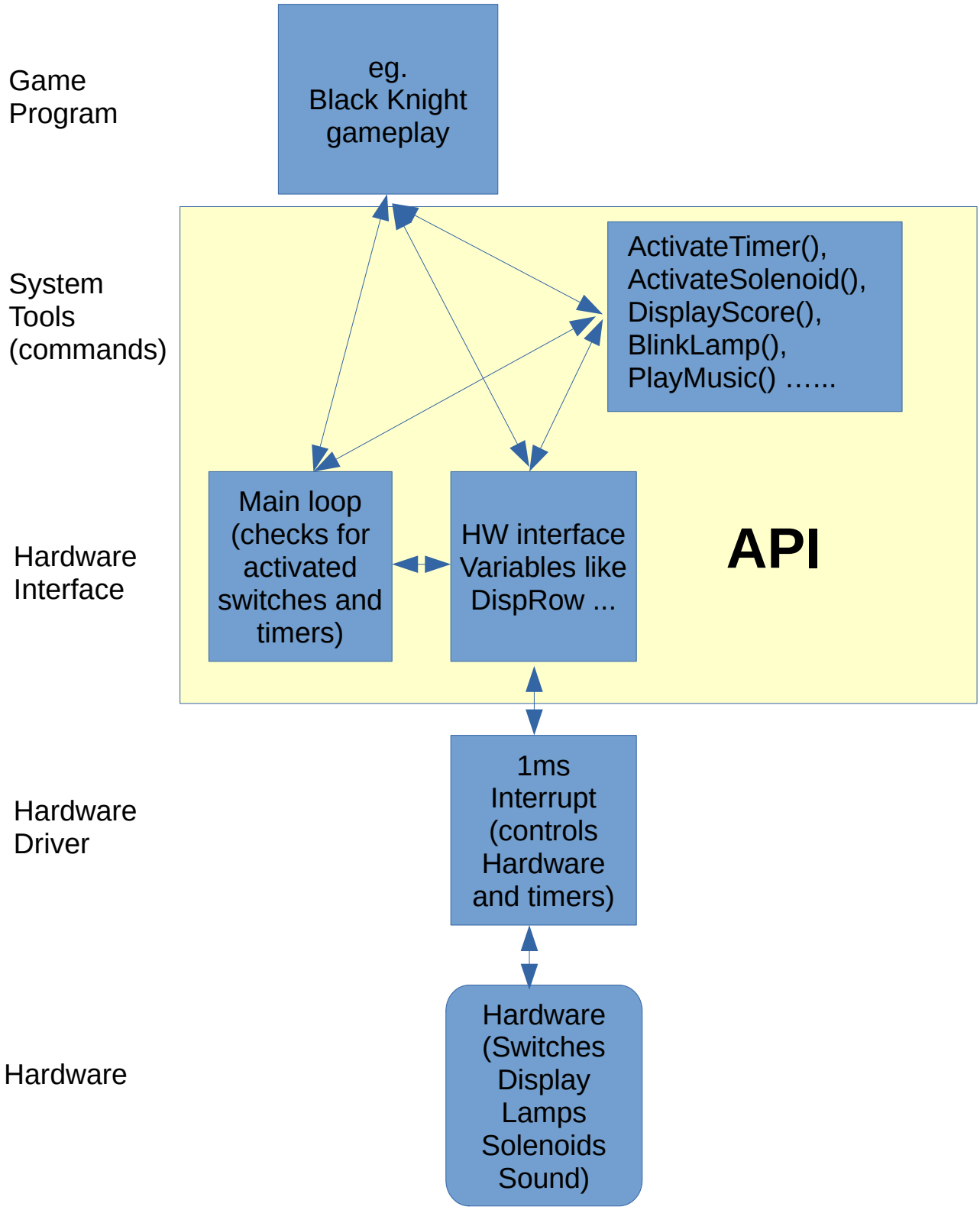
APC Software reference

1. Structure

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 APC.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 QuerySwitch() command. If it's return value is true, the switch is being active otherwise it's off.

In most cases it is not necessary to query the switches manually 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

The status of a lamp can be changed by the command TurnOnLamp(), TurnOffLamp() respectively. To get the current status, use the QueryLamp() command. If it's return value 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 System11 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)(byte) void (*AfterSound)(byte)	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	Has to be set to the game definition structure of the current game. Refer to the Game Code Tutorial Wiki for an example how to use this.
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.
Byte MusicVolume	Sets the volume for the music channel. 0 is the maximum volume, 1 is -6dB, 2 is -12dB and so on.
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.

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 instantaneously. If the given duration is zero, the command will use the default activation times. These are defined in the SolTimes array which is referenced in the GameDefinition structure. In pre System11 machines this command and ReleaseSolenoid are sufficient for solenoid control. In System11 however it is recommended to mainly use ActA_BankSol and it’s C_Bank counterpart as they use a queue to ensure a sequential activation and the correct state of the A/C relay for each requested solenoid. In these games the ActivateSolenoid command may mainly be used for those solenoids requiring an instantaneous response like bumpers and slingshots or when a specific duration is required. You can specify a recycle time for each solenoid by setting SolRecycleTime[SolNumber-1] to a value between 1 and 255. After the solenoid has been released it will wait the given value in ms before the solenoid can be activated again. It makes no difference whther the solenoid is turned off by the ReleaseSolenoid command or by the duration time being over. Example: SolRecycleTime[15-1] = 100 will set a recycle time of 100ms

	for solenoid 15.
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.
QuerySwitch	(byte Switch) Returns the status of the given switch. 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 Switch[73] is the Up/Down switch
ReleaseSolenoid	(byte Solenoid) Will release the given solenoid. In most cases you wont use this command as the duration of the solenoid being activated is already handled by the activation command. However, some solenoids like relays, motors or the flipper fingers have no default activation time and have to be turned off by this command.
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.
TurnOnLamp	(byte Lamp) Turns the given lamp permanently on.

TurnOffLamp	(byte Lamp) Turns the given lamp permanently off.
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 assigned are blinking synchronously
QueryLamp	(byte Lamp) Returns the status of the given lamp. True means the lamp is currently being lit.
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 one, 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). The end of a sequence is indicated by a duration of zero. The command can handle up to 255 of these steps. If your game has the first lamp column in the backbox you have to select

	‘column 1’ in the ‘backbox lamps’ setting in the system settings and the first byte of the pattern part in LampPat has to be zero. Hence one line looks like Duration,0,(7 Byte of lamp pattern data). If your game has the last lamp column in the backbox you have to select ‘column 8’ in the ‘backbox lamps’ setting in the system settings and the last byte of the pattern part in LampPat has to be zero. Hence one line looks like Duration,(7 Byte of lamp pattern data),0. If your game has no lamps in the backbox select ‘none’ in the ‘backbox lamps’ setting, and fill all 8 bytes of the lamp pattern with data. - 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. To abort the command while it is still processing a sequence, just call it with Step being zero.
StrobeLights	(byte Time) Makes all playfield lamps flicker with the period given by the Time parameter (period = Time * 10ms). Time needs to be larger than 2 to start the flickering. To stop the flickering callStrobeLights(0).
TurnOnLamp	(byte Lamp) Turns the given lamp permanently on.
TurnOffLamp	(byte Lamp) Turns the given lamp permanently off.

4.3. Solenoid related commands

ActivateSolenoid	(unsigned int Duration, byte Solenoid) Activates the given solenoid instantaneously. If the given duration is zero, the command will use the default activation times. These are defined in the SolTimes array which is referenced in the GameDefinition structure. In pre System11 machines this command and ReleaseSolenoid are sufficient for solenoid control. In System11 however it is recommended to mainly use ActA_BankSol and it’s C_Bank counterpart as they use a queue to ensure a sequential activation and the correct state of the A/C relay for each requested solenoid. In these games the ActivateSolenoid command may mainly be used for those solenoids requiring an instantaneous response like bumpers and slingshots or when a specific duration is required. You can specify a recycle time for each solenoid by setting SolRecycleTime[SolNumber-1] to a value between 1 and 255. After the solenoid has been released it will wait the given value in ms before the solenoid can be activated again. It makes no difference whther the solenoid is turned off by the ReleaseSolenoid command or by the duration time being over. Example: SolRecycleTime[15-1] = 100 will set a recycle time of 100ms for solenoid 15.
ActA_BankSol	(unsigned byte Solenoid)

	Used for activating solenoids in machines featuring an A/C relay (System11). The given Solenoid is activated for the default duration specified in the SolTimes array which is referenced in the GameDefinition structure. All solenoid activation requests issued by this command, ActC_BankSol or PlayFlashSequence are queued to ensure a sequential activation and the correct state of the A/C relay for each requested solenoid. If an instantaneous activation or a special duration is required you have to use ActivateSolenoid, but in this case you have to tale care of the A/C relay by yourself.
ActC_BankSol	(byte Solenoid) The same as ActA_BankSol but for A bank solenoids.
PlayFlashSequence	(byte* Sequence) Adds a complete solenoid sequence to the queue of solenoids to be activated (the same queue ActA_BankSol and ActC_BankSol are using). This command is intended for flash lamp animations, but you can activate any solenoid with it. The given sequence has to be an array of pairs of bytes representing the individual steps of the animation. The solenoid to be activated has to be in the first byte with the second byte being the time before the next step is activated. Note that the second byte is not the duration of the solenoid activation as this is derived from the SolTimes array, but just the waiting time before the next step is processed. This waiting time is multiplied by ten, which means the pair 5,9 would mean that after activating solenoid number 5 the program waits 9*10ms = 90ms before proceeding to the next step. The last solenoid number of a sequence has to be zero to indicate the end of the sequence. C-Bank flashers start at number 25, so you have to add 24.
QuerySolenoid	(byte Solenoid) Returns the activation status of the given Solenoid.
ReleaseSolenoid	(byte Solenoid) Will release the given solenoid. In most cases you wont use this command as the duration of the solenoid being activated is already handled by the activation command. However, some solenoids like relays, motors or the flipper fingers have no default activation time and have to be turned off by this command.
ReleaseAllSolenoids	() The name says it all.

4.4. Audio commands

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.
StopPlayingSound /Music	() The name says it all. Note that this command will terminate the current playback without calling the AfterMusic / AfterSound command.
QueueNextMusic	(const char* Filename) Queues the music with the given filename (filename must not exceed 8 characters plus suffix – e.g. sound.snd) to be played after the current music has run out. To stop the music file from being looped, the AfterMusic variable must be set to zero.
RestoreMusicVolume	(byte Speed) Restores the music volume by decreasing the MusicVolume variable. The variable will be decreased by one every Speed*10ms.
RestoreMusicVolumeAfterSound	(byte Speed) Works as above, but it waits for the current sound to run out before restoring the music volume.

4.5. Miscellaneous commands

ActivateTimer	(unsigned int Value, byte Argument, void (*EventPointer)(byte)) Activates a timer with Value being the duration until the timer runs out in ms. After running out, the timer calls the subroutine referenced by *EventPointer with Argument. ActivateTimer returns the timer number of the activated timer, this number can be used to kill the timer before it runs out.
KillTimer	(byte TimerNo) Kills timer number TimerNo, which is the timer number that has been returned by the ActivateTimer command.
KillAllTimers	() Total annihilation for timers
ActivatePrioTimer	(unsigned int Value, byte Argument, void (*EventPointer)(byte)) The same as ActivateTimer, but for Priority Timers. The difference between Prio and normal timers is that when a normal timer runs out, the call to the target routine can be delayed depending on the workload. For a prio timer on the other hand this call is made directly inside of the main interrupt routine which eliminates any delay. This delay is mainly caused when a new audio file is opened and can be up to 10ms depending on the SD card. That means in 99% you won't recognize any difference between the two kinds of timers, just in the very rare cases when every ms counts. Use PrioTimers only if absolutely necessary and note that they must only be used for short actions, as they run inside the interrupt and the interrupt handles the system timing. That means if the target routine of a Prio Timer uses an audio command for example, then the displays and lamps might flicker and your other audio channel might start to stutter. Only 8 Prio Timers can be used simultaneously.
KillPrioTimer	() The same as KillTimer only for PrioTimers.

SelSetting	(byte Switch) This is the base for all kinds of settings menus. To use it you have to set the following variables: SettingsPointer → Has to point to an array of bytes where the settings are to be stored. Each setting is represented by a byte value. SettingsList → Has to point to a structure of the type SettingTopic in APC.ino. Read the comments of this structure for details. The last entry of the settings list has to be zero. SettingsFileName → Has to point to the filename which is used to save the array referenced by SettingsPointer. SelSetting must be called with 0 as the argument. In the second entry of SettingTopic a subroutine is being referenced to handle this particular setting topic. The routine to choose depends on the kind of setting. The easiest way to understand how it works is probably to take a look at the APC_setList which defines the system settings menu. However, the following subroutines are available: HandleTextSetting → requires the *TxTpointer of the topic to point to an array of char arrays (2 dimensional array) where all text entries for this topic are listed. The number of text entries has to be specified in the UpperLimit byte of SettingTopic. The value stored in the settings array (referenced by SettingsPointer) does represent the position of the chosen text in the list of text entries. HandleNumSettings → Let's the user choose a numerical value between LowerLimit and UpperLimit. HandleBoolSetting → Offers the choice just between Yes or No which is represented by 0 or 1 in the settings array. ExitSettings → Quits the settings menu and writes a file with the name referenced by SettingsFileName. There are two more subroutines available HandleVolumeSetting and RestoreDefaults, but these are tailored to be used in the System and Game-Settings menu.
WriteToHwExt	(byte Data, byte Selects) Writes one byte of data to the HW extension interface and sets the select pins according to the following scheme: • Bit 0 = 1 sets sel5 to the level of Bit 7 • Bit 1 = 1 sets sel6 to the level of Bit 7 • Bit 2 = 1 sets sel7 to the level of Bit 7 • Bit 3 = 1 sets the SPI_CS1 pin to the level of Bit 7 • Bit 14= 1 sets sel14 to the level of Bit 7 That means a rising and falling edge of sel5 and sel7 requires two calls of WriteToHwExt with Selects = 0b10000101 for the rising and 0b00000101 for the falling edge. Note that sel14 is the select for the latch of the data bits D0 – D7 itself while the other selects are just individual select lines at the HW extensions interface. This function returns a TRUE as an OK signal and FALSE if the write buffer is full. In the latter case the function must be called again later.