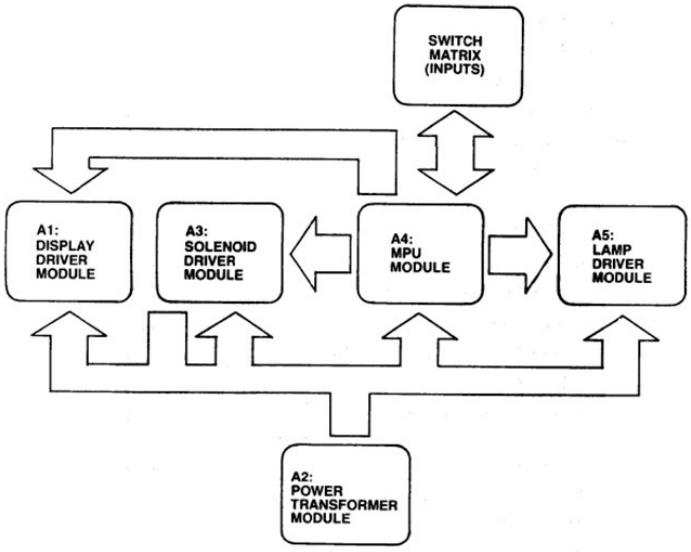
CSCD 462 – Embedded Real-Time Systems

Module 3: Anatomy of a Pinball Machine

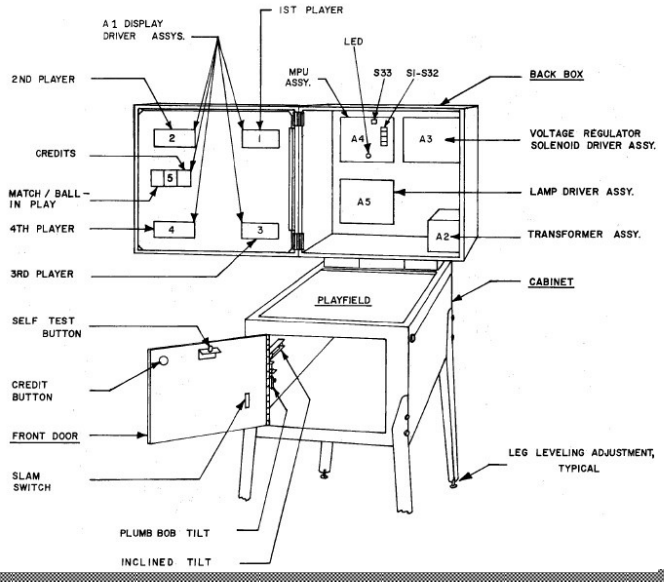
Why Pinball?

- **There will be two midterm and one final project on programming the pinball machine**
  - **why? Because those exercises encapsulate many common problems in embedded systems programming**
    - reading switches and switch matrices, de-bouncing switches in S/W, controlling power devices (solenoids and lamps), interrupt service routines, time multiplexing of outputs
  - **I will provide you with an SDK (API?) that handles much of the detail in this module, including the necessary ISRs**
    - but only after the deadline for Project 1, which requires you to do some low-level programming for the score displays
  - **should you use ARTK for this project? It's not necessary, and I do not recommend it (we'll discuss this more later)**
- **My intent is that you work in teams of two**
  - **feel free to find your own partners**
  - **let me know if you need help finding a partner before week 4**
  - **you will have to schedule time working with the machine (in CEB 101)**

Typical Overall Block Diagram



Where do these Circuits Live?

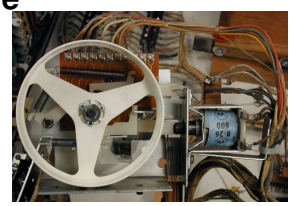


MPU (CPU) Board

- There are 85 solid state pinball machines from two companies that have identical interfaces to/from the CPU control board
  - Stern stole the MPU design from Bally
  - Bally knew this, but waited until Stern was well into production before suing
  - they then collected a royalty for every machine Stern produced
- These CPU boards used the Motorola 6800
  - an 8-bit CPU that is still in production
  - but it is impossible to obtain an in-circuit emulator (ICE)
  - the only way to reprogram it would be to burn EPROMs, plug them in, hit reset, and see what happens
  - so I've created a replacement CPU board that uses the Arduino Mega
  - as you know by now, the Arduino is pretty simple to program

Score Displays

- In the old days score displays were simple number reels
  - which were ratcheted by briefly activating a solenoid
  - H/W took care of rollover between digits



- The logic, H/W or S/W, to advance the score was thus simple
  - to add n points, hit the solenoid n times
- Early 7 segment displays used gas plasma
  - the displays have digital components to convert BCD (4-bit binary for 0-9) into segment activations
  - but it takes 190V to light up gas plasma, so the actual tubes were driven by power transistors, similar to the way solenoid coils are driven
  - the power requirements are so high that only one digit out of the six or seven is ever lit at any given time!
  - the S/W must sequence thru the digits fast enough that the eye does not notice this (but a camera will)
  - this *multiplexing* is driven by an interrupt (3.1 msec)

Score Displays

- S/W maintains the current score in memory, and one digit at a time is copied to the displays as a result of an ISR
  - so the software is quite a bit more complicated for this case
- There are retrofits for these gas-plasma displays
  - including LEDs, which operate on 5V, ignoring the 190V
  - they may not need to be time-multiplexed, as the H/W may latch the values that are sent from S/W, depending on how the aftermarket board is designed
  - our machine uses one such aftermarket display for the credit / ball-in-play display
  - but the embedded code still acts as if they are multiplexed, since in general it cannot count on such H/W upgrades being present
- More recent pinballs use 2D dot-matrix displays, with pictures and animations as well as scores

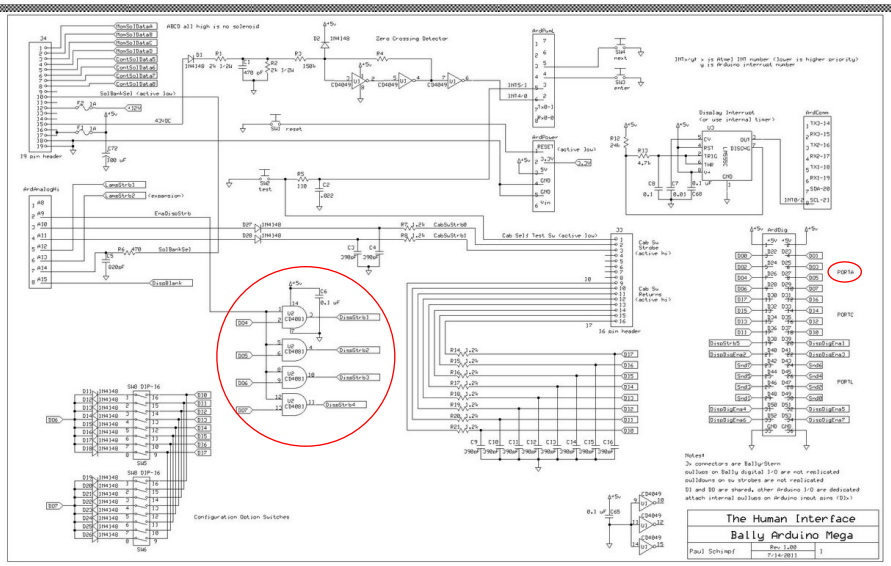
Details of Score Display Control

- On each display interrupt
  - Atmel INT0 = Arduino interrupt #2
  - set the display blanking bit high (A15)
  - set the enable bit for the previous digit low (digits 0:6 are enabled on Arduino pins 39,40,41,50,51,52,53)
  - update the current digit tracking number
  - enable the display strobes by setting pin A9 high
  - for each score display (1-4 players + credit/ball display)
    - set the lower 4 data out bits (PORTA) to the value to display (anything over 9 will result in a blank digit)
    - toggle the display's strobe line high then low (Arduino pins 26,27,28,29,38)
  - disable the display strobes by setting pin A9 low
  - set the enable bit for the new digit high (39,40,41,50,51,52,53)
  - Set the display blanking bit low (A15)

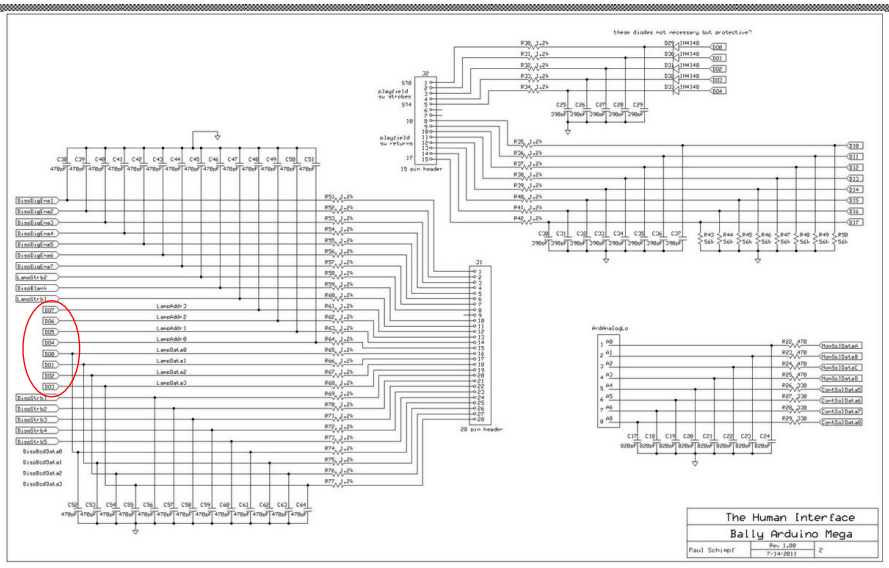
Enable/Disable Display Strobes?

- You won't find the blue steps in the original Bally assembly code, or the Bally Theory of Operation Document
  - because they come about as part of the Arduino interface board
- Arduino PortA (pins 29:22) is used for multiple purposes
  - Lamp addresses and data (upper and lower nibble, respectively)
  - Score Display digit data (BCD, in the lower nibble)
  - 4 of the 5 Score Display strobes (upper nibble)
- We wouldn't want manipulation of the lamp addresses to strobe the display inputs
  - thus the 4 display strobes are further enabled by Arduino bit A9
  - the 5th display strobe is on a dedicated I/O pin (38)

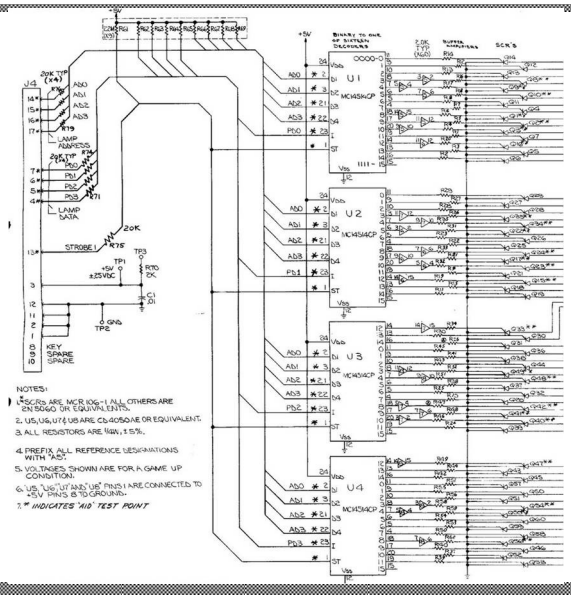
Arduino Bally/Stern Shield Page 1



Arduino Bally/Stern Shield Page 2



Feature Lamp Drivers



- backglass and table lamps are AC
- turned on just after the zero crossing of the AC supply in order to reduce in rush current
- a zero-crossing detector fires an interrupt every 8.3 msec
- similar to solenoid drive, except the logic circuits turn on an SCR, which is similar to a diode, except with a control line
- once triggered, current will flow until it falls below a holding level (at next zero crossing)
- thus lamps that are lit must be re-activated on every interrupt
- (latched lamp control lines, as opposed to address lines could have eliminated that)

Lamp Update Procedure

- **Lamps are updated 4 at a time**
  - **useful to represent data as array of 15 nibbles (15x4)**
- **On zero-crossing interrupt**
  - **Atmel INT4 = Arduino interrupt #0**
  - **for each nibble of 4 lamps (each of 15 rows, or addresses)**
    - load upper nibble of PORTA with the inverse of the lamp state (data)
    - load lower nibble of PORTA with the row number (addr)
    - toggle the lamp strobe line high then low (Arduino pin A12)
  - **load the lower nibble (address) of PORTA with 1111**
  - **toggle the lamp strobe line high then low (A12)**
- **the Bally Theory of Operation manual gives an incorrect explanation for the last 2 steps**
  - **what it actually does is prevent the last 4 lamps (addr 14) from being held on longer than the next zero crossing**

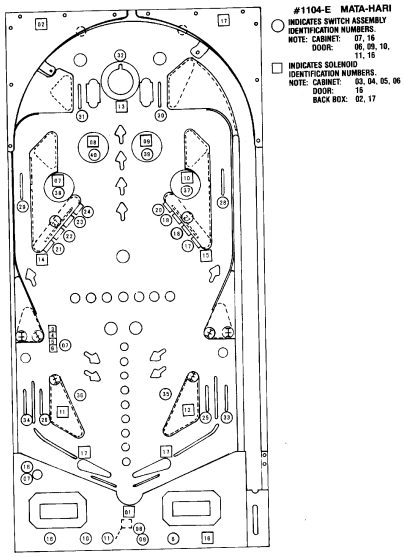
Lamp Map

- **There are lots of lamps**
  - **a full mapping is not usually included in the pinball operations manual**
  - **the built-in diagnostics usually have a test option to flash all lamps, and the operator simply replaces bulbs that are not flashing**
- **Midterm project 2 asks you to write a test routine that allows you to map the lamps for yourself**
  - **you will be allowed to use the SDK for this**
  - **you can make use of the results for your project assignment**
  - **you can check your findings against the pinball schematics if you wish (get schematics from canvas or ipdb and look at the "Lamp Driver" schematic – ask if you need help decoding it)**
  - **please use only the cabinet switches to control your test routine (coin return, credit/start, test switch inside coin door)**
  - **please do NOT attempt to use the CPU board switches**
  - **a partial mapping is provided on the next page**

Mata Hari Partial Lamp Map

Row	Col3	Col2	Col1	Col0	Row	Col3	Col2	Col1	Col0
0	Bonus 4k			Bonus 1k	8			x	x
1					9				x
2					10	Credit Indicator			Shoot Again
3			x	x	11				x
4	x	x	x	x	12		Game Over		
5					13				
6	x				14				Player 1 Up
7			x	x	15	x	x	x	x

Switch, Solenoid Physical Map

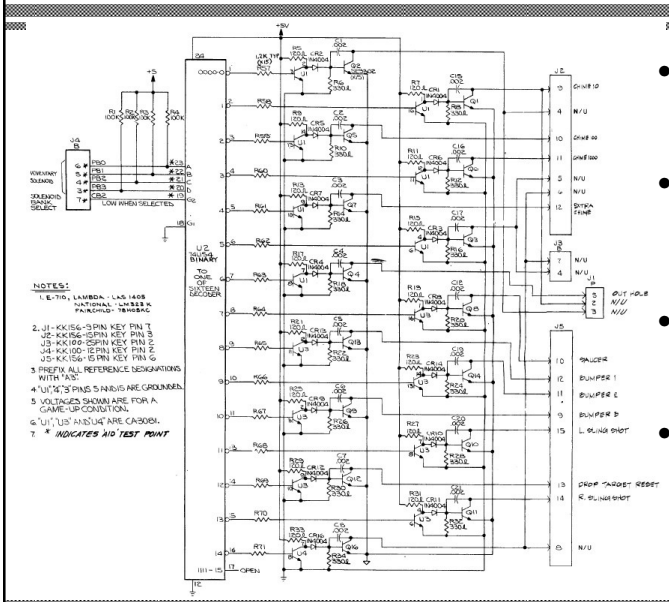


- **The programmer needs to know row,col numbers for the switches**
  - **see slides 30, 31 for Mata Hari**
- **The programmer also needs to know solenoid numbers**
  - **see slides 28, 29 for Mata Hari**

Solenoid Drivers

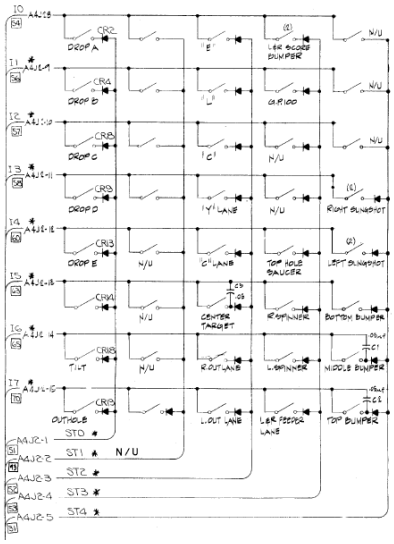
- Solenoids are classified into two groups
- Momentary solenoids
  - only one momentary solenoid may be activated at a time
  - we'll see that the H/W enforces this
  - The Bally Theory of Operation says a momentary solenoid should not be activated for more than roughly 30 msec or it may damage the coil (however, I have found it necessary to push this to 60 msec on the Mata Hari bank target resets)
  - the S/W must enforce this (Bally/Stern S/W tracks this in the zero-crossing ISR, discussed later in conjunction with lamps)
  - this is one of the reasons that I will be providing you with a library – I don't want you burning out the solenoids
  - the library will automatically deactivate a solenoid 30 or 60 msec after your code asks for it to be activated
- Continuous solenoids
  - it is safe to activate these indefinitely, so the SDK will allow that: your code will request both activation and deactivation

Solenoid Driver Circuit



- S/W outputs a 4-bit code indicating the desired solenoid (15 is none)
- the bank select line is then driven low to activate the single decoder outputs
- the single decoder output activates a pair of current drive transistors
- these complete a power circuit thru the solenoid coil, connecting it to ground

Switch Matrix



- **Too many switches to dedicate a digital input line for each**
  - **so they are arranged in a matrix**
  - **the S/W activates one column by putting a logic high on one digital output (ST4:0 = PORTA4:0)**
  - **It then reads a byte-wide digital input (I7:0 = PINC on Arduino)**
  - **a closed switch in the selected column will produce a logic high reading and the position in the byte indicates the row, open switches produce a 0**
  - **there are two such matrices: one for playfield switches and one for cabinet switches (coins, play, tilts, etc)**
  - **plus a few dedicated switches (test, reset, next, enter)**

Reading Playfield Switches

- **Playfield switches are thus read 8 at a time**
  - **the Bally/Stern hardware does NOT debounce switches**
  - **nor does it generate an interrupt when a switch closes**
  - **thus Bally/Stern S/W, as well as ours, must periodically *poll* the switches**
  - **all switches are read during the zero-crossing interrupt, and we look for consistent consecutive readings for switch debouncing**
- **To read a particular row of switches:**
  - **set the corresponding row output high (PORTA4:0)**
  - **wait for 50 usec (allow capacitive discharge from previous row)**
  - **read 8 switch states from PINC**
- **Pretty simple to read the switches**
  - **but they should be debounced before using**

SW Switch Debounce Strategy

- We consider a switch to be no longer bouncing when two consecutive reads agree
- Maintain two arrays for the switch states
  - switchState[nrows]: the last read switch states (8 bits per row)
  - deBounced[nrows]: the latest debounced switch states
  - initialized to 0's
- Suppose we just read the switch states for row i, into an 8-bit variable called temp
  - how do we update these two arrays? A couple hints:

    (it is bouncing if it is now high and was previously low or is now low and was previously high)

    isBouncing = temp <op?> switchState[i]

    (if it is not bouncing copy the state just read; otherwise copy the previous debounced state)

    tempDebounced = f(temp, isBouncing, deBounced[i])

SW Switch Debounce Strategy

- If you read the original Bally assembly code, or the Bally Theory of Operation document, you'll find a somewhat simpler strategy:
  - requires consecutive reads of LOW, HIGH, HIGH, before concluding that a switch is debounced high
  - this debounces only the high state, which is acceptable because all machine actions taken by S/W occur ONLY as a result of a switch being closed (which reads as high, or 1)
  - this is not always the case in an embedded system, nor is it always, or even usually, the case that switch closure reads as a high or 1 (they are more often wired to read 0 when closed)
  - note that the LOW requirement allows Bally S/W to ignore switches that are STUCK high (the machine will still play, but some scores will be missed). It could thus be referred to as a debounced rising edge detection.
  - one downside of the Bally approach is that it requires 3 consecutive reads to conclude that a switch was closed

Rising Edge Detection Strategy

- My BallyLib SDK also *captures* rising edges
  - and maintains them as a sticky state (until the client reads it)
  - it does this just in case the main routine does not get around to reading the high state while it is still high
  - for example, the ball may have rolled off the switch by the time the S/W gets around to checking that switch
  - my SDK offers sticky rising edge detection on both undebounced and debounced switch states
- How does it do that?

  a rising edge has occurred if the new state is high and either the previous state was low or a rising edge was previously recorded)

  debRedge[i] = f(tempDebounced, deBounced[i], debRedge[i])
- A more modern approach:
  - debounce switches with hardware
  - add hardware that generates an interrupt when a switch closes
  - the ISR then reads the switch matrix only when something has changed, as opposed to periodically

BallyLib Overview

#define	description
N_SWITCH_ROWS	number of playfield switch rows
N_SWITCH_COLS	number of playfield switch columns
N_CAB_SW_ROWS	number of cabinet switch rows
N_CAB_SW_COLS	number of cabinet switch columns
N_LAMP_ROWS	number of lamp rows (or addresses)
N_LAMP_COLS	number of lamp columns (data bits)
N_DISPLAYS	number of displays
N_DIGITS	number of digits in each display
N_SOLENOIDS	number of momentary solenoids
N_CONT_SOL	number of continuous solenoids
NEXT	bit mask for the Next button (on CPU board)
ENTER	bit mask for the Enter button (on CPU board)
TEST	bit mask for the Test button (on CPU or coin door)
CREDIT	bit mask for the Credit (Play) button (on coin door)
COIN	bit mask for Coin entry switch (on coin door)

BallyLib Overview

function	description
bool fireSolenoid(int num, bool forcewait, bool doubleDuration=false)	Fire momentary solenoid num for approx 30 msec. If another solenoid is still firing, ignore unless forcewait is true. Returns whether the solenoid was fired.
bool setContSolenoid(int num, bool val)	Set the state of continuous solenoid num to val.
bool setLamp(int row, int col, bool val)	Set lamp at row, col to val
unsigned char getLampRow(int row)	get the 4 lamp states for a particular row
bool setDisplay(int disp, int digit, unsigned char bcdval)	set a particular digit of a particular display
bool getSwitch(int row, int col)	get the last read state of a single switch
bool getDebounced(int row, int col)	get the last debounced state of a single switch
bool getRedge(int row, int col)	Get the rising edge detection of a single switch. Once read, the edge detection is set back to false.
bool getDebRedge(int row, col)	Get the rising edge detection of a debounced switch
unsigned char getSwitchRow(int row)	Get the last state of a row of switches
unsigned char getDebouncedRow(int row)	Get the last debounced state of a row of switches

BallyLib Overview

function	description
unsigned char getRedgeRow(int row)	Get the rising edge state of a row of switches
unsigned char getDebRedgeRow(int row)	Get the rising edge state of a row of debounced switches
void zeroSwitchMemory()	Zero the switch memory, so that any unread sticky rising edges can be ignored (e.g., new game starting)
void setSwitchDelay(int delay)	Set the delay (column settling) time for reading switches, in usec. Defaults to 50 usec.
void setCabSwMonitor(bool onDemand)	If true, the SDK does NOT monitor cabinet switches, and only reads them on demand (when the following function is called). Defaults to false.
bool getCabSwitch(int row, int col)	Get the state of a single Cabinet switch. Note that cabinet switches are NOT debounced, nor are sticky rising edges detected. If onDemand is set, this function currently causes a 9 msec busy delay, so should probably not be called during live play.
int waitForNextEnterTest()	Busy wait until one of the following buttons is pressed: Next, Enter, or Test. These are on the CPU board, but there is also a Test button inside the coin door. The bitwise return value indicates which buttons are pressed. See #defines (slide 24) for the mask codes.

BallyLib Overview

function	description
int waitForTestCreditCoin()	Busy wait until one of the following buttons is pressed: Test, Credit (Start), or Coin. These are all on the coin door (the Test button is on the inside of the door). The bitwise return value indicates which buttons are pressed. See #defines (slide 24) for the mask codes.
int getNextEnterTest()	get the current state of these CPU board buttons. See #defines for the bitmask mask codes.
bool playSound(unsigned char num)	play the indicated sound clip once (1-255, 0 is no sound).

- **Lots of options for reading switches**
  - **read the current state directly**
  - **read the debounced state**
  - **read whether there has been a rising edge on either the debounced or non-debounced state**
  - **read a single switch or row of switches**
- **Note that rising edge detections are sticky, and maintained separately for debounced and non-debounced**
  - **reading one does NOT reset the other**

Mata Hari Momentary Solenoids

Solenoid #	Function
0	Top Center Kick Out Hole (also called the “saucer”)
8	Chime 10
4	Chime 100
12	Chime 1000
2	Chime 10000
10	Knocker
6	Out Hole Kicker (into launch lane)
14	Bottom Left Thumper Bumper
1	Top Left Thumper Bumper
9	Top Right Thumper Bumper
5	Bottom Right Thumper Bumper
13	Left Slingshot
3	Left Drop Target Reset
11	Right Slingshot
7	Right Drop Target Reset

Mata Hari Continuous Solenoids

- **There are only 4 possible with this solenoid driver board**
- **Mata Hari uses only 2**
- **You will need to use 1 of these**
  - **You choose when to enable flippers (at power up is OK)**
  - **PLEASE do not try to tilt or slam this machine**

Solenoid	Function
0	Not Used
1	Coin Lockout
2	Flipper Disable (on by default, must turn off)
3	Not Used

Mata Hari Playfield Switch Map

Row	Col	Switch	Row	Col	Switch
0	6	Tilt	3	3	Right B Lane
0	7	Out Hole	3	4	Left A Lane
2	0	Right Drop Target D (Bottom)	3	5	Top B Lane
2	1	Right Drop Target C	3	6	Top A Lane
2	2	Right Drop Target B	3	7	Top Center Kick Out
2	3	Right Drop Target A (Top)	4	0	Right Out Lane
2	4	Left Drop Target D (Bottom)	4	1	Left Out Lane
2	5	Left Drop Target C	4	2	Right Slingshot
2	6	Left Drop Target B	4	3	Left Slingshot
2	7	Left Drop Target A (Top)	4	4	Bottom Right Pop Bumper
3	0	Right Flipper Feed Lane	4	5	Bottom Left Pop Bumper
3	1	Left Flipper Feed Lane	4	6	Top Right Pop Bumper
3	2	Drop Target Rebound	4	7	Top Left Pop Bumper

Mata Hari Cabinet Switch Map

Row	Col	Switch
0	5	Credit (Start) Button
1	0	Coin III (rewired to Coin Return button)
1	1	Coin I (Not Used)
1	2	Coin II (Not Used)
1	7	Slam

Suggested Initial Design for Final

```
setup:
   initialize serial port (for sending debug messages), blank score
   displays, set any lamps that should always be on
loop:
   each time thru the loop represents one complete game
   init S/W state: scores, player number, ball number, drop target
   counters, any other game and/or ball state variables
   init H/W, such as game over lamp
   wait for credit (play) button to be pressed
   turn off game over, indicate #players=1, init score displays to zero
   loop for each ball (3 balls per player per game)
         zero the switch memory so don't retain sticky hits from before
         init any S/W and H/W state that should reset on each ball
         light current player up and display the ball number
         fire the outhole solenoid to eject a ball
         loop, reading each playfield switch
             for each switch hit, take appropriate action (add player,
             fire solenoid, add points, play chime, arm bonus, etc.)
         until the outlane switch is read
         advance current player and/or ball number
   until each player has played 3 balls
   check for high score (optional)
   perform random score match (optional), fire knocker on match
```

Adding Players is Tricky

- **Note that a ball is ejected after the first press of the credit (play) button**
  - **the standard is to press the credit button to add players**
  - **but once the ball is put into play, then it should no longer be allowed to add more players**
  - **some pinballs may include a switch in the launcher lane to detect when the ball has been launched**
  - **older pinballs, such as this, do not**
- **So how do they detect that the ball has been put into play?**
  - **you need a S/W state variable that is set when the ball is ejected, and cleared by *any* playfield switch**
  - **allow adding players only if that state is set**
  - **on some pinballs the ball will have to travel thru one of a subset of switches at the top of the playfield**
  - **that is not the case here – it could go straight to the outhole**

Adding Players is Tricky

- **This all means you will have to *poll* the credit button as part of the work in the innermost loop, until a playfield switch has been triggered**
  - **the Bally.waitForXxx() routine is inappropriate here, because it is a busy wait**
  - **the Bally.getCabSwitch() does not debounce or provide sticky rising edge detection – it simply provides the raw switch state at the time of the call (within the last interrupt cycle)**
  - **if you simply use the current state of the start button, you are very likely to add multiple players for one press**
  - **thus you will need to apply something like the Bally debounce strategy yourself here (save the last two reads and detect a low, high, high sequence)**
  - **in this case you can get by with just saving the last read, and executing your "add player" code when a low state is followed by a high state**

Some More Hints

- **read switches by row when possible**
  - **the only switches I processed other than by a single row read were the credit switch and the outhole switch**
- **reading the debounced rising edge was my *default* choice**
  - **just in case the switch is no longer depressed by the time you get around to reading it (actually pretty unlikely)**
- **except for active bumpers and slingshots …**
  - **you should read the raw switch state for these, because the time delay of debouncing and edge detection could result in your ball strike coming too late (see homework problem)**
- **the ball sits in the outlane or saucer until you eject it**
  - **so there is no possibility of missing that switch closure**
- **note there are routines that will busy wait for a cabinet switch closure (handy for the initial start)**

