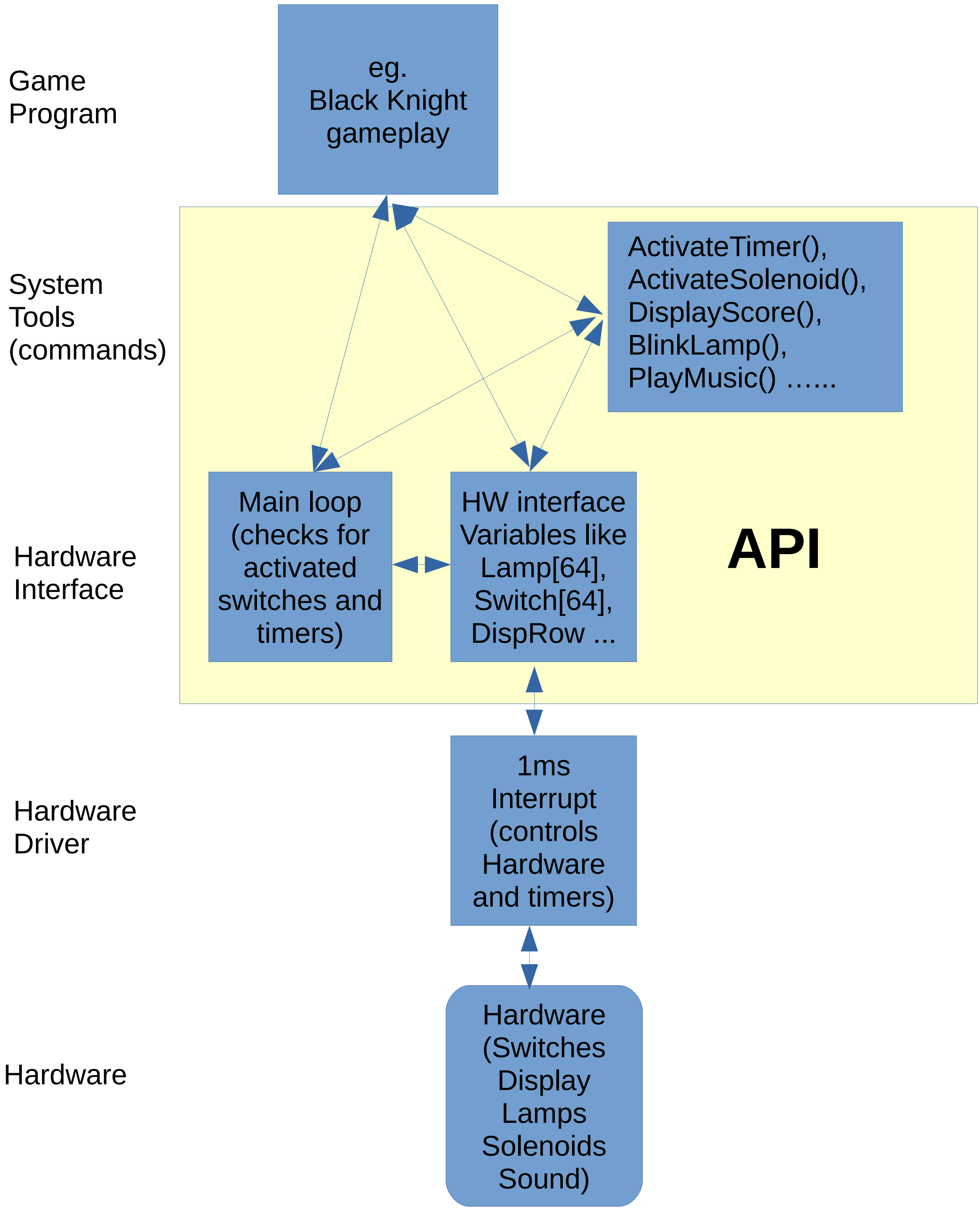
APC Software reference

1. Strukture

The APC software consists mainly of four layers.



Hardware Driver

The hardware driver consists of an interrupt running every ms. This interrupt controls the system bus of the APC which provides access to the various hardware elements.

The timing of this interrupt is important for strobing the display, the lamp and switch matrix. It can easily damage your pinball machine if this interrupt doesn‘t work correctly. So please don‘t mess with it unless you know exactly what you‘re doing!

Hardware Interface

The hardware driver provides several arrays to monitor the status of switches and timers as well as to control the lamps and the display. These arrays are used by the main loop of the program to check for activated switches and timers and call the corresponding subroutines. The loop also reads audio information from the SD card and buffers it for the hardware driver (interrupt) to send it to the DAC.

System tools

Provides several subroutines to control timers, display, music and so on. Together with the hardware interface, the system tools represent the application interface (API) of the APC software (see API section how to use it)

Game Program

All game specific program parts belong to the game program. Normally this is an own .ino file which is located in the same directory as the APC.ino. The compiler will concatenate both files and compile them together. If you want to program the software for a new game, you have to set some variables in the APC.ino to make your new rules selectable in the settings, but after that it shouldn‘t be necessary for you to do any changes in ARC.ino but only in your own game program.

2. How to use the API

What is needed to control a pinball machine? Well, the controller has to query the status of the switches and as a reaction to this it has to change the content of the display, activate solenoids and lamps and play the right sound and/or music at the right time.

To help you understand how the APC software works I'm going to explain these basic functions before proceeding to the quite tedious API section which is meant more as a reference. However, for more information about the mentioned functions, please refer to the tools part of the API reference.

2.1. The inputs (Triggers)

There are only two events that can trigger an event in the APC software: timers and switches having changed their status.

2.1.1. Timers

The timers are best handled by the ActivateTimer() and KillTimer() commands which belong to the system tools and are explained in the corresponding section.

2.1.2. Switches

The status of the switches can be queried by using the switch[] array, but in most cases this is not necessary as the hardware interface will automatically call the procedure to which the pointers Switch_Pressed and Switch_released are referring to, depending on how the status of the switch has changed.

2.2 The outputs

2.2.1. Lamps

Lamps are handled by the array which bool *LampPattern is pointing to. Normally this is the Lamps[] array. If Lamp[x] is true, the lamp is being lit otherwise it's off.
Blinking lamps are controlled by AddBlinkLamp() and RemoveBlinkLamp(). For complex lamp animations there is the function ShowLampPatterns().

2.2.2. Solenoids

In early machines, solenoids are quite simple to use with the functions ActivateSolenoid() and ReleaseSolenoid(). The latter one is rarely needed as the former one is considering the hold time of a certain solenoid.
With System11 things became a bit more complicated as these machines use two banks of solenoids with a relay to switch between them. Therefore it's safer to call the functions ActA_BankSol() and ActC_BankSol() in Systemm1 machines, since they delay the activation of the solenoid until the correct bank has been activated by the relay.

2.2.3. Display

The use of the display is not that trivial as there are various types of displays to support.

3. Hardware interface API

This section lists the variables and routines/commands that are directly controlling the hardware. Basically these are enough to control the pinball.

3.1. Basic variables of the hardware interface

The following table shows the basic variables used to control the APC. They belong directly to the hardware interface. These variables and the commands in the next section are sufficient to control the display, lamps, sound/music, switches and solenoids.
However, there are some system tools built in that are supposed to make using the hardware a bit easier, but all variables that belong to these system tools are explained in the system tools section.

void (*AfterMusic) void (*AfterSound)	Points to the subroutine being executed after the current sound/music has run out. If the value is 0 then no subroutine is called.
byte DisplayUpper[32] byte DisplayLower[32] byte DisplayUpper2[32] byte DisplayLower2[32]	These are buffers for the upper and lower display row which contain the segment patterns to form the characters. There are 2 bytes needed for each character, as every one consists of 14 segments plus dot and comma. For the displays having 16 columns at most, each buffer contains 32 bytes. The pointers *DispRow1 and 2 determine which buffer is used. Normally the routines WriteUpper and WriteLower are used to fill these buffers, since they can be called with normal strings and use the segments patterns which are stored in the DispPattern arrays. Example: If you call WriteUpper("EXAMPLE TEXT "), the WriteUpper routine will use the pointer *DispPattern1 to determine the segment patterns for these characters and write them into the DisplayUpper buffer. Now *DispRow1 has to point to DisplayUpper to let the text appear in the upper row.
byte *DispRow1 byte *DispRow2	Points to the display buffer currently being shown. Normally these are DisplayUpper / DisplayLower or DisplayUpper2 / DisplayLower2, but you can also use your own buffer. For certain display effects it makes sense to have them stored as an array of buffers and just change DispRow1 / 2 to show them one after another.
byte *DispPattern1 byte *DispPattern2	Points to the display segment patterns that have to be used for the type of display currently being used. AlphaLower[] for example stores the segment patterns for all displays that use alphanumeric displays for the lower row. That means if you use an early System11 you have to let *DispPattern2 point to NumLower[] as they use numeric displays for the lower row.

struct GameDef GameDefinition	
struct HighScores HallOfFame	
bool Lamp[65]	Status buffer for the lamps. If *LampPattern points to Lamp[] then a true value will light the corresponding lamp.
bool *LampPattern	Points to the lamp array currently being used. Normally it should point to Lamp[], but it can also point to NoLamps[] which contains only false values and AllLamps[] with all lamps true except of those in the backbox. Some routines also change this pointer to show lamp animations.
bool Switch[72]	Represents the status of the switches. True means the switch is currently closed. Switch[0] is not used as there is no Switch 0 in a pinball machine. The switches up to 64 are those of the switch matrix as listed in the manual of your pinball machine. The switches 65 to 70 are the special solenoid switches. The original Williams controllers up to System 11 use logic to fire the special solenoids as soon as the special solenoid switch is closed. The APC treats special solenoid switches like normal switches and therefore requires the software to activate the corresponding solenoid. Switch[71] is the Memory protect switch Switch[72] is the Advance switch
void (*Switch_Pressed)(byte);	Points to the subroutine being executed when a switch is activated (status changes from released to pressed) with the byte argument being the number of the activated switch. This should point to the switch handler of your game to be able to react to changes without having to query every ship in order to look for changes. Let it point to DummyProcess if not needed.
void (*Switch_Released)(byte);	Same as above only for switches being released.
UpDown	This constant contains the number of the Arduino pin connected to the Up/Down switch. The state of the switch can be determined by digitalRead(UpDown) Note that in this case false means the UpDown button is pressed

3.2 Basic System Tools (commands) to control the hardware

The following table shows the basic commands used to control the APC. They belong directly to the hardware interface. These variables are sufficient to control the display, lamps, sound/music, switches and solenoids.

ActivateSolenoid	(unsigned int Duration, byte Solenoid) Activates the given solenoid for the given duration. If Duration is 0, the default activation time for the given solenoid is used. These default activation times are taken from the SolTimes array which is part of the GameDefinition structure and has to be specified for every game /

	machine.
ActivateTimer	(unsigned int Value, byte Argument, void (*EventPointer)(byte)) Activates a timer for the time given in Value (in ms). After the timer has been run out, the routine which *EventPointer references is called with Argument. The command returns the number of the timer which has been activated. This number can be used to kill the timer with the KillTimer command.
KillTimer	(byte TimerNo) Deactivates TimerNo which is the timer number returned by the ActivateTimer command.
PlaySound / Music	(byte Priority, char* Filename) Starts playing the sound / music with the given filename (filename must not exceed 8 characters plus suffix – e.g. sound.snd). If a sound / music is currently being played it depends on the given priority whether it is stopped for the new one. Priorities can range between 0 and 99. If the given priority is < 100, the new sound / music is played when the given priority is equal or larger than the one of the file currently being played. If the given priority is > 99 then 100 will be subtracted and the result has to be larger than the current priority for the given file to be played.
ReleaseSolenoid	(byte Solenoid) Stops the activation of the given solenoid.
StopPlayingSound /Music	The name says it all. Note that this command will terminate the current playback without calling the AfterMusic / AfterSound command.
SolenoidStatus	(byte Solenoid) Returns the current status of the given solenoid. True means that the solenoid is active.
WriteUpper / Lower WriteUpper2 / Lower2	(char* DisplayText) Uses the patterns referenced by DispPattern1 / 2 to translate the given ASCII text (in bold letters) to the corresponding display segment patterns that can be displayed. WriteUpper2 / Lower2 do the same, they just use DispPattern2 as the target buffer. Hence you have to change the *DispRow pointers to point to the *2 buffers. Don't become confused with *DispRow1 and 2 which are just referencing the two rows used by the average display. Example: If you call WriteUpper("EXAMPLE TEXT "), the WriteUpper routine will use the pointer *DispPattern1 to determine the segment patterns for these characters and write them into the DisplayUpper buffer. Now *DispRow1 has to point to DisplayUpper to let the text appear in the upper row.

4. System Tools (commands)

The most basic system tools have already been described in the previous section. Those were the mandatory basics to use display, solenoids, switches, lamps and audio.

This section on the other hand handles those commands that are not a must, but may ease your life quite a lot.

4.1. Display related commands

AddScrollUpper	(byte Step) Takes what is present in DisplayUpper2 and scrolls it from the right into DisplayUpper. It will stop when it has reached anything different from blanks. Must be called with Step being 1.
BlinkScore	(byte Event) blinks the score display of the active player. The number of the timer being used is stored in BlinkScoreTimer. There is no stop command, so you have to use KillTimer(BlinkScoreTimer) to stop the blinking.
DisplayScore	(byte Position, unsigned int Score) Shows Score in the display with Position being the player number.
ScrollLower	(byte Step) Takes what is present in DisplayLower2 and scrolls it from the right into DisplayLower. Is mainly suited for machines with the player 3 display on top of 4, as it scrolls display 3 first and 4 afterwards.
ScrollLower2	(byte Step) Same as above but for machines with the displays for players 3 and 4 next to each other, as it scrolls to display 3 through 4.
ScrollUpper	(byte Step) Same as above but for the upper display row.
ShowAllPoints	(byte Dummy) Shows the points of all players
ShowMessage	(byte Seconds) Switches the display to DisplayUpper2 and DisplayLower2 for the given Seconds. After that the display is set to DisplayUpper and DisplayLower.
ShowNumber	(byte Position, unsigned int Number) Shows Number at the given Position which can range from 0 to 15 for the upper and 16 to 31 for the lower row.
ShowPoints	(byte Player) just a shortcut for DisplayScore(Player, Points[Player]); but can be called by a timer as it has only one byte as argument.
SwitchDisplay	(byte Event) Switches the display between DisplayUpper/Lower to DisplayUpper2/Lower2 depending on Event.
WriteLower	(char* DisplayText) Writes the given text to DisplayLower. The text has to be in capital letters. For a complete description of the display system refer to the basic variables section.
WriteLower2	(char* DisplayText) Same as above but with DisplayLower2 as the destination.
WriteUpper	(char* DisplayText) Writes the given text to DisplayUpper. The text has to be in capital

	letters.
WriteUpper2	(char* DisplayText) Same as above but with DisplayUpper2 as the destination.

4.2. Lamp related commands

AddBlinkLamp	(byte Lamp, unsigned int Period) Makes the given Lamp blink with the given Period as on- and off-time. Note that all blinking lamps having the same period assignet are blinking synchronously
RemoveBlinkLamp	(byte LampNo) Makes the given lamp stop blinking
ShowLampPatterns	(byte Step) Shows an animated sequence of lamp patterns. While the command must be called with step being zero, there are three variables that have to be set beforehand: - PatPointer has to point to a structure of the LampPat type which contains the various lamp patterns and the duration for which indicates how long each one has to be shown (in ms). - FlowRepeat determines how often the whole animation has to be repeated - LampReturn points to the routine to be called after the command has finished. If it is zero the command just quits.
StrobeLights	(byte State) Makes all playfield lamps flicker. There is no stop command, so you have to use KillTimer(StrobeLightsTimer) to stop it.

4.3. Solenoid related commands

ActA_BankSol	(unsigned int Duration, byte Solenoid) Used for activating solenoids in machines featuring an A/C relay (System11). The given Solenoid is activated for the given Duration but the A/C relay is switched to the A bank first. If a C bank solenoid is currently active, the activation of the given solenoid is being postponed until the relay can be switched back to the A bank.
ActC_BankSol	(byte Solenoid) Used for activating solenoids in machines featuring an A/C relay (System11). The C bank has priority over the A bank. That means the relay will be switched to the C bank, regardless of any active A bank solenoids. I did this because flashers are usually assigned to the C bank and they can be active quite a long time, especially when animated flashlamp sequences are being shown. This makes their activation time somehow unpredictable.
ReleaseAllSolenoids	() The name says it all.
SolenoidStatus	(byte Solenoid) Returns the activation stats of the given Solenoid.

4.4. Miscellaneous commands
