0, 3);

        LCD_write(0,2);

        LCD_wait_BF();

        LCD_write8(0, 32+8);   // 2 lines, 5x8 font

        LCD_write8(0, 8);             // display mode off

        LCD_write8(0, 1);             // clear

        LCD_write8(0, 4+2);           // entry mode advance

        LCD_write8(0, 8+4);           // display mode on
}

/*
        Position cursor
*/
void LCD_at(char x, char y)
{
        LCD_write8(0, 128+(x&63)+((y&1)<<6));
}

/*
        Draw a ASCII character on display
*/
void LCD_char(char x)
{
        LCD_write8(1, x);
}

/*
        Draw a null terminated string on display
*/
void LCD_str(char *s)
{
        while(*s)
                LCD_char(*s++);
}

/*
        Clear screen
*/
void LCD_clear(void)
{
        LCD_write8(0, 1);
}

static const unsigned char mask[8] = { 1,2,4,8,16,32,64,128 };

/*
        Control LEDs underneath LCD
*/
void SetLED(unsigned char LED, unsigned char is_on)
{

        if(is_on)
                P_PORT.PDR6.BYTE |= mask[LED & 7];
        else
                P_PORT.PDR6.BYTE &= ~mask[LED & 7];

}


/*
        Scan a button state. Returns true if pressed.
*/
```

```
unsigned char ScanButton(unsigned char button)
{
        int i,x;

        x=0;

        for(i=0;i<255;i++)
        {
                if(P_PORT.PDR5.BYTE & mask[button & 15])
                        x++;
        }

        if(x > 127)
                return 0;
        else
                return 1;
}

/*
        Initialise hardware registers
*/
void InitDrivers(void)
{
        P_TMRZ.TFCR.BIT.CMD = 0;

        P_TMRZ.TOCR.BYTE = 0;

        P_TMRZ.TPMR.BYTE = 0;

        P_TMRZ.TOER.BYTE = 0xFF;

        P_PORT.PCR6.BYTE       = 0xFF;
        /* Sets the bits in the LED Port DDR to O/P */
        P_PORT.PDR6.BYTE       = 0x00;
        /* Sets the bits in the LED Port DR 0x00 */

        P_PORT.PMR5.BYTE = 0;      // Port5 is general purpose I/O
        P_PORT.PCR5.BYTE = 0xF0;  // Port5 lower nibble = input, upper nibble = output

        P_PORT.PUCR5.BYTE = 0xF;  // Enable pullups on switch inputs

        P_PORT.PCR3.BYTE = 0xF;   // LCD control lines are output, data is input
        P_PORT.PDR3.BYTE = 0x0;   // Deselect LCD
        P_PORT.PCR1.BYTE = 0xFF;  // Port 1 set to output
        P_PORT.PDR1.BYTE = 0x00;

        LCD_init();

}

/*
        Control the buzzer on the chess board
*/
void Chess_Beep(void)
{
        unsigned long  i;

        P_PORT.PDR1.BYTE = CHESS_BUZZER;
        for (i=0; i<600000; i++);
        P_PORT.PDR1.BYTE = 0x00;
        for (i=0; i<600000; i++);
}

#define LED_BRIGHTNESS 120

/*
        Control the chess board membrane X generator
*/
void Chess_LedX(char x)
{
        P_PORT.PCR5.BYTE = 0xFF;
        P_PORT.PUCR5.BYTE = 0x00;
        P_PORT.PDR5.BYTE = x;
        P_PORT.PDR1.BYTE = CHESS_LEDX;
        for (x=0; x<LED_BRIGHTNESS; x++);
        P_PORT.PDR1.BYTE = 0x00;
        for (x=0; x<LED_BRIGHTNESS; x++);
}

/*
        Control the chess board membrane Y generator
*/
void Chess_LedY(char y)
{
        P_PORT.PDR6.BYTE = y;
        P_PORT.PDR1.BYTE = CHESS_LEDY;
        for (y=0; y<LED_BRIGHTNESS; y++);
        P_PORT.PDR1.BYTE = 0x00;
        for (y=0; y<LED_BRIGHTNESS; y++);
}

/*
        Read the membrane state at the X,Y position
*/
void Chess_LedStat(char status)
{
        if (status!=-1)
                P_PORT.PDR6.BYTE = status;
        P_PORT.PDR1.BYTE = CHESS_LEDSTAT;
        for (status=0; status<LED_BRIGHTNESS; status++);
        P_PORT.PDR1.BYTE = 0x00;
        for (status=0; status<LED_BRIGHTNESS; status++);
}

/*
        Scan the membrane
*/
char Chess_ScanBoard(char position,char status)
{
        static  const   char    Mask[] = {1,2,4,8,16,32,64,128};
        char    x;
        char    row,column=1;

        for (;;)
        {
                if (position!=-1)
                {
                        Chess_LedX(1<<RANK(position));
                        Chess_LedY(1<<FILE(position));
                }
                Chess_LedStat(status);
                P_PORT.PCR6.BYTE = 0x00;
                P_PORT.PDR5.BYTE = column;
                for (x=0; x<LED_BRIGHTNESS; x++)
                        row = P_PORT.PDR6.BYTE;
                P_PORT.PCR6.BYTE = 0xFF;
                if (row!=0)
                {
                        Chess_Beep();
                        break;
                }
                column *= 2;
                if (column==0)
                        column = 1;
        }
        // Calculate Row Index
        for (x=0; x<8; x++)
        {
                if (row==Mask[x])
                        break;
        }
        row = x;
        // Calculate Column Index
        for (x=0; x<8; x++)
        {
                if (column==Mask[x])
                        break;
        }
        column = x;
        return  (8*row+column);
}
```

5 DRIVERS.H

```
/*
Program:       H8 evaluation board hardware drivers
Author:        H3215
*/

#define CHESS_BUZZER   0x01
#define CHESS_LEDX     0x10
#define CHESS_LEDY     0x04
#define CHESS_LEDSTAT  0x40
#define CHESS_KEYPAD   0x20

#define CHESS_WHITE    0x01
#define CHESS_BLACK    0x02
#define CHESS_CHECK    0x04
#define CHESS_MATE     0x08

#define RANK(X)        (X%8)
#define FILE(X)        (X/8)

//
// Initialisation
//
void InitDrivers(void);

//
// Control LEDs
//
void SetLED(unsigned char LED, unsigned char is_on);

//
// Scan buttons
//
unsigned char ScanButton(unsigned char Button_No);

//
// 16x2 LCD interface
//
void LCD_at(char x, char y);
void LCD_char(char c);
void LCD_str(char *s);
void LCD_clear(void);

//
// Chess membrane interface
//
void Chess_Beep(void);
void Chess_LedX(char x);
void Chess_LedY(char y);
void Chess_LedStat(char status);
char Chess_ScanBoard(char position,char status);
void ChessMain(void);
```

6 MAIN.C

```
/*
Program:       Main line
Author:        H3215
*/

#include <machine.h>
#include "iodefine.h"
#include "edk3687def.h"
#include "drivers.h"
#include "sci.h"

struct SCI_Init_Params SCI_Init_Data={B57600,P_NONE,1,8};

void main(void)
{
        InitSCI(SCI_Init_Data);

        InitDrivers();

        ChessMain();
}
```