

Guide to the Pinball Command Language

3-2009, Chris Eddy, Copyright 2009

This is a super preliminary listing, as there is a lot to do yet, and this information will surely change as the operating system changes to meet the requirements of game rules.

The **Pinball Command Language** is specifically written to allow the game owner or builder to create and modify a set of rules and load them into a machine that is running the **Pinball Operating System**.

Table of contents:

The thread concept

What is a thread made of?

The thread environment

Future editing and loading of threads

The player context

The machine context

The setup menu function values

How scores work

How switches work

Thread commands

The lamp tag names list

The switches tag names list

The solenoids tag names list

Anatomy of a switch monitor thread

The thread concept

A thread is defined as a group of commands that are executed in a manner that is independent from other threads. Each thread has a separate environment or context. A group of threads are all executed one after the other, and back to the beginning, fast enough that each thread appears to be operating independently. A running pinball machine will likely be executing a number of threads to perform different tasks. For instance, in attract mode, one thread may be monitoring a saucer, and if a ball was left in it, it kicks it out with the solenoid. Another thread is monitoring the coin slots (three slots? Three threads). Another thread or threads may be decorating the playfield with a light show. When a game is started, most or all of these threads may be killed and replaced by other threads that manage audit menus, or game on operations.

A thread can have commands that spawn (add another thread to the list of active threads), kill (remove a thread from the list of active threads), stop (suspend the execution of a thread), or start (allow a stopped thread to begin execution again). Threads have commands that perform math, and keep the results in an accumulator and status flags. Threads also have commands that modify the flow of the thread within itself, such as 'goto' commands, and 'if ***, goto' commands. And of course, there are commands that specifically control items such as lamps, solenoids, score displays, sounds, and the like.

What is a thread made of?

In order to make the Pinball Command Language portable across different hardware solutions, the command language is comprised of strings of constants. Each command is a pair of numbers.. the first number is the specific type of command, and the second is the argument. For instance, if I write one line of a thread command:

```
THCMD_SET_LAMP_ON, LAMP_1_TARGET
```

The operating system knows that these text labels are actually represented by some numeric values. This specific command indicates to the operating system that we want to turn on the lamp for the 1 target. As you grow to understand the commands and how to assemble them, you will see that the language bears a strong resemblance to a microprocessor's assembly language. This is true, but since the constants are interpreted by software, they are not truly an assembly language.

The thread environment

Each thread has an environment. By environment, we mean that each thread has some variables that it alone accesses. Examples are values that link the threads together, so that the operating system can track the order of threads when executing, and also allow the operating system to add and subtract threads from the list. There are variables that keep time in milliseconds and seconds, for use in the thread by the user. There is an 'accumulator' in a similar way to a microprocessor. There are flags such as zero, carry, and borrow, that allow some math operations to be performed and then decisions to be made based on the results. And finally, there are some variables that the user can access when writing the thread that can be used for data or flags.

Future editing and loading of threads

Throughout this manual, threads are shown without regard to how you actually edit and load them. This is due to the fact that the threads are currently written in C, where they are actually incorporated into the Pinball Operating System. The future will see a game programmer editing the threads in a PC program written to manage and display the threads and information, and either a USB cable link to the target board or a flash thumb drive to transfer the information to the target board. This work has yet to be done.

The player context

There are variables in the operating system that hold information about the player, of which there will be more than one. These variables hold information such as which goals have been accomplished in the game.. for instance, I earned the 1 target, and when my turn comes back around, my player context remembers that. I write a thread to take that data back out and update the playfield with it when my turn comes around.

The machine context

There are variables referred to as system variables that relate to the machine as a whole. These values represent items such as which player is up, which ball is in play, how many players are coined up, a queue of earned scores yet to be tallied (more on scores later), and how much bonus has been earned. Note that the last two relate to player information, which one would think belong in player data. But if the data only occurs once, and does not have to be remembered from player to player, then the system area is fine for that.

The setup menu function values

In the setup menus, there are a number of non-volatile values that can be set dealing with everything from coin credit management to game play options. For the most part, they are reflected directly from the original setup scheme. This list does not include the audits data.. the function values start after that (function 18 in system 6 for instance).

How scores work

Scores on machines vary by what generation the machine is from. Some are 6 digit, some are 7, or more. So the pinball operating system keeps a score pad that holds up to 10 million, and stores it in separate digits. The reason that the digits are kept separate is that the operating system does not have to break a big variable down into separate digits every time it goes to display the information.

There is a concept called the score rack. A variable is kept for each digit.. for instance, there is a variable specifically set aside on the rack for 1000's. Frequently, when a score is earned, the score does not necessarily flash all 5000 on the display at once. There may be 5 separate issues of 1000 points, and a sound each time. To keep the threads easier to understand, the rack concept was implemented. One thread, when a goal is earned, will add say 5 to the 1000's place in the rack. Another thread is solely responsible for monitoring the rack, and when a value appears, doling out the score to the display and sound system one at a time with appropriate delays. In this manner, scores are earned and displayed.

How switches work

Switches on a machine are scanned or multiplexed by an interrupt at a specific timed rate. Each switch has a variable to keep track of how many times that that switch has been opened or closed. The game programmer will want different switches to react with different timing. For instance, a rollover will have to operate quickly, since the ball is moving quite quickly. But the outhole, for instance, may need a slower reaction, as a false trigger there will erroneously end the ball for the player and confuse game play enormously. A ball save will want a medium response to closure, but a long response to recognize open, as it takes time for the solenoid to clear the hole. For the whole game, there is a table of switch closure thresholds, and a table of switch open thresholds. These values are used as a comparison when determining if a switch is open or closed. The tables are currently coded into the operating system, but I am considering some commands that the game programmer can override the defaults with.

Thread commands: a glossary of commands.

The increment column represents what happens to the program counter within each thread. Most commands will increment the program counter by one, such that the next command will execute. But a wait for switch close command does not want to increment the program counter until the event occurs, so the thread will repeatedly come back to that command upon every run. The flags are kept to keep track of the results of arithmetic and math events, and to allow a subsequent command such as IFGE to choose an action based on those flags.

Command name:	Purpose of command:	Increment program counter?	Affects flags?
THCMD_SPAWN_THREAD	Add a thread to the execution list	+1	no
THCMD_KILL_THREAD	Delete a thread from the execution list	+1	no
THCMD_START_THREAD	Permit a thread to execute (from stop)	+1	no
THCMD_STOP_THREAD	Halt a thread from executing	+1	no
THCMD_KILL_THIS_THREAD	Delete this thread from the exec list	N/A	N/A
THCMD_KILL_THREAD_TYPE	Delete all of a type from the exec list	+1	no
THCMD_SET_THREAD_TYPE	Set the type of this thread	+1	no
THCMD_WAIT_SW_CLOSE	Wait for given switch to close	If closed	no
THCMD_WAIT_SW_OPEN,	Wait for given switch to open	If open	no
THCMD_WAIT_SW_HELD	Wait for a given switch to be held for X	If held	no
THCMD_CHECK_SW_CLOSE	Check switch closure, update accumulator with results	+1	no
THCMD_CHECK_SW_OPEN	Check switch open, update accumulator with results	+1	no
THCMD_SET_LAMP_ON	Set the given lamp to on state	+1	no
THCMD_SET_LAMP_OFF	Set the given lamp to off state	+1	no
THCMD_SET_LAMP_FLASH	Set the lamp to the flash flag.. transparent, must exec over and over.	+1	no
THCMD_CLEAR_LAMPS_ALL	Set all lamps to off state	+1	no
THCMD_SELECT_SOL	Select a solenoid for next operation	+1	no
THCMD_SET_SOL_ON	Set selected solenoid to on state (danger.. use sparingly)	+1	no
THCMD_SET_SOL_TIME	Set selected solenoid to on state for specified milliseconds	+1	no
THCMD_SET_SOL_OFF	Set selected solenoid to off state	+1	no
THCMD_SET_DELAY_mS	Set a counter to the given milliseconds	+1	no
THCMD_WAIT_DELAY_mS	Wait for the millisecond counter to expire	If expired	no

THCMD_SET_DELAY_SEC	Set a counter to the given seconds	+1	no
THCMD_WAIT_DELAY_SEC	Wait for the second counter to expire	If expired	no
THCMD_SCORE_CLEAR	Clear the score for the given player	+1	no
THCMD_SET_CREDITS_DISPLAY	Set the credit display to the accumulator value	+1	no
THCMD_SET_BALL_DISPLAY	Set the ball display to the accumulator value	+1	no
THCMD_PROCESS_COIN	Register a credit given the specified slot	+1	no
THCMD_GOTO_LINE	Go directly to the line specified (+/- offset)	+/- given	no
THCMD_LINE_TAG	Identify a line with a number for a goto target	+1	no
THCMD_LOAD_BYTE_NUMB	Load a numeric number into the accumulator	+1	Z
THCMD_SUBTRACT_BYTE_NUMB	Subtract number from accumulator	+1	Z, B
THCMD_STORE_BYTE_PDATA	Store the accumulator in a given player data location	+1	no
THCMD_CLEAR_BYTE_PDATA	Clear a given player data location	+1	no
THCMD_SET_BYTE_PDATA	Set a given player data location to 1	+1	no
THCMD_INC_BYTE_PDATA	Increment a given player data location by 1	+1	no
THCMD_DEC_BYTE_PDATA	Decrement a given player data location by 1	+1	no
THCMD_SUBTRACT_BYTE_PDATA	Subtract a given player data location from the accumulator	+1	Z, B
THCMD_LOAD_BYTE_PDATA	Load a given player data location into the accumulator	+1	no
THCMD_STORE_BYTE_LAMP	Store whatever is in the accumulator into the given lamp position.	+1	no
THCMD_LOAD_BYTE_FDATA	Load a given function value location into the accumulator	+1	no
THCMD_SUBTRACT_BYTE_FDATA	Subtract a given function value location from the accumulator	+1	Z, B
THCMD_STORE_BYTE_SDATA	Store whatever is in the accumulator into the given system variable location.	+1	no
THCMD_CLEAR_BYTE_SDATA	Clear a given system variable location	+1	no
THCMD_SET_BYTE_SDATA	Set a given system variable location to 1	+1	no
THCMD_INC_BYTE_SDATA	Decrement a given system variable location by 1	+1	no

THCMD_DEC_BYTE_SDATA	Increment a given system variable location by 1	+1	no
THCMD_SUBTRACT_BYTE_SDATA	Subtract a given system variable location from the accumulator	+1	Z, B
THCMD_LOAD_BYTE_SDATA	Load a given system variable location into the accumulator	+1	no
THCMD_GOTO_IFGE	Goto offset if the accumulator was greater than or equal to the argument	+1 or offset	no
THCMD_GOTO_IFGT	Goto offset if the accumulator was greater than the argument	+1 or offset	no
THCMD_GOTO_IFEQ	Goto offset if the accumulator was equal to the argument	+1 or offset	no
THCMD_GOTO_IFLT	Goto offset if the accumulator was less than the argument	+1 or offset	no
THCMD_GOTO_IFLE	Goto offset if the accumulator was less than or equal to the argument	+1 or offset	no
THCMD_GOTO_IFNE	Goto offset if the accumulator was not equal to the argument	+1 or offset	no
THCMD_SET_SOUND	Send the given sound code to the sound board	+1	no
THCMD_INIT_PLAYERDATA	Clear all variables in the given player's data set	+1	no
THCMD_INC_SCORE_BY_1	Add one to the ones score rack for the player that is up	+1	no
THCMD_INC_SCORE_BY_10	Add one to the tens score rack for the player that is up	+1	no
THCMD_INC_SCORE_BY_100	Add one to the hundreds score rack for the player that is up	+1	no
THCMD_INC_SCORE_BY_1000	Add one to the thousands score rack for the player that is up	+1	no
THCMD_INC_SCORE_BY_10K	Add one to the 10K score rack for the player that is up	+1	no
THCMD_INC_SCORE_BY_100K	Add one to the 100K score rack for the player that is up	+1	no
THCMD_INC_SCORE_BY_1M	Add one to the Millions score rack for the player that is up	+1	no
THCMD_INC_SCORE_BY_10M	Add one to the 10 Millions score rack for the player that is up	+1	no

LAMP_TILT, LAMP_GAME_OVER, LAMP_SAME_PLAYER_AGAIN,
LAMP_HIGH_SCORE_TODATE

The switches tag names list

For any given game, there are a multiplexed list of switches available to the software. The operating system recognizes them as switches number 1 to N, where for instance in systems 3/4/6/7, N is 64. A table of tag names is built by the user to represent the names of the switches, so that the game programmer does not have to use numbers to represent the switch. The following table is an example of the tag name list for Firepower.

SW_PLUMB_BOB, SW_BALL_ROLL_TILT, SW_CREDIT_BUTTON, SW_RIGHT_COIN,
SW_CENTER_COIN, SW_LEFT_COIN, SW_SLAM_TILT, SW_HIGH_SCORE_RESET,
SW_OUTHOLE, SW_LEFT_OUTSIDE_ROLLOVER, SW_LEFT_INSIDE_ROLLOVER,
SW_LEFT_KICKER, SW_LEFT_EJECT_HOLE, SW_UPPER_MIDDLE_LEFT_STANDUP,
SW_SPINNER, SW_TOP_LEFT_STANDUP,
SW_1_TARGET, SW_2_TARGET, SW_3_TARGET, SW_NOTUSED_20, SW_4_TARGET,
SW_5_TARGET, SW_6_TARGET, SW_NOTUSED_24,
SW_BOTTOM_LEFT BUMPER, SW_TOP_LEFT BUMPER, SW_TOP_RIGHT BUMPER,
SW_BOTTOM_RIGHT BUMPER, SW_TOP_CENTER_TARGET, SW_RIGHT_EJECT_HOLE,
SW_UPPER_TOP_RIGHT_STANDUP, SW_F_TARGET,
SW_I_TARGET, SW_R_TARGET, SW_E_TARGET, SW_UPPER_RIGHT_EJECT_HOLE,
SW_LOWER_TOP_RIGHT_STANDUP, SW_MIDDLE_RIGHT_STANDUP,
SW_TOP_POWER_TARGET, SW_CENTER_POWER_TARGET,
SW_BOTTOM_POWER_TARGET, SW_RIGHT_KICKER, SW_RIGHT_INSIDE_ROLLOVER,
SW_RIGHT_OUTSIDE_ROLLOVER, SW_LANE_CHANGE, SW_BALL_SHOOTER,
SW_PLAYFIELD_TILT, SW_LOWER_RIGHT_STANDUP,
SW_CENTER_MIDDLE_LEFT_STANDUP, SW_LOWER_MIDDLE_LEFT_STANDUP,
SW_LEFT_BALL_RAMP, SW_NOTUSED_52, SW_LEFT_EJECT_ROLLOVER,
SW_RIGHT_EJECT_ROLLOVER, SW_NOTUSED_55, SW_NOTUSED_56,
SW_RIGHT_BALL_RAMP, SW_CENTER_BALL_RAMP, SW_NOTUSED_59,
SW_NOTUSED_60, SW_NOTUSED_61, SW_NOTUSED_62, SW_NOTUSED_63,
SW_NOTUSED_64

The solenoids tag names list

For any given game, there are a number of direct controlled and special controlled solenoids. The software access them by tag names that represent the number of the solenoid, so that the game programmer does not have to use numbers to represent the solenoids. The following table is an example of the tag name list for Firepower.

SOL_BALL_RELEASE, SOL_NOTUSED1, SOL_NOTUSED2, SOL_LEFT_EJECT_HOLE,
SOL_RIGHT_EJECT_HOLE, SOL_UPPER_RIGHT_EJECT_HOLE,
SOL_LEFT_BALL_SAVE_KICKER, SOL_BALL_RAMP_THROWER,
SOL_SOUND1, SOL_SOUND2, SOL_SOUND3, SOL_SOUND4, SOL_SOUND5,
SOL_CREDIT_KNOCKER, SOL_FLASH_LAMPS, SOL_COIN_LOCKOUT,
SOL_TOP_LEFT BUMPER, SOL_BOTTOM_LEFT BUMPER, SOL_TOP_RIGHT BUMPER,
SOL_BOTTOM_RIGHT BUMPER, SOL_RIGHT_KICKER, SOL_LEFT_KICKER,
SOL_FLIP_ENABLE

Anatomy of a switch monitor thread

Listed here is a typical thread which deals with target 1 on Firepower. It has been simplified for illustration. Note that the future editor will probably display them differently, but the idea will still be the same.

Thread #20 (for instance) // A thread that monitors target 1

```
1:  THCMD_SET_THREAD_TYPE, THREAD_GAME_ON_MASK,
2:  THCMD_WAIT_SW_CLOSE, SW_1_TARGET,
3:  THCMD_INC_BYTE_SDATA, SYSTEMDATA_BONUS,
4:  THCMD_INC_SCORE_BY_1000, 0, // always applied to player that is up
5:  THCMD_SET_BYTE_PDATA, PLAYERDATA_TARGET1,
6:  THCMD_GOTO_LINE, -4
```

Let's start with line 1. Line 1 sets the thread type to `THREAD_GAME_ON_MASK`, which allows other threads to kill entire lists of threads of certain types if necessary.

Line 2 waits for a closure of target switch 1. When a closure occurs, the thread proceeds to the next line.

Line 3 increments the bonus variable held in the system variables. Another thread elsewhere will deal with displaying the current bonus.

Line 4 increments the score by 1000. In reality, it increments the score rack value, which will make sounds and alter the score on the display elsewhere.

Line 5 sets the player's variable that tracks target 1 to a 1. This allows the game to cycle around, and when we return to that player, he still has credit for target 1.

Line 6 simply places the program counter back to line 2, where we will wait once again for a switch closure.

Anatomy of a saucer clearing thread

Listed here is a typical thread which runs during attract mode, which if it finds a ball left in the save saucer, will clear the ball.

Thread #11 (for instance) // A thread that will clear a saucer if a ball is left in it.

```
1:   THCMD_SET_THREAD_TYPE, THREAD_GMOVER_ATTRACT_MASK
2:   THCMD_WAIT_SW_HELD, SW_LEFT_EJECT_HOLE
3:   THCMD_SELECT_SOL, SOL_LEFT_EJECT_HOLE
4:   THCMD_SET_SOL_TIME, 100
5:   THCMD_SET_DELAY_mS, 200
6:   THCMD_WAIT_DELAY_mS, 0
7:   THCMD_GOTO_LINE, -5
```

Let's start with line 1. Line 1 sets the thread type to `THREAD_GMOVER_ATTRACT_MASK`, which allows other threads to kill entire lists of threads of certain types if necessary.

Line 2 waits for a closure of the switch in the left eject hole. When a closure occurs, the thread proceeds to the next line.

Line 3 selects the solenoid for the left eject hole.

Line 4 sets the selected solenoid to 100 milliseconds of on time. The operating system will then operate the solenoid.

Line 5 sets the delay timer to 200ms. It is important to delay the solenoid action, as the program would wrap to the top so quickly that the ball would still be in the saucer, as the solenoid takes time to operate.

Line 6 waits for the delay timer to expire to 0, and then proceeds.

Line 7 simply places the program counter back to line 2, where we will wait once again for a switch closure.