Program Listing – “Buzz_Wire_Race.bas”

```
#Region "Program Notes"
#REM

           File: "Buzz_Wire_Race.bas"
        License: MIT (See end of file)
  Change History: 2024/10/27 by Alan Hunt - 1st issue.


Microcontroller Pin Usage
=========================

Pin Usage                                PICAXE 20X2 Pinout & Functions                  Pin Usage
                                 ------------------------------------------------------------
+5V Supply                       |1  Vdd                                        Gnd 20|  0V Supply
Programming Input                |2  Serial In                          Serial out 19|  Debugging Output
BATTERY (ADC3 VoltageBatt)       |3  C.7(ADC3/Out/In)       B.0(In/Out/ADC1/hint1) 18|  AUDIO_TX (to DF Player @T9600)
HelpButton                       |4  C.6(In)             B.1(In/Out/ADC2/hint1/SRQ) 17|  CELL (ADC2 for Cell 1 voltage)
LeaderboardButton                |5  C.5(hpwmA/pvmC.5/Out/In)  B.2(In/Out/ADC4/Comp2+) 16|  AUDIO_STATUS (DF Player active low)
RaceButton                       |6  C.4(hwpmB/SRNQ/Out/In)    B.3(In/Out/ADC5/Comp2-) 15|  LED_RED1 (1st "Get Ready")
InputPosL (Left Start Point)     |7  C.3(hwpmC/ADC7/Out/In)  B.4(In/Out/ADC6/hpwmD/Comp1-) 14|  PWM_BUZZ (Tone for Buzz Wire)
InputPosR (Right Start Point)    |8  C.2(kbclk/ADC8/Out/In)  B.5(In/Out/ADC10/hi2csda/hspisdi) 13|  LED_RED2 (2nd "Get Ready")
InputBuzz (Buzz Wire)            |9  C.1(hspisdo/kbdata/ADC9/Out/In)  B.6(In/Out/ADC11/hserin) 12|  LED_RED3 (3rd "Get Ready")
RESET_BUTTON                     |10 C.0(hserout/Out/In)       B.7(In/Out/hi2cscl/hspisck) 11|  LED_Green(Race "Go" light)
                                 ------------------------------------------------------------


PICAXE 20X2 Notes
=================
    * Chip hardware is the Microchip PIC18F14K22 MCU:
        - Power supply 1.8V to 5.5V. Default clock 8MHz.
        - Pullup and pulldown input currents measured respectively as 1.5nA and 300pA on B.1 when set as a digital input.
        - When an ADC channel is enabled it draws approximately 280 uA and the recommended source impedance is 10 kOhm.
        - Input over/under voltage protection has a maximum clamp current of 20mA.

Program Notes
=============
    * The game has 7 digital inputs monitored on Port C.  Five of the inputs are monitored by software interrupts with "setint" (Only
c.1 to c.5 permitted for 20M2 and 20X2).  The symbol "getInputs" (pinsC) is used to read PortC simultaneously and place it in the symbol
"GameInputs" (variable b3).  Then bits of b3 are used to check specific inputs, like InputBuzz(bit25).
    * Clock rate was set to 16MHz but it's not necessary.  It probably helps to speed up inputs, particularly as I left debugging on for
show.
    * Timer3 word value is set to increment every 131 mS (Prescale8:1 * 65536 * 4 / Clock), so very roughly divide 8 to get seconds.


#EndREM
#EndRegion

#Region "Main Compiler Directives"
'Compiler Directives
#picaxe 20X2
#no_data
#no_table
#no_debug
#define Debugger_On                         'Comment out this line to disable Debugging.

#Endregion

#Region "Resources"
'Pins
    symbol AUDIO_TX = B.0                   '@DF_Player_Mini.basinc: Serial Tx to audio module via 1K resistor.
    symbol CELL1 = pinB.1                   '@Voltage.basinc: Cell 1 voltage monitoring is ADC2(B.1).
    symbol AUDIO_STATUS = pinB.2            '@DF_Player_Mini.basinc: Busy is active low after time lag (AUDIO_IDLE and AUDIO_BUSY).
    symbol LED_RED1 = B.3                   '1st track light.
    symbol PWM_BUZZ = B.4                   'Tone for Buzz Wire.
    symbol LED_RED2 = B.5                   '2nd track light.
    symbol LED_RED3 = B.6                   '3rd track light.
    symbol LED_GREEN = B.7                  '4th "go" and running track light.
    symbol RESET_BUTTON = pinC.0            'Circuit board button to clear game stats if pressed during initialisation.
    symbol HELP_BUTTON = pinC.6             'Button to explain game.
    symbol BATTERY = pinC.7                 '@Voltage.basinc: Battery voltage monitoring is ADC3(C.7).

'Variables
    symbol GameTime = timer                 'Alias for the system "timer", measures elapsed game/idle time time in tenths of a second.
    symbol UpTime = timer3                  'Alias for "timer3", used for random number generation and uptime in 8th's of a second.

    symbol _Debugger = b0                   '"Debugger.basinc" module is limited to using b0 for Byte to ASCII bitstream conversion.

    symbol InterruptSetting = b1            'Interrupt setting for input pins.
    symbol IntSetBuzz = bit9
    symbol IntSetPosR = bit10
    symbol IntSetPosL = bit11
    symbol IntSetRace = bit12
    symbol IntSetLeaderboard = bit13

    symbol InterruptMask = b2               'Interrupt mask for input pins.
    symbol IntMaskBuzz = bit17
    symbol IntMaskPosR = bit18
    symbol IntMaskPosL = bit19
    symbol IntMaskRace = bit20
    symbol IntMaskLeaderBoard = bit21

    symbol GameInputs = b3                  'Game inputs on PortC with quick byte getter "getInputs" and bit checkers in b3.
    symbol getInputs = pinsC
    symbol InputBuzz = bit25
    symbol InputPosR = bit26
    symbol InputPosL = bit27
    symbol RaceButton = bit28
    symbol LeaderboardButton = bit29
```

```
    symbol HelpButton = bit30

    symbol GameStatus = b4                  'Game Status has enumerated state constants
    symbol GameDirection = b5

    symbol SoundLoopIdle = b6               'Pointer for a list of tracks played when no game is running.
    symbol SoundLoopBackground = b7         'Pointer for a list of tracks played as background music during the game.
    symbol SoundLoopCrash = b8              'Pointer for a list of crash sounds that interupt the background music.
    symbol SoundLoopFanfare = b9            'Pointer for a list of tracks played at the end of a game.
    symbol SoundLoopAdvert = b10            'Pointer for a list of adverts played when the game is idle for too long.

    symbol CrashCount = b11
    symbol LapTime = w6                     'Time in tenths of a second.
    symbol PenaltyTime = w7                 'Time added, in tenths, for a premature race start.

    symbol LapRecord = w8
    symbol PlayerNumber = w9
    symbol PlayerRanking = b20

    symbol GameTimerLog = b21
    symbol GameTimer1 = w11
    symbol GameTimer2 = w12
    symbol GameTimer3 = w13

    symbol VoltsCell2 = w14
    symbol VoltsCell1 = w15
    symbol BatteryCapacity = b32

    'Internal variables for gosubs (Try and keep these unique due to ease Interrupt handling)
    symbol _LeaderBoard_ResultPTR = b33     'For Gosub PlayerRank only.
    symbol _LeaderBoard_TimePTR = b34       'For Gosubs PlayerRank and SayLeaderboard.
    symbol _LeaderBoard_PlayerPTR = b35     'For Gosubs PlayerRank and SayLeaderboard.
    symbol _GameEndW = w18                  'For Gosub GameEnd only.
    symbol _LeaderBoardW = w19              'For Gosubs PlayerRank and SayLeaderboard.
    symbol _LeaderBoardB = b40              'For Gosubs PlayerRank and SayLeaderboard.

    '@DF_Player_Mini.basinc Variables
    symbol _Timing = w21                                        'Assign symbol to any spare Word.

    '@DF_Player_Mini.basinc Variables
    symbol _Audio_Digit = b41               'Assign symbol to any spare Byte.
    symbol _Audio_Integer = w22             'Assign symbol to any spare Word.
    symbol _Audio_Fraction = w23            'Assign symbol to any spare Word.
    symbol _Audio_Track = b48               'Assign symbol to any spare Byte.
    symbol _Audio_Byte = b49                'Assign symbol to any spare Byte.

    '@Voltage.basinc Variables
    symbol Voltage_Vdd = w25                'Assign symbol to any spare Word. Stores PICAXE supply voltage as mV.
    symbol _VoltageADCval = w26             'Assign symbol to any spare Word.
    symbol _VoltageModulus = w27            'Assign symbol to any spare Word.

'Constants
    'For interrupts
    symbol VAL_DEFAULT = %00000000          'Interrupt value default
    symbol MASK_DEFAULT = %00111110         'Interrupt mask default
    symbol NOT_MASK_DEFAULT = %11000001
    symbol CONTACT = 1                      'Logic level for contact with buzz wire or end points.
    symbol NO_CONTACT = 0
    symbol PRESSED = 1                      'Logic level for button press.
    symbol NOT_PRESSED = 0

    'For GameStatus variable
    symbol GAME_NOT_SET = 0                 'Not at a valid start point.
    symbol GAME_WAITING = 1                 'At a valid start point and waiting for Race or practise.
    symbol GAME_BEGINNING = 2               'Race button pressed.
    symbol GAME_NOT_STARTED = 3             'Race Go signal was given but not moved off yet.
    symbol GAME_PLAYING = 4
    symbol GAME_ENDING = 5                  'Reached end point and stating performance.
    symbol GAME_PRACTISE = 6                'Moved off without pressing Race button.

    'For GameDirection variable
    symbol R_TO_L = 1
    symbol L_TO_R = 2

    'For Penalty time (tenths of a second)
    symbol PENALTY_PREMATURE_START = 100
    symbol PENALTY_CRASH = 30

    'Initialise Variables
    symbol SOUND_IDLE_START = 177           'MP3 177 to 215 play backgound music when no game is running.
    symbol SOUND_IDLE_END = 215
    symbol SOUND_BACKGROUND_START = 218     'MP3 218 to 238 play background music during the game.
    symbol SOUND_BACKGROUND_END = 238
    symbol SOUND_FANFARE_START = 250        'MP3 250 to 254 play different game endings for leaderboard entries.
    symbol SOUND_FANFARE_END = 254
    symbol SOUND_CRASH_START = 1            'ADVERT 1 to 8 play crash sounds that interupt the background music.
    symbol SOUND_CRASH_END = 8
    symbol SOUND_ADVERT_START = 15          'ADVERT 15 to 22 interuppt idle music to encourage game playing.
    symbol SOUND_ADVERT_END = 22

    'Timer Values (tenths of a second)
    symbol GAME_TIME1 = 100                 'Some encouragement after 10S, or quit a failed race start.
    symbol GAME_TIME2 = 300                 'Some encouragement after 30S.
    symbol GAME_TIME3 = 900                 'Suggest you let someone else play after 90S.
    symbol IDLE_TIMER = 900                 'Play an advert to encourage playing every 90S.

    'For EEPROM storage locations
    symbol aGAME_COUNT  = 2
    symbol LEADERBOARD_SIZE = 20            'Limited to 20 to make speaking nth's easier.
    symbol aLEADERBOARD_PLAYER = 30         'The Leaderboard player numbers occupy EEPROM addresses from here.
    symbol aLEADERBOARD_PLAYER_END = 68     'Player numbers are stored as words, the last byte used is this plus 1.
    symbol aLEADERBOARD_TIME = 70           'Laptimes are stored as words from here.
    symbol aLEADERBOARD_TIME_END = 108      'The last byte is this +1.
    symbol LEADERBOARD_SPOKEN_SIZE = 4      'Number of entries spoken after LEADERBOARD_BUTTON pressed (limited to 4 by track 120).
```

```
    'For Battery Monitoring (Voltage in mV)
    symbol VOLTS_CELL_GOOD = 3400           'A li-ion Samsung INR18650-30Q with constant 200mA load was tested to be about 18% at 3400
mV.
    symbol VOLTS_CELL_EXHAUSTED = 3100      'The battery spec is 2.5V cut-off but tested down to 2.8V, this gave about 6% at 3100 mV.
                                            'Caution: Figures change drastically for 3.0V cut-off cells, or high current loads.

    '@Timing.basinc Constants
    'CLOCK_SET_KHZ options: 31,250,500,1000,2000,4000,8000,16000,32000,or 64000 (Dependenton chip). No equals sign and nothing after
number.
    #define CLOCK_SET_KHZ 16000
    #define TIMER_IN_HUNDRETHS 10

    '@DF_Player_Mini.basinc Constants
    symbol AUDIO_VOLUME = 31                '0 to 31(Maximum).
    symbol AUDIO_BAUD = T9600_16            'Tx is idle high (T) and 9600 bps, the last characters you adjust to your clock speed.
    symbol AUDIO_ACTIVE = 0                 'The AUDIO_STATUS pin is active low, indicating audio playing.
    symbol AUDIO_IDLE = 1                   'AUDIO_STATUS high indicates the module is idle, or seeking to play a track on the disk.

   '@Voltage_X2.basinc Constant
    symbol VoltageDividend = 10595       'Adjust value for FVR inaccuracy, see "Voltage_X2.basinc" Calibration.


'Modules
    #include "Debugger.basinc"
    #include "Timing.basinc"
    #include "DF_Player_Mini.basinc"
    #include "Voltage_X2.basinc"
#Endregion

#Region "Programme Initialisation"
Init:
    if PenaltyTime = PENALTY_PREMATURE_START then goto Main    'A premature start causes a Run 0

    Pause1S                                 'Allows PICAXE Editor time to open the terminal window.
    DebugLine(ppp_filename)                 'Displays the programming filename in PICAXE Editor Terminal if "Debugger_On" is defined.
    DebugLine(ppp_datetime)                 'Displays the programming date and time.

'Setup Pins
    dirsB = %11111001                       'Set port B7 to B0 pins (1=output).
    dirsC = %00000000                       'Ensure all C pins are inputs.
    adcsetup = %0000000000000110            'Use ADC3(Pin 3, C.7) and ADC2 (Pin 17, B.1), this disables digital interface circuitry.
    hpwm PWMDIV16, 0, 0, %1000, 255, 511    'Use PWM to produce lowest tone possible for a potential buzz wire contact sound.

'Setup Timers
    tmr3setup %10110001                     'Enables timer3 to generate randomness and for approx timing (8 per Sec, see notes).

'Initialise Variables
    SoundLoopIdle = SOUND_IDLE_START                'MP3 150 to 172 play background music when no game is running.
    SoundLoopBackground = SOUND_BACKGROUND_START    'MP3 200 to 212 play background music during the game.
    SoundLoopCrash = SOUND_CRASH_START              'ADVERT 1 to 9 play crash sounds that interupt the background music.
    SoundLoopFanfare = SOUND_FANFARE_START          'MP3 250 to 254 play different game endings.
    SoundLoopAdvert = SOUND_ADVERT_START            'ADVERT 15 to 21 to encourage play when idle too long.

    GameTimer1 = GAME_TIME1                 'Some encouragement after 10S, or quit a failed race start.
    GameTimer2 = GAME_TIME2                 'Some encouragement after 30S.
    GameTimer3 = GAME_TIME3                 'Suggest you let someone else play after 90S.

'Initialise Audio Module and check battery
    gosub AudioInitialise
    PlayMP3(063,"Restarted")              'Play "The system has restarted"
    gosub CheckBattery

'Reset game records during startup if necessary
    if RESET_BUTTON = PRESSED then
        DebugLine("@Reset Stats")
        PlayerNumber = 0
        write aGAME_COUNT, WORD PlayerNumber
        LapRecord = 0
        for PlayerRanking = aLEADERBOARD_Player to aLEADERBOARD_TIME_END step 2
            write PlayerRanking, WORD PlayerNumber
        next PlayerRanking
        PlayMP3(0145,"game stats reset")
        gosub AudioWaitForIdle
    endif

'Speak the game stats
    gosub SayGamesPlayed
    gosub SayLapRecord

'Ensure the car is at one end of the track and prepare interrupt settings
    gosub GameGetReady
    InterruptSetting = VAL_DEFAULT
    InterruptMask = MASK_DEFAULT

'Confirm Initialisation
    PlayMP3(0151," Hello the game is ready")
    PlayMP3(0133," Press Help or start now")
    DebugLine("@Initialised")
    debug                                   'Display values in PICAXE Editor unless "no_debug" is defined.

#Endregion

#Region "Main"
Main:
    'Contiunally loop around Main to see if music has ended or GameTimers have expired.
    setint NOT InterruptSetting, InterruptMask  'Set interupts for low state on C1 to C5 (BuzzWire, PosR, PosL, Race and Leaderboard)


    'Check if the audio has finished playing.
    if AUDIO_STATUS = AUDIO_IDLE then
        DebugLine("@Main Without Audio")
        gosub DebugGameStatus
        select case GameStatus
```

```
            case GAME_NOT_SET
                PlayNow(0134," Place car at 1 end of the track")
                Pause10S
            case GAME_WAITING
                gosub CheckBattery
                gosub AudioWaitForIdle
                DebugOut("Play Idle ",#SoundLoopIdle,cr,lf)
                PlayNow(SoundLoopIdle," IdleTrack")         'Play the next rotating idle track
                inc SoundLoopIdle
                if SoundLoopIdle > SOUND_IDLE_END then
                    SoundLoopIdle = SOUND_IDLE_START
                endif
            case GAME_NOT_STARTED, GAME_PLAYING, GAME_PRACTISE
                DebugOut("Play Background ",#SoundLoopBackground,cr,lf)
                PlayNow(SoundLoopBackground," GameTrack")   'Play the next rotating game track
                inc SoundLoopBackground
                if SoundLoopBackground > SOUND_BACKGROUND_END then
                    SoundLoopBackground = SOUND_BACKGROUND_START
                end if
            else
            'Other values GAME_BEGINNING and GAME_ENDING are temporary and should not occur.
        endselect
    endif

    'Check if timers have expired.
    select case GameStatus
        case GAME_WAITING
            if GameTime > IDLE_TIMER then
                PlayAdvert(SoundLoopAdvert,"Roll-up etc")
                Gametime = 0
                inc SoundLoopAdvert
                if SoundLoopAdvert > SOUND_ADVERT_END then
                    SoundLoopAdvert = SOUND_ADVERT_START
                endif
            endif
            if HELP_BUTTON = PRESSED then
                DebugLine("Help Pressed")
                GameTime = 0                    'Reset the timer for encouragement adverts
                PlayNow(0132," GameIntro")
                gosub AudioWaitForIdle
            endif
        case GAME_BEGINNING
            gosub GameBegin
        case GAME_NOT_STARTED
            'DebugOut("GameTime: ",#GameTime,cr,lf)
            'DebugOut(", GameTimer1: ",#GameTimer1,cr,lf)
            if GameTime > GameTimer1 then
                DebugLine("Not started after 10S")
                gosub GameQuit
                PlayNow(0153, "Oh no it's nap time")
                PlayMP3(0152, "Press the race button to start again")
                gosub AudioWaitForIdle
            endif
        case GAME_PLAYING
            select case GameTime
                case > GameTimer1
                    if CrashCount = 0 then
                        PlayAdvert(11,"Too slow")
                    elseif CrashCount > 2 then
                        PlayAdvert(14, "Slow down")
                    endif
                    GameTimer1 = 65535
                case > GameTimer2
                    if CrashCount < 2 then
                        PlayAdvert(11,"Too slow")
                    elseif CrashCount > 5 then
                        PlayAdvert(14, "Slow down")
                    endif
                    GameTimer2 = 65535
                case > GameTimer3
                    PlayNow(0135, "You've been playing a while, let someone else too")
                    GameTimer3 = 65535
            endselect
    endselect

    Pause10mS
    goto Main

#Endregion

#Region "Interrupts"
Interrupt:
    GameInputs = getInputs
    DebugLine("@Interupt")
    gosub DebugGameStatus
    gosub DebugGameInputs

    select case GameStatus
    case GAME_NOT_SET
        if InputPosL = CONTACT OR InputPosR = CONTACT then          'Car now placed correctly at one end of the track.
            GameStatus = GAME_WAITING
            if InputPosL = CONTACT then
                GameDirection = L_TO_R
            else
                GameDirection = R_TO_L
            endif
            PlayNow(062," Thank you")
        endif

    case GAME_WAITING
        if InputPosL = NO_CONTACT AND InputPosR = NO_CONTACT then   'Car moving without Race button, so it's a practise.
            GameStatus = GAME_PRACTISE
            SoundLoopCrash = SOUND_CRASH_START
            CrashCount = 0
```

```
            PlayNow(0147,"having a practise hey.  There's no laptime for you")
        endif
        if LeaderboardButton = PRESSED then
            if HelpButton = PRESSED then
                gosub AudioClear
                gosub CheckBattery                                  'If both pressed then state battery cell voltages
                PlayNow(0070, "The cell voltages are")
                AudioSpeakNumber(VoltsCell1)
                PlayMP3(0130, "millivolts")
                PlayMP3(0060, "and")
                AudioSpeakNumber(VoltsCell2)
                PlayMP3(0130, "millivolts")
            else
                gosub SayLeaderboard
            endif
            GameTime = 0                                            'Reset the timer for encouragement adverts
        endif
        if RaceButton = PRESSED then
            GameStatus = GAME_BEGINNING
        endif

    case GAME_BEGINNING
        if InputPosL = NO_CONTACT AND InputPosR = NO_CONTACT then   'Car moved too early while game still begining (before green light).
            Pause100ms                                              'Pause a while to debounce a player wiggling the handle.
            GameInputs = getInputs
            if InputPosL = NO_CONTACT AND InputPosR = NO_CONTACT then   'Car is really moving too early, so start the game and log the
penalty.
                GameStatus = GAME_PLAYING
                'DebugLine("Started too Soon")
                PlayNow(0148,"Time penalty. Started too soon.")
                low LED_RED1, LED_RED2, LED_RED3
                high LED_GREEN
                GameStatus = GAME_PLAYING
                GameTimer1 = GAME_TIME1
                GameTimer2 = GAME_TIME2
                GameTimer3 = GAME_TIME3
                SoundLoopCrash = SOUND_CRASH_START
                CrashCount = 0
                PenaltyTime = PENALTY_PREMATURE_START
                GameTime = 0
                IntSetPosL = InputPosL
                IntSetPosR = InputPosR
                IntSetBuzz = InputBuzz
                run 0
            endif
        endif

    case GAME_NOT_STARTED
        if InputPosL = NO_CONTACT AND InputPosR = NO_CONTACT then   'Car moving correctly after given the green light.
            DebugLine("Started OK")
            GameStatus = GAME_PLAYING
        endif

    case GAME_PLAYING
        if InputPosL = CONTACT then                                 'Car touching left end of the track.
            if GameDirection = R_TO_L then
                gosub GameEnd
            else
                if GameTime > 20 then                               'debounce a return to start
                    PlayNow(0144," Going back huh")
                    PlayMP3(0152," Press the race button to start again")
                    gosub AudioWaitForIdle
                    gosub GameQuit
                endif
            endif
        endif
        if InputPosR = CONTACT then                                 'Car touching right end of the track.
            if GameDirection = L_TO_R then
                gosub GameEnd
            else
                if GameTime > 20 then                               'debounce a return to start
                    PlayNow(0144," Going back huh")
                    PlayMP3(0152," Press the race button to start again")
                    gosub AudioWaitForIdle
                    gosub GameQuit
                endif
            endif
        endif
        if InputBuzz = CONTACT then                                 'Buzz wire contact
            gosub GameCrash
        endif

    case GAME_PRACTISE
        if InputPosL = CONTACT or InputPosR = CONTACT then          'Car finished practising on the track.
            GameStatus = GAME_WAITING
            if InputPosL = CONTACT then
                GameDirection = L_TO_R
            else
                GameDirection = R_TO_L
            endif
            PlayNow(0150," Practise over, press green button")
        endif
        if InputBuzz = CONTACT then                                 'Buzz wire contact
            gosub GameCrash
        endif

    endselect

    IntSetPosL = InputPosL
    IntSetPosR = InputPosR

    gosub DebugGameStatus
    DebugLine("@Interupt End")
    setint NOT InterruptSetting, InterruptMask
```

```
    return
#Endregion

#Region "Subroutine GameGetReady"
GameGetReady:
    DebugLine("@GameGetReady")
    GameInputs = getInputs
    gosub DebugGameInputs

    if InputPosL = NO_CONTACT and InputPosR = NO_CONTACT then
        do until InputPosL = CONTACT or InputPosR = CONTACT          'Ensure car is at 1 end of the track
            PlayMP3(0134,"Place car at 1 end of the track")
            Pause3S
            GameInputs = getInputs
        loop

        PlayNow(062," Thank you")
    endif

    if InputPosL = CONTACT then
        GameDirection = L_TO_R
    else
        GameDirection = R_TO_L
    end if

    IntSetPosL = InputPosL
    IntSetPosR = InputPosR

    GameStatus = GAME_WAITING

    return
#Endregion

#Region "Subroutine GameBegin"
GameBegin:
    DebugLine("@GameBegin")
    GameStatus = GAME_BEGINNING
    read aGAME_COUNT, WORD PlayerNumber
    inc PlayerNumber                                                 'Don't flash the PlayerNumber increase until a successful end.

    PlayNow(173," Arcade token")
    PlayMP3(157," Player")
    AudioSpeakNumber(PlayerNumber)
    PlayMP3(174," Wait for the green light")
    PlayMP3(175," Mario Kart beeps")

    high LED_RED1
    Pause1S
    high LED_RED2
    Pause1S
    high LED_RED3
    Pause1S
    low LED_RED1, LED_RED2, LED_RED3
    high LED_GREEN

    GameStatus = GAME_NOT_STARTED
    GameTimer1 = GAME_TIME1
    GameTimer2 = GAME_TIME2
    GameTimer3 = GAME_TIME3
    SoundLoopCrash = SOUND_CRASH_START
    CrashCount = 0
    PenaltyTime = 0
    GameTime = 0

    return
#Endregion

#Region "Subroutine GameCrash"
GameCrash:
    DebugLine("@GameCrash")
    PlayAdvert(SoundLoopCrash, "Crash")                              'There is a timed delay in PlayAdvert, which helps as a crash rate
limiter.
    if CrashCount < 255 then
        inc CrashCount
        inc SoundLoopCrash
        if SoundLoopCrash > SOUND_CRASH_END then
            SoundLoopCrash = SOUND_CRASH_START
        endif
    endif
    DebugOut("CrashCount=",#CrashCount,cr,lf)
    return
#Endregion

#Region "Subroutine GameQuit"
GameQuit:
    DebugLine("@GameQuit")
    low LED_RED1, LED_RED2, LED_RED3, LED_GREEN
    gosub GameGetReady
    GameTime = 0
    return
#Endregion

#Region "Subroutine GameEnd"
GameEnd:
    DebugLine("@GameEnd")

    'Record Game and notify of GameEnd
    GameStatus = GAME_ENDING
    LapTime = GameTime
    low LED_GREEN
    gosub AudioClear
    write aGAME_COUNT, WORD PlayerNumber
    PlayNow(0136," You finished!")
```

```
    'Add 1 minute penalty if crash time is greater than laptime.
    _GameEndW = PENALTY_CRASH * CrashCount
    if _GameEndW > Laptime then
        PenaltyTime = PenaltyTime + 600
        PlayMP3(246," Slow down stop the insanity!")
        PlayMP3(155," 1 minute penalty because crashtime > laptime")
    endif

    'Calculate laptime and penalty time (measured in tenths of a second)
    DebugOut("GameTime=", #Laptime)
    DebugOut(", CrashTime=", #_GameEndW)
    PenaltyTime = _GameEndW + PenaltyTime
    LapTime = LapTime + PenaltyTime
    DebugOut(", Adjusted=",#LapTime,cr,lf)

    'Leaderboard ranking and associated messages.
    gosub PlayerRank
    select case PlayerRanking
    case 0
        PlayMP3(0170," Ah you just missed out on the leaderboard, have another try!")
    case 1
        LapRecord = LapTime
        PlayMP3(0149," New lap record")
        _GameEndW = UpTime % 4
        _GameEndW = _GameEndW + 240
        PlayMP3(_GameEndW," Winners track")
        gosub PlayFanfare
    else
        gosub PlayFanfare
    endselect

    'Speak about crashes
    if CrashCount = 0 then
        PlayMP3(0142," No crashes!")
        if LapTime <> LapRecord then
            PlayMP3(0247," Wow all I can say is wow!")
        endif
        PlayMP3(0137," Your time was")
    else
        if CrashCount > 9 then
            PlayMP3(0143," The car is a right off.")
        endif

        PlayMP3(0140," You crashed")
        AudioSpeakNumber(CrashCount)
        PlayMP3(0141," times & each cost")'
        AudioSpeakNumber(PENALTY_CRASH/10)
        PlayMP3(0083," seconds")
        PlayMP3(154," Your adjusted time was")
    endif

    'Speak the laptime seconds
    AudioSpeakTenths(LapTime)
    PlayMP3(0083," seconds")

    If PlayerRanking <> 1 then gosub SayLapRecord

    gosub GameQuit
    PlayMP3(0156," Waiting for next player!")

    return
#Endregion

#Region "Subroutine SayGamesPlayed"
SayGamesPlayed:
    read aGAME_COUNT, WORD PlayerNumber
    DebugOut("@SayGamesPlayed: ",#PlayerNumber,cr,lf)
    PlayMP3(0138," There have been")
    AudioSpeakNumber(PlayerNumber)
    PlayMP3(0139," games")
    return
#Endregion

#Region "Subroutine SayLeaderboard"
SayLeaderBoard:
    read aGAME_COUNT, WORD PlayerNumber
    DebugLine("@SayLeaderBoard")
    if PlayerNumber = 0 then
        '@No games yet
        gosub AudioClear
        gosub sayGamesPlayed
    else
        '@Speak Leaderboard
        PlayNow(0158, "The top")
        If PlayerNumber > LEADERBOARD_SPOKEN_SIZE then
            PlayMP3(LEADERBOARD_SPOKEN_SIZE, "{n}")
        else
            PlayMP3(PlayerNumber, "{n}")
        endif
        PlayMP3(0159," leaderboard positions are:")
        _LeaderBoard_PlayerPTR = aLEADERBOARD_PLAYER
        _LeaderBoard_TimePTR = aLEADERBOARD_TIME
        _LeaderBoard_ResultPTR = 0
        do
            read _LeaderBoard_PlayerPTR, WORD _LeaderBoardW
            if _LeaderBoardW = 0 then exit                      'Quit leaderboard if the entry isn't filled yet.

            _LeaderBoardB = 160 + _LeaderBoard_ResultPTR        'Track 0160 to 0163 is "1st place" to "4th place".
            PlayMP3(_LeaderBoardB," {nth} place is")
            PlayMP3(0157," Player")
            AudioSpeakNumber(_LeaderBoardW)

            read _LeaderBoard_TimePTR, WORD _LeaderBoardW
```

```
            PlayMP3(0164," with laptime")
            AudioSpeakTenths(_LeaderBoardW)
            PlayMP3(0083," seconds")

            inc _LeaderBoard_ResultPTR
            _LeaderBoard_PlayerPTR = _LeaderBoard_PlayerPTR + 2
            _LeaderBoard_TimePTR = _LeaderBoard_TimePTR + 2

        loop until _LeaderBoard_ResultPTR = LEADERBOARD_SPOKEN_SIZE         'Only speak the first n entries.

        if PlayerNumber > LEADERBOARD_SPOKEN_SIZE then gosub sayGamesPlayed 'Speak game count if more records.
        PlayMP3(0165, "Press green button to beat them")
    endif
    GameTime = 0
    return
#Endregion

#Region "Subroutine SayLapRecord"
SayLapRecord:
    read aLEADERBOARD_TIME, WORD LapRecord
    DebugOut("@SayLapRecord: ",#LapRecord,cr,lf)
    if LapRecord > 0 then
        PlayMP3(0146," The lap record is")
        AudioSpeakTenths(LapRecord)
        PlayMP3(0083," seconds")
    endif
    return
#Endregion

#Region "Subroutine PlayFanfare"
PlayFanfare:
    PlayMP3(0166, "Congratulations you are")
    _GameEndW = 32 + PlayerRanking
    PlayMP3(_GameEndW, "nth")
    PlayMP3(0167, "on the leaderboard")
    _GameEndW = PlayerNumber / 2
    if PlayerRanking <= _GameEndW then
        PlayMP3(168," Musical tribute just for you.")
        PlayMP3(SoundLoopFanfare," Fanfare")
        inc SoundLoopFanfare
        if SoundLoopFanfare > SOUND_FANFARE_END then
            SoundLoopFanfare = SOUND_FANFARE_START
        endif
    endif
    return
#Endregion

#Region "Subroutine PlayerRank"
PlayerRank:
    'DebugLine("@PlayerRank")

    '@Work down the leaderboard to determine ranking from the 1st beaten or blank time entry.
    _LeaderBoard_ResultPTR = aLEADERBOARD_TIME
    do
        read _LeaderBoard_ResultPTR, WORD _LeaderBoardW
        if _LeaderBoardW = 0 OR LapTime < _LeaderBoardW then exit
        inc _LeaderBoard_ResultPTR
        inc _LeaderBoard_ResultPTR
    loop until _LeaderBoard_ResultPTR > aLEADERBOARD_TIME_END

    '@Chose action based on ranking.
    select case _LeaderBoard_ResultPTR
    case = aLEADERBOARD_TIME_END
        '@Last entry on the leaderboard.
        PlayerRanking = aLEADERBOARD_TIME_END - aLEADERBOARD_TIME / 2 + 1
        write aLEADERBOARD_PLAYER_END, WORD PlayerNumber
        write aLEADERBOARD_TIME_END, WORD LapTime
        'DebugOut("RankLast=",#PlayerRanking,cr,lf)
    case > aLEADERBOARD_TIME_END
        '@Not on the leaderboard.
        PlayerRanking = 0
        'DebugLine("Rank=None")
        return
    else
        '@Other placing on the leaderboard.
        PlayerRanking = _LeaderBoard_ResultPTR - aLEADERBOARD_TIME / 2 + 1
        'DebugOut("RankElse=",#PlayerRanking,cr,lf)

        '@Work up the leaderboard to shuffle slower entries down.
        _LeaderBoard_PlayerPTR = aLEADERBOARD_PLAYER_END - 2
        _LeaderBoard_TimePTR = aLEADERBOARD_TIME_END - 2
        do
            'Move time entry down
            read _LeaderBoard_TimePTR, WORD _LeaderBoardW
            _LeaderBoard_TimePTR = _LeaderBoard_TimePTR + 2
            write _LeaderBoard_TimePTR, WORD _LeaderBoardW
            _LeaderBoard_TimePTR = _LeaderBoard_TimePTR - 4

            'Move player entry down
            read _LeaderBoard_PlayerPTR, WORD _LeaderBoardW
            _LeaderBoard_PlayerPTR = _LeaderBoard_PlayerPTR + 2
            write _LeaderBoard_PlayerPTR, WORD _LeaderBoardW
            _LeaderBoard_PlayerPTR = _LeaderBoard_PlayerPTR - 4

        loop until _LeaderBoard_TimePTR < _LeaderBoard_ResultPTR

        '@Insert new leaderboard entry
        _LeaderBoard_PlayerPTR = _LeaderBoard_PlayerPTR + 2
        write _LeaderBoard_PlayerPTR, WORD PlayerNumber
        _LeaderBoard_TimePTR = _LeaderBoard_TimePTR + 2
        write _LeaderBoard_TimePTR, WORD LapTime
    endselect
    'DebugOut("Rank=",#PlayerRanking,cr,lf)
    #rem'For Debug output only:
```

```
    for _LeaderBoard_PlayerPTR = aLEADERBOARD_PLAYER_END to aLEADERBOARD_PLAYER STEP -2
        'DebugOut("PlayerRankPTR=",#_LeaderBoard_PlayerPTR)
        read _LeaderBoard_PlayerPTR, WORD _LeaderBoardW
        DebugOut("Player=",#_LeaderBoardW)
        _LeaderBoard_TimePTR = _LeaderBoard_PlayerPTR + aLEADERBOARD_TIME - aLEADERBOARD_PLAYER
        read _LeaderBoard_TimePTR, WORD _LeaderBoardW
        DebugOut(", Time=",#_LeaderBoardW,cr,lf)
    next _LeaderBoard_PlayerPTR
    #endrem
    return
#EndRegion

#Region "Subroutine CheckBattery"
CheckBattery:
    GetVdd
    VoltsByVdd(VoltsCell2,3)                        'Get half battery voltage on ADC3 and store in VoltsCell2 as mV.
    VoltsByVdd(VoltsCell1,2)                        'Get cell voltage on ADC2 and store in VoltsCell1 as mV.
    VoltsCell2 = VoltsCell2 * 2 - VoltsCell1        'Convert ADC3 input to Cell2 voltage.
    'DebugOut("Vdd: ",#Voltage_Vdd,"mV",cr,lf)
    'DebugOut("Cell 2: ",#VoltsCell2,"mV",cr,lf)
    'DebugOut("Cell 1: ",#VoltsCell1,"mV",cr,lf)

    if VoltsCell2 > VOLTS_CELL_GOOD AND VoltsCell1 > VOLTS_CELL_GOOD then
        'DebugLine("Battery cells good")
    else
        if VoltsCell2 < VOLTS_CELL_EXHAUSTED OR VoltsCell1 < VOLTS_CELL_EXHAUSTED then
            'DebugLine("Battery cells exhausted")
            PlayMP3(0069,"System shutting down due to low battery")
            sleep 0                                 'Enter low power state, essentially off.
        else
            'DebugLine("Battery cells low")
            PlayMP3(0068,"Warning, battery is below 20%")
        endif
    endif
    return
#Endregion

#Region "Subroutine DebugGameStatus"
#ifndef Debugger_On
    DebugGameStatus:
    Return
#else
    DebugGameStatus:
        DebugOut("GameStatus: ")
        select case GameStatus
        case GAME_NOT_SET
            DebugOut("NotSet")
        case GAME_WAITING
            DebugOut("Waiting")
        case GAME_BEGINNING
            DebugOut("Beginning")
        case GAME_NOT_STARTED
            DebugOut("NotStarted")
        case GAME_PLAYING
            DebugOut("Playing")
        case GAME_ENDING
            DebugOut("Ending")
        case GAME_PRACTISE
            DebugOut("Practising")
        endselect

        DebugOut(", Direction: ")
        select case GameDirection
        case L_TO_R
            DebugLine("Left to Right")
        case R_TO_L
            DebugLine("Right to Left")
        else
            DebugLine("Not Set")
        endselect
    return
#endif
#Endregion

#Region "Subroutine DebugGameInputs"
#ifndef Debugger_On
    DebugGameInputs:
    Return
#else
    DebugGameInputs:
        DebugOut("GameInputs: ", #GameInputs, " ")
        if GameInputs = 128 then
            DebugLine("Nothing")
        else
            if InputBuzz = CONTACT then
                DebugOut("BuzzWire ")
            endif
            if InputPosR = CONTACT then
                DebugOut("Right ")
            endif
            if InputPosL = CONTACT then
                DebugOut("Left ")
            endif
            if RaceButton = PRESSED then
                DebugOut("Race ")
            endif
            if HelpButton = PRESSED then
                DebugOut("Help")
            endif
            DebugOut(cr,lf)
        endif
    return
#endif
#Endregion
```

```
#Region "License"
#Rem

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

#EndRem
#EndRegion
```

Program Listing – “Debugger.basinc”

```
#Region "Read Me"
#REM

           File: "Debugger.basinc"
        License: MIT (See end of file)
 Change History: 2024/10/1 by Alan Hunt - Improved help text.
                 2024/9/30 by Alan Hunt - 1st issue.


Description
===========
This simple module provides:

    *  A debugging feature that can easily be turned on or off using a single compiler directive.
    *  Improved program readability in addition to debugging.
    *  An efficient and visually easy way to represent a byte (or port) as individual bit values.
    *  The option of having 2 debug levels.

It can be useful to easily turn off debugging because this avoids serial transmissions delays that could affect program performance.
These transmissions also temporarily disable software interrupts too.

The statements in this module start with "Debug", which I find it easier on the eye than sertxd.  In fact the sertxd command is being
used, so any information about program execution points or particular program state can be sent.  Hopefully, you will find a key benefit
of using brief text is that the debug messages help static reviews of the program too, such as ``DebugLine("Not started after 10S")``.

Taking this further, it's nice to see macros with text explanations highlighted by the PICAXE Editor, rather than over use of comments
in your program.  For example, it would be awful to remember or lookup the contents of audio track 112, but this line makes it very
clear ``PlayNow(0112, "The cell voltages are")``.  In the PlayNow macro you can use the text with debugging statements and comment out
these lines when things are working, or turn all the debugging off in one go.  Whatever you choose the PICAXE Editor keeps on displaying
the bright red text as part of your program.

The DebugBinary(val) statement can be used when a byte is holding important information in individual bits.  For readability the byte is
written as two nibbles separated by a space.  For example, to view logic level inputs on port C it's just ``DebugBinary(pinsC)``, which
would show "0010 0001" if C.0 and C.5 were high.

The Debugger's sertxd output is the usual way to perform PICAXE debugging because it sends text or data to the PICAXE Editor Terminal
window with the same connection as the programming interface.  Most of the functionality in this module is achieved by creating a unique
alias for sertxd.  When debugging is turned off this alias becomes comment lines instead off sertxd.  Any sertxd commands in your
program remain intact, so these can provide always-on basic information in addition to turning on or off detailed debugging.  However,
in normal situations, just using the Debugger statements is recommended.

In comparison to the Debugger/sertxd output, the PICAXE debug command only performs a large snapshot of variable values.  For an X2 chip
with default clock rate the debugger/sertxd output will send 10 bytes in 15mS, compared to the debug output which takes 180mS, without
any indication of the execution points.  For M2 chips at 4800bps the times will double.  The PICAXE Editor will also try to disable its
Terminal when debug is active, rendering sertxd useless.  Debugging output would then need to be directed to another pin using
serout/hserout and to a different com port and Terminal on the PC.


Programming Guide
=================
The following statements are defined in this module:

    DebugOut               This is an alias for sertxd, so use in the same way, e.g. ``DebugOut("Player ",#player," Wins",cr,lf)``.
    DebugBinary(val)       This is a 2 line macro that displays a byte value as two nibbles, e.g. ``DebugBinary(18)`` displays "0001
0010".
                           NB: It's very useful to display the status of 8 bit input/output ports, e.g. ``DebugBinary(pinsC)``.
    DebugLine(msg)         This is a 1 line macro that takes text and adds a CR and LF to it, e.g. ``DebugLine("@Initialising")``.


This module uses just a few lines of pre-processor directives, with no program memory space unless the debug statements are used and
debugging is turned on.

The debugging information can be descriptive text, such as status or program position, and/or variables and constants.  For example,
``Debugger("@Interrupt")`` would indicate the passing of a significant program execution point.  To output a value as text use the "#"
sign, such as ``Debugger("Battery = ",#BattVoltage," mV",cr,lf)``. The cr and lf provide a carriage return and line feed to move the
cursor down to the start of the next line.

The Debugger statements are turned off in your main program with the compiler directive ``#define no_debugger`` and turned on by just
commenting out that line.  The compiler directive must be defined before the include statement for this file.

Variable b0 is used by this module to provide the byte to binary string conversion.  To remind yourself, it's recommended that you add
the line ``symbol _Debugger = b0`` in your program.

The PICAXE Editor will take some time to open its Terminal window for the debug output, so a two second pause is recommended in your
program initialisation before its first use.

The problem with keeping both sertxd commands and debug statements in your program is that debugging can't be fully turned off.  The
recommended approach is to rely only on the Debugger statements, just create them and then comment out one's that were used for detailed
debugging.

Debugging quickly consumes program memory space, so text should be kept as short as possible and debugging that is no longer necessary
should be commented out.


Below is a sample configuration, assuming the following program structure:

    1) Program Notes
    2) Compiler Directives
    3) Resources
        3a) Pins
        3b) Variables
        3c) Constants
        3d) Modules
    4) Initialisation
    5) Main:
    6) Interrupt:
    7) Subroutines
```

```
Sample Configuration
====================
'Compiler Directives
    #picaxe 20X2
    #no_data
    #no_table
    #no_debug
    #terminal 9600
    '#define no_debugger                     'Comment out this line to enable Debugging.

'Variables
    symbol _Debugger = b0                   '"Debugger.basinc" module is limited to using b0 for Byte to ASCII bitstream conversion.

'Modules
    #include "Debugger.basinc"

'Initialisation
    pause 2000                              'Wait for the PICAXE Editor Terminal window to come online.

#EndRem
#EndRegion

#Region "Program"
'If ``#define no_debugger`` is present the pre-processor comments out the Debugger statements.
#ifndef no_debugger
    #define DebugOut sertxd
    #define DebugBinary(val) b0=val : sertxd(#bit7,#bit6,#bit5,#bit4," ",#bit3,#bit2,#bit1,#bit0)
    #define DebugLine(msg) sertxd(msg,cr,lf)
#else
    #define DebugOut                        'Replaced with comment.
    #define DebugBinary(val)                'Replaced with comment.
    #define DebugLine(msg)                  'Replaced with comment.
#endif

#Endregion

#Region "License"
#Rem
License
=======
MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

#EndRem
#EndRegion
```

Program Listing – “Timing.basinc”

```
#Region "Read Me"
#REM

           File: "Timing.basinc"
        License: MIT (See end of file)
 Change History: 2024/10/4 by Alan Hunt - Added the error handling of #terminal if it is already defined in the main program.
                 2024/10/4 by Alan Hunt - 1st issue.


Description
===========
This software was originally used for a game where the clock rate hadn't been finalised and the audio interface required consistent
timings to be near minimum levels.  Modifying more than a few pause commands by hand is unwieldy and error prone, particularly because
arithmetic like "Pause 1 * ClockMultiplier" is not allowed with the pause commands.  Ultimately, a short list of logarithmic timings was
easy to remember and they provided all the accuracy needed on most projects.  It became a bit of fun to delve deeper into PICAXE timing
and record some timing on an oscilloscope and this module emerged.  This module's list of pause statements doesn't consume any extra
program memory, and the module has these features:

    * A list of pause statements that provide timing delays regardless of the PICAXE clock rate: Pause1mS, Pause3mS, Pause10mS,
Pause30mS, Pause100mS, Pause300mS, Pause1S, Pause3S and Pause10S.
    * The macro ``Pause_mSec(t)`` allows you to specify specific pause times regardless PICAXE clock rate.
    * The PICAXE clock rate is set using a kHz value: 31, 250, 500, 1000, 2000, 4000, 8000, 16000, 32000, or 64000 (Dependent on chip).
    * The debug terminal baud rate is automatically set to the appropriate value for the PICAXE clock rate.
    * For X2 chips, the "timer" variable is set to increment at a configured rate if ``TIMER_IN_HUNDRETHS`` is defined in your program
and given a value.  This proved useful to measure lap time in a game.
    * A reference list of actual pause timings is shown in the Programming Guide notes below, these are valid if you use this module, or
just use regular pause commands and do integer math based on the ratio of actual clock rate to default clock.  From this some subjective
recommendations are made for minimum clock rates with certain durations.  Just don't expect a 1mS delay to be added when you use ``pause
1`` :)


Limitations
===========

Processing Delays
-----------------
There are processing delays around a pause statement, just to process each line of a program. For example, an output toggling test with
``high b.1`` immediatley followed by ``low b.1`` created a 267uS pulse width with a 20X2 or 419uS with a 20M2 when using their default
clock rates.  Please note, if you just wanted an accurate or short time pulse output then you would use the ``pulsout pinName,
timeWord``, it supports 10us increments based on either an M2 or X2 running at 4MHz.

Continuing with the testing and processing delays, the processing time is approximately doubled when inserting a ``pause 1 `` between
the high and low sequence.  The test pulse then became 1.50mS with a 20X2 and 2.1mS with an 20M2, instead of the idealistic 1.00mS.  The
2 lines of processing delay grow significantly at low clock rates.  For an 20X2 with a 31kHz clock the ClockDivider is 256, so ``pause
1`` and ``pause 2`` should cause a 256mS and 512mS pulse, but the results were 371mS and 633mSm respectively.  They were roughly 256mS
apart but with an added 115mS.  Even with a minimum ``pauseus 1``, this added 54mS to the output toggling test.  Changing to an X2
running at 64MHz the ClockMultiplier is 8, so 125us and 250us expected, but 185uS and 316us totals were seen, so the gap is still about
right and the processing delays have come down to 60us.

So it's worth keeping in mind that: incrementing pause values gives quite accurate results; the pause command introduces a procesing
delay in addition to its timed delay; the processing delays are a large factor for small pause times or slow clock rates; and high clock
rates will make pauses more accurate, albeit at the cost of power and heat.

It would be possible to compensate for processing delays in most situations, but it should not be necessary. A pause statement may be
only one of many statements each causing their own processing delay.  If more accurate delays are needed, you can take a measurement and
make a delta change if the clock rate is fixed.  Just remember to check timings again if you modify any of your program between the
events you're trying to pause.


Available Clock Rates
---------------------
Parameters for CLOCK_SET_KHZ have chipset limitations as defined in PICAXE documentation, see Manual 2, setfreq.  Only the common
internal clock rates are included in this module: K31, K250, K500, M1, M2, M4, M8, M16, M32 and M64, where K=kHz and M=MHz.


Other Factors
-------------
When using clock rates below the chips default, the accuracy of integer division needs to be considered. This module compensates for
most of this by scaling up to use ``pauseus``.  Short delays are realy not be possible anyway due to the processing time mentioned
earlier.  Another key consideration is the granularity of the Pause_mS(t) macro at lowered clock rates.  Its simplistic implementation
means the macro will only affect changes if the value of t increases by the modulus of the clock divisor amount, so if an X2 is clocked
at 1MHz instead of the default 8MHz, the actual pause value increments at 8, 16, 24 etc.

Unfortunately, when using a clock rate of 31kHz it's impossible to use the sertxd command with the PE (PICAXE Editor) Terminal.  The
data is still being sent at 37.5bps, so it takes a very long time, but the minimum baud rate for the PE Terminal is 110 bps.


Programming Guide & Timing Tests
================================
Below is a list of statements defined in this module, these can be used just like any other PICAXE command if this module is included.
The statements are equivalent to using ``pause t * ClockRatio`` when using integer maths and converting some statements to ``pauseus t *
ClockRatio * 100`` to help with scaling of low clock rates.  The timings shown are based on the test described in the Processing Delays
section above.  The recommended minimum clock rates are subjective, based on needs:

    Pause1mS            Minimum recommended clock rate is 8MHz, the X2 default:
                            - 20X2@8MHz(Default) ``Pause 1`` and ``Pause1mS``=1.50mS, @64MHz=1.07mS.
                            - 20M2@4MHz(Default)=2.11mS, @8MHz=1.55mS, @32MHz=1.15mS
    Pause3mS            Minimum recommended clock rate is 4MHz, the M2 default:
                            - 20X2@4MHz=3.98mS, @8MHz=3.56mS, @64MHz=3.08mS.
                            - 20M2@4MHz=4.23mS, @8MHz=3.66mS, @32MHz=3.15mS
    Pause10mS           Minimum recommended clock rate is 1MHz:
                            - 20X2@1MHz=14.1mS, @8MHz=10.6mS, @64MHz=10.1mS.
                            - 20M2@1MHz=14.4mS, @4MHz=11.4mS, @8MHz=10.6mS, @32MHz=10.2mS
    Pause30mS           Minimum recommended clock rate is 500kHz.
                            - 20X2@500KHz=38.3mS, @8MHz=30.6mS.
                            - 20M2@500kHz=40.1mS
    Pause100mS          Minimum recommended clock rate is 250kHz.
                            - 20X2@250kHz=118.6mS, @8MHz=100.7mS.
                            - 20M2@250kHz=120.1mS
```

```
    Pause300mS          Use any clock rate.
                            - 20X2@31kHz=418.4mS, @250kHz=318.7mS.
                            - 20M2@31kHz=477.8mS, @250kHz=320.2mS.
    Pause1S                     Use any clock rate.
    Pause3S                     Use any clock rate.
    Pause10S            Maximum clock rate is 32MHz due to word variable overflow at 64MHz.

This macro is available for use from this module:

    Pause_mSec(t)       This macro takes a variable or constant "t" and conveniently creates the pause according to the clock rate.
                        A good use of this macro is where you need to increment a pause fairly accurately, for example
``Pause_mSec(1)``=2.15ms, whereas values above 1 add precisely 1.024mS per step.  The macro also provides a quick and dirty solution
when playing with clock timing, but other than that this macro should be avoided. The issue is that it consumes program memory and
speed/accuracy in addition to requiring "_Timing" to be defined to a spare word variable, for example "#define _Timing w21".  For fixed
pause times that are not already included in this file, you can create your own based on the definitions of the standard delays shown in
this file.  They are calculated in pre-compilation, which avoids memory usage and calculation delays.

This module requires the type of PICAXE chip to be declared in your program, which is normal practise, e.g. #picaxe 20X2.

If you have ``#terminal`` defined in your main program you should delete or comment out this line.  The #terminal directive is defined
within this module with the optimal baud rate for your clock rate.  If the terminal is not open it should open after a program download,
but the PE will not change the Terminal baud rate if the Terminal is already open.  So if you download a new clock rate, either close
the Terminal beforehand or change the baud rate manually.

As shown in the sample configuration below, your program needs to "#define" pre-processor directives before the inclusion of this file:


Sample Configuration
====================
#Region "Resources"

'Variables
    'Timing.basinc Variables
    #define _Timing w21                 'Assign symbol to any spare Word.

'Constants
    '@Timing.basinc Constants
    'CLOCK_SET_KHZ options: 31,250,500,1000,2000,4000,8000,16000,32000,or 64000 (Dependent on chip). No equals sign and nothing after
number.
    #define CLOCK_SET_KHZ 16000
    'The defintion below is optional, if present settimer is used to set the major tick rate.  The example below makes the Timer
variable increment every tenth of a second.
    #define TIMER_IN_HUNDRETHS 10

'Modules
    #include "Timing.basinc"

#Endregion

#EndREM
#EndRegion

#Region "Program"

'Determine the Default Clock Rate in kHz
'=======================================
'Your program should specify the chip that it's aligned to, for example ``#picaxe 20X2`` will define "_20X2".
#ifdef _20X2
    #define CLOCK_DEFAULT_KHZ 8000
#elseifdef _28X2
    #define CLOCK_DEFAULT_KHZ 8000
#elseifdef _40X2
    #define CLOCK_DEFAULT_KHZ 8000
#else
    'The chip is assumed to have a 4MHz default clock if the preceding if commands are false.
    #define CLOCK_DEFAULT_KHZ 4000
#endif

'Determine and Set key parameters
'================================
'NB: TerminalRate is the bit rate for the Serial Out pin, as used for debugging with the sertxd command.
#ifndef CLOCK_SET_KHZ
    #define CLOCK_SET_KHZ CLOCK_DEFAULT_KHZ
#endif

#if CLOCK_DEFAULT_KHZ = 8000
    #if CLOCK_SET_KHZ = 31
        #define ClockDivider 256
        setfreq K31
        'NB: PE Terminal will not operate at 37.5 bps but PICAXE still sends.
    #elseif CLOCK_SET_KHZ = 250
        #define ClockDivider 32
        setfreq K250
        #terminal 300
    #elseif CLOCK_SET_KHZ = 500
        #define ClockDivider 16
        setfreq K500
        #terminal 600
    #elseif CLOCK_SET_KHZ = 1000
        #define ClockDivider 8
        setfreq M1
        #terminal 1200
    #elseif CLOCK_SET_KHZ = 2000
        #define ClockDivider 4
        setfreq M2
        #terminal 2400
    #elseif CLOCK_SET_KHZ = 4000
        #define ClockDivider 2
        setfreq M4
        #terminal 4800
    #elseif CLOCK_SET_KHZ = 8000
        #define ClockMultiplier 1
        setfreq M8
        #terminal 9600
```

```
    #elseif CLOCK_SET_KHZ = 16000
        #define ClockMultiplier 2
        setfreq M16
        #terminal 19200
    #elseif CLOCK_SET_KHZ = 32000
        #define ClockMultiplier 4
        setfreq M32
        #terminal 38400
    #elseif CLOCK_SET_KHZ = 64000
        #define ClockMultiplier 8
        setfreq M64
        #terminal 76800
    #endif
#else
    #if CLOCK_SET_KHZ = 31
        #define ClockDivider 128
        setfreq K31
        'NB: PE Terminal will not operate at 37.5 bps but PICAXE still sends.
    #elseif CLOCK_SET_KHZ = 250
        #define ClockDivider 16
        setfreq K250
        #terminal 300
    #elseif CLOCK_SET_KHZ = 500
        #define ClockDivider 8
        setfreq K500
        #terminal 600
    #elseif CLOCK_SET_KHZ = 1000
        #define ClockDivider 4
        setfreq M1
        #terminal 1200
    #elseif CLOCK_SET_KHZ = 2000
        #define ClockDivider 2
        setfreq M2
        #terminal 2400
    #elseif CLOCK_SET_KHZ = 4000
        #define ClockMultiplier 1
        setfreq M4
        #terminal 4800
    #elseif CLOCK_SET_KHZ = 8000
        #define ClockMultiplier 2
        setfreq M8
        #terminal 9600
    #elseif CLOCK_SET_KHZ = 16000
        #define ClockMultiplier 4
        setfreq M16
        #terminal 19200
    #elseif CLOCK_SET_KHZ = 32000
        #define ClockMultiplier 8
        setfreq M32
        #terminal 38400
    #elseif CLOCK_SET_KHZ = 64000
        #error "An M2 chip cannot clock at 64MHz"
    #endif
#endif


'Determine ClockRatio and Tick Rate and define Pause_mSec(t) macro
'=================================================================
'NB: ClockRatio is a pre-processor symbol that is used as a multiplier or divisor in timing symbols to generate pause statements.
'NB: For X2 chips, TickRate is the incrementing rate for the system "timer" variable.
#ifdef ClockMultiplier
    symbol MinorTickuS = 32 / ClockMultiplier   'Minor ticks = 256/Clock (Default=32uS, x2=16uS, x4=8uS, x8=4uS).
    #define ClockRatio * ClockMultiplier
    #ifdef _Timing then
        #define Pause_mSec(t) _Timing = t * ClockMultiplier: pause _Timing
    #endif
#else
    symbol minorTickuS = 32 * ClockDivider      'Minor ticks = 256/Clock (/2=64uS, /4=128uS, /8=256uS, /16=512uS, /32=1024uS,
/256=8096uS).
    #define ClockRatio / ClockDivider
    #ifdef _Timing then
        #define Pause_mSec(t) _Timing = t / ClockDivider: pause _Timing
    #endif
#endif


'If an X2 chip then set the timer rate
'=====================================
#if CLOCK_DEFAULT_KHZ = 8000
    #ifdef TIMER_IN_HUNDRETHS then
        symbol MinorTicksIn_10mS = 10000 / MinorTickuS
        symbol TicksRequired = TIMER_IN_HUNDRETHS * MinorTicksIn_10mS
        symbol TickToSet = 65536 - TicksRequired
        settimer TickToSet                                  'Sets incrementing rate for the system "timer" word variable.
        'For example, at 16MHz the "timer" word value increments every 0.1S with a setting of 59,286 (65,536-(100,000/minorTickuS).
    #endif
#endif


'Pre-Processor definitions for standard pause statements
'=======================================================
'NB: These pause deifintions work regardless of clockrate, just enter a statement like "Pause10mS" in your program:
    'A "pauseus" has 10uS units at default clock rate.  It's used here to provide better accuracy and low range.
    symbol ustime1mS = 100 ClockRatio
    #define Pause1mS pauseus ustime1mS
    symbol ustime3mS = 300 ClockRatio
     #define Pause3mS pauseus ustime3mS
    symbol ustime10mS = 1000 ClockRatio
    #define Pause10mS pauseus ustime10mS
    symbol ustime30mS = 3000 ClockRatio
    #define Pause30mS pauseus ustime30mS

    'This prevents a word overflow when the chip is clocked at 8 times its default.
    #ifdef ClockMultiplier
```

```
        symbol time100mS = 100 ClockRatio
        #define Pause100mS pause time100mS
    #else
        symbol ustime100mS = 10000 ClockRatio
        #define Pause100mS pauseus ustime100mS
    #endif

    'This extends the use of pauseus for low clock rates.
    #ifdef ClockDivider
        symbol ustime300mS = 30000 ClockRatio
        #define Pause300mS pauseus ustime300mS
    #else
        symbol time300mS = 300 ClockRatio
        #define Pause300mS pause time300mS
    #endif

    'A "pause" has 1mS units at default clock rate. Using pause here instead of pauseus gives top range.
    symbol time1S = 1000 ClockRatio
    #define Pause1S pause time1S
    symbol time3S = 3000 ClockRatio
    #define Pause3S pause time3S
    symbol time10S = 10000 ClockRatio
    #define Pause10S pause time10S

#Endregion

#Region "License"
#Rem

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

#EndRem
#EndRegion
```

Program Listing – “DF_Player_Mini.basinc”

```
#Region "Read Me"
#REM

            File: "DF_Player_Mini.basinc"
         License: MIT (See end of file)
  Change History: 2024/10/16 by Alan Hunt - 1st issue.


Description
===========
Designed for the DF Player Mini audio board, this module provides macros and subroutines to speak numbers and play audio tracks to
emulate speech and play music and sound effects.

The DF Player module does not behave entirely as expected, so this module eases its use.  The busy pin takes time to become active and
effectively reports audio output without file seek time delays.  Testing also found the minimum pause times between different
instructions, so this module takes care of that too and shares the info with inline comments.  Using the module should mean that
individual words can be sequenced to be as quick and reliable as practical.

The list of macros include:
    PlayMP3(Track, Title)               Plays the Track after the previous track is finished. Title is for debugging/information.
    PlayNow(Track, Title)               Stops anything currently played or queued and plays the selected Track.
    PlayAdvert(Track, Title)            Plays the Advert and temporarily suspends anything currently playing.
    AudioSpeakNumber(Val)               Speaks the number Val, which is a Word value from 0 to 65,535.
    AudioSpeakTenths(Val)               Speaks the number Val divided by 10
    AudioSpeakHundreths(Val)            Speaks the number Val divided by 100
    AudioSpeakThousandths(Val)          Speaks the number Val divided by 1000

The list of subroutines include:
    AudioInitialise                     To be included in program initialisation.
    AudioWaitForIdle                    Wait for sound to stop.
    AudioClear                          Stop sound.
    _AudioPlayMP3                       Internal routine to play the track defined by _Audio_MP3Track.
    _AudioPlayAdvert                    Internal routine to play the advert defined by _Audio_AdvertTrack.
    _AudioSpeakInteger                  Internal routine to speak the number defined by _Audio_Integer.
    _AudioSpeakDecimalPlaces            Internal routine to speak a number as individual digits, as spoken after a decimal place.


Limitations
-----------
The DF Player Mini comms is 9,600bps, so the minimum clock speed for a PICAXE chip is 8MHz, this is the X2 default whereas an M2 needs
setting to run faster than default.

The DF Player Mini can support several folders and thousands of files but this module is limited to using the default MP3 and ADVERT
folders, so a maximum of 256 files in each.

Even with trimming silences, chaining tracks with individual words is a little staccato.  For personal use it's quick and very clear but
not smooth, or so impressive for others.  To get the best results use joined phrases, such as "The total solar power is" {n} "kilo watt
hours over the last week".  Otherwise, you need solutions to replace the DF Player, such as online interfaces or a far more powerful
speech synthesis device.


Dependencies
------------
Certain pause statements and symbols must be defined for timing, pin usage and variable usage, see the Programming Guide section.



About the DF Player Mini
========================

Sourcing the hardware
---------------------
The DF Player Mini can be obtained from many sources and in many variants.  Some of these are unreliable or appeared faulty.  Here's a
summary of types I compared:

1) Good: The PICAXE SPE035, labelled "DF Player Mini", with processor "YX5200-24SS, OE64 1607" provided good results and this utility is
based on its timings.  Like other units, completion of a track is reported twice from its serial output and the track number appears to
be a random number, which is perhaps the file position in the SD card FAT.
2) Good: The "DF Player Mini, VO5.1, HW-247A" with processor "TD5580A".  Initially I thought only 1 of 3 modules worked but this was due
to timing differences from (1).  The differences I noted were that:  Boot-up delays were more than 100mS, so the subroutine
AudioIniailise now pauses for 1 second; the Busy signal went active earlier but stayed on later, typically 160mS after the modules track
completion message; and there's a 10ms gap in the double reporting of track completion.
3) Bad: A batch labelled "MP-TF-16P-V3.0" with processor "MH2024K-24SS, 240113" were not usable.  They were very slow, only reported
track completion occasionally and did not reliably accept track requests.
4) Based on the findings above I'd buy from an electronics store unless I had time on my hands to check cheap packs from eBay, Amazon,
AliExpress etc.


Some Reference Notes
--------------------
The DF Player Mini is loaded with an SD card containing .mp3 or .wav tracks in the root MP3 folder, these are then easily selected for
playback.  The filename simply must begin with a number, I use 0000 to 0255 followed by a space and a short description.  Likewise,
numbered Advert tracks can also be played and these temporarily suspend the main playing track.  The Adverts have uses such as playing a
crash sound during a game and they are placed in the root ADVERTS folder.  Adverts are a really cool feature!

* The "DF Player Mini" supply is 3.2V to 5.0V.  TTL is 3.3v level so a 1kOhm series must be placed between its Rx pin and 5V signals.
* At startup the module may need up to 4 seconds to boot and load a very long file list.
* The Busy signal only indicates audio output.  After requesting a track to be played there's a delay before it goes active.
* Various commands take time to respond or need short wait periods before another command is supplied.  This module takes care of it and
includes comments on them.
* The serial comms is True(Idle high), 9600 bps.  This module needs the baud rate parameter adjusting when PICAXE clock speed changes
from 8MHz.
* The command packet format is: $7E(Start), $FF(Version), $06(Length), $xx(Command), $00(No feedback), $xx(Param1), $xx(Param2),
{Optional 2 byte checksum},$EF(End)
    - Normally the optional checksums are not used, the packet length stays as $06 regardless.
    - If required, the checksum is easily calculated as zero minus each of the byte values in the message.
* Manuals are widely available on the Internet, including https://picaxe.com/docs/spe033.pdf.
* A simple starter kit is shown here: https://picaxe.com/docs/spe035.pdf.
* A nice little weatherproof speaker is the Eagle B185.
```

```
* The speaker volume is software controllable, and the maximum is very loud indoors.  Outdoors it won't carry far in a noisy
environment.
* Playing an Advert when an advert is already running seems to either clear the underlying track or cause longer delays on the Busy
signal (I haven't investigated this).



Programming Guide
=================
The main 3 macros PlayMP3, PlayNow and PlayAdvert include a Title parameter, which allows short descriptive text to be passed.  The
Title can greatly improve readability of your program, and it consumes no program memory if debugging remains commented out, e.g. if
track 29 is configured to say "thousand" the program statement to use it is PlayMP3(29,"thousand").  The Title output can be helpful
when debugging is enabled, but program memory will diminish very quickly, so the text should be short.

This module requires the following pause statements to be defined Pause3mS, Pause30mS, Pause100mS, Pause300mS and Pause1S.  These can be
defined in your main program with pre-processor directives like '#define Pause3mS pause 3'; or instead, you can '#include
"Timing.basinc"' before this file, which will provide these statements and give consistent timing across different clock rates.

When changing PICAXE clock rates be sure to change the AUDIO_BAUD constant too, valid settings are T9600_8, T9600_16, T9600_32 and, if
using X2 chips, T9600_64.

Using phrases instead words for most tracks will greatly reduce program memory space.  It will also make speech smoother and the code
more readable with fewer lines.

When reading voltage levels it's tempting to use millivolts but this adds extra words like "thousand", "and" and "hundred".  If you have
a variable representing millivolts, try using AudioSpeakThousandths(Val) because "12325" would be spoken as separate digits like twelve-
point-three-two-five, which is quicker and seems natural with a little staccato.

Your program can and probably should work asynchronously with the DF Player.  In the sample config below, you can see the DF Player Busy
Pin is connected to the PICAXE B.2 input, which is symbolised by AUDIO_STATUS; and the Busy Pin idle state of 1 is symbolised by
AUDIO_IDLE.  So for example, in a main program loop you can test to see if a music track is finished and play the next track with "if
AUDIO_STATUS = AUDIO_IDLE then {.....} endif".  Taking this a stage further you could probably trigger interrupts on the pin represented
by AUDIO_STATUS but take care of time delays in this going active and inactive.

You should be careful that your program isn't blocked unintentionally by the PlayMP3(Track, Title) macro.  This macro waits for any
existing track to complete before it sends its track request to the DF Player.  PlayNow is an alternative macro, but this simply stops
anything being played and plays what it wants.


My preferred program structure is:

    1) Program Notes
    2) Compiler Directives
    3) Resources
        3a) Pins
        3b) Variables
        3c) Constants
        3d) Modules
    4) Initialisation
    5) Main:
    6) Interrupt:
    7) Subroutines


Based on the program structure above, here is an example configuration for this module:

#Region "Resources"
'Pins
    symbol AUDIO_TX = B.0               '@DF_Player_Mini.basinc: PICAXE Serial Tx to DF Player via 1K resistor.
    symbol AUDIO_STATUS = pinB.2        '@DF_Player_Mini.basinc: Busy indication from DF Player is active low after about 230mS.

'Variables
    '@DF_Player_Mini.basinc Variables
    symbol _Audio_Digit = b43           'Assign symbol to any spare Byte.
    symbol _Audio_Integer = w22         'Assign symbol to any spare Word.
    symbol _Audio_Fraction = w23        'Assign symbol to any spare Word.
    symbol _Audio_Track = b48           'Assign symbol to any spare Byte.
    symbol _Audio_Byte = b49            'Assign symbol to any spare Byte.

'Constants
    '@DF_Player_Mini.basinc Constants
    symbol AUDIO_VOLUME = 15            '0 to 31(Maximum).  You can assign this symbol to a variable if you don't want fixed volume.
    symbol AUDIO_BAUD = T9600_8         'Tx logic is idle high (T) and 9600 bps, adjust the last character(s) to your clock speed in
MHz.
    symbol AUDIO_ACTIVE = 0             'The AUDIO_STATUS pin is active low, indicating audio playing.
    symbol AUDIO_IDLE = 1               'AUDIO_STATUS high indicates the module is idle OR seeking a track to play on the disk.

'Modules
    #include "Timing.basinc"
    #include "DF_Player_Mini.basinc"

'Initialisation
    gosub AudioInitialise               'Prepares PICAXE output pin, Resets the DF Player, selects SD input and sets volume.

#Endregion

#EndREM
#EndRegion

#Region "Program"

goto DF_Player_End                                  'This module only contains macro and subroutine definitions.

'Debugging
'=========
'Caution, debugging provides some nice info but causes serial transmission delays and consumes program space.
'#define DebugAudio_On                              'Comment out this line to turn debugging off.
#ifdef DebugAudio_On
    #define DebugAudio sertxd
#else
    #define DebugAudio 'Do nothing
#endif
```

```
'Audio Macros
'============
    #macro PlayMP3(Track, Title)
        'DebugAudio("@PlayMP3 ",#Track,Title,cr,lf)     'Enabling this line consumes program space extremely quickly with track title
info.
        _Audio_Track = Track
        gosub AudioWaitForIdle
        gosub _AudioPlayMP3
    #endmacro
    #macro PlayNow(Track, Title)
        'DebugAudio("@PlayNow ",#Track,Title,cr,lf)     'Enabling this line consumes program space extremely quickly with track title
info.
        _Audio_Track = Track
        gosub AudioClear
        gosub _AudioPlayMP3
    #endmacro
    #macro PlayAdvert(Track, Title)
        'DebugAudio("@PlayAdvert ",#Track,Title,cr,lf)  'Enabling this line consumes program space extremely quickly with track title
info.
        _Audio_Track = Track
        gosub _AudioPlayAdvert
    #endmacro
    #macro AudioSpeakNumber(Val)
        _Audio_Integer = Val
        gosub _AudioSpeakInteger
    #endmacro
    #macro AudioSpeakTenths(Val)
        _Audio_Integer = Val / 10
        _Audio_Fraction = Val % 10
        gosub _AudioSpeakInteger
        PlayMP3(031,"point")
        _Audio_Byte = 1
        gosub _AudioSpeakDecimalPlaces
    #endmacro
    #macro AudioSpeakHundreths(Val)
        _Audio_Integer = Val / 100
        _Audio_Fraction = Val % 100
        gosub _AudioSpeakInteger
        PlayMP3(031,"point")
        _Audio_Byte = 2
        gosub _AudioSpeakDecimalPlaces
    #endmacro
    #macro AudioSpeakThousandths(Val)
        _Audio_Integer = Val / 1000
        _Audio_Fraction = Val % 1000
        gosub _AudioSpeakInteger
        PlayMP3(031,"point")
        _Audio_Byte = 3
        gosub _AudioSpeakDecimalPlaces
    #endmacro


'Audio Subroutines
'=================
AudioInitialise:
    DebugAudio("@AudioInitialise",cr,lf)
    high AUDIO_TX                                       'Output pins are default low, so this prepares communication.
    Pause1S                                             'Some types of DF Player need more than 100mS boot time.

    serOut AUDIO_TX, AUDIO_BAUD, _
        ($7E, $FF, $06, $0C, $0, $0, $0, $EF)           'Command $0C (Reset)
    Pause1S                                             'AUDIO_ACTIVE after 220mS and module messages going to sleep at 900mS.

    serOut AUDIO_TX, AUDIO_BAUD, _
        ($7E, $FF, $06, $09, $0, $0, $2, $EF)           'Command $09 (Playback source), Arg $0002 (SD card)
    Pause1S                                             'Pause to read SD card

    serOut AUDIO_TX, AUDIO_BAUD, _
        ($7E, $FF, $06, $06, $0, $0, AUDIO_VOLUME, $EF) 'Command $06 (Set Audio volume)
    Pause100mS                                          'Pause to ensure volume set
    return

AudioWaitForIdle:
    DebugAudio("@AudioWaitForIdle",cr,lf)
    do while AUDIO_STATUS = AUDIO_ACTIVE
        Pause3mS
    loop
    return

AudioClear:
    DebugAudio("@AudioClear",cr,lf)
    serOut AUDIO_TX, AUDIO_BAUD, _
        ($7E, $FF, $06, $15, $0, $0, $0, $EF)           'Command $15.  Stop Advert(Crash), if playing.
    Pause100mS                                          '75mS required for settling time.
                                                        'NB: Contrary to manual, command $16 did not stop everything.
    serOut AUDIO_TX, AUDIO_BAUD, _
        ($7E, $FF, $06, $16, $0, $0, $0, $EF)           'Command $16.  Stop main track, if playing
    Pause30mS                                           '14mS required before requesting the new track
    return

_AudioPlayMP3:
    DebugAudio("@_AudioPlayMP3 - ",#_Audio_Track,cr,lf)
    serOut AUDIO_TX, AUDIO_BAUD, _
        ($7E, $FF, $06, $12, $0, $0, _Audio_Track, $EF) 'Command $12.  Play from MP3 folder.
    do while AUDIO_STATUS = AUDIO_IDLE                  'Ensures AUDIO_STATUS caught up (seen delays up to 290mS)
        Pause3mS
    loop
    return

_AudioPlayAdvert:
    'Known issue:
    '   - This applies to the PICAXE SPE035 hardware, but not the DF Player Mini, VO5.1, HW-247A.
    '   - When an advert tries to play over the top of another advert the underlying MP3 track is lost.
    '   - Finally, a bug I liked because it kept my Buzz Wire Game more active and interesting by shifting to the next track :)
```

```
    DebugAudio("@_AudioPlayAdvert - ",#_Audio_Track,cr,lf)
    serOut AUDIO_TX, AUDIO_BAUD, _
        ($7E, $FF, $06, $13, $0, $0, _Audio_Track, $EF) 'Command $13.  Play ADVERT (suspends current track while playing).
    Pause300mS                                          '170mS required because AUDIO_STATUS goes Idle before main track resumed.
    return

_AudioSpeakInteger:
    DebugAudio("@_AudioSpeakInteger - ",#_Audio_Integer,cr,lf)
    'Thousands
    _Audio_Byte = _Audio_Integer / 1000
    if _Audio_Byte > 0 then
        gosub _AudioSpeak1to99
        PlayMP3(29,"thousand")
    endif

    'Hundreds
    _Audio_Digit = _Audio_Integer dig 2
    if _Audio_Digit > 0 then
        PlayMP3(_Audio_Digit,"aDigit[0to9]")
        PlayMP3(28,"hundred")
    endif

    'Units
    _Audio_Byte = _Audio_Integer % 100
    if _Audio_Integer > 99 AND _Audio_Byte > 0 then
        PlayMP3(0030,"and")
    endif
    gosub _AudioSpeak1to99
    if _Audio_Integer = 0 then
        PlayMP3(0,"zero")
    endif

    return

_AudioSpeakDecimalPlaces:
    DebugAudio("@_AudioSpeakDecimalPlaces - ",#_Audio_Byte," places for ",#_Audio_Fraction,cr,lf)
    dec _Audio_Byte
    for _Audio_Digit = _Audio_Byte to 0 step -1
        _Audio_Track = _Audio_Fraction dig _Audio_Digit
        PlayMP3(_Audio_Track,"aDigit[0to9]")
    next _Audio_Digit
    return

_AudioSpeak1to99:
    'Speak numbers 1 to 99 (zero is not sounded here).  NB: "dig" operator needs to be the first or only argument.
    _Audio_Digit = _Audio_Byte dig 1
    if _Audio_Digit >= 2 then
        _Audio_Track = _Audio_Digit + 18
        PlayMP3(_Audio_Track,"aDecade[20to90]")         'Twenty=20, Thirty=21 etc
        _Audio_Track = _Audio_Byte dig 0
        if _Audio_Track > 0 then
            PlayMP3(_Audio_Track,"[0to9]")
        endif
    else
        if _Audio_Digit = 1 then
            _Audio_Track = _Audio_Byte dig 0 + 10
            PlayMP3(_Audio_Track,"[10to19]")
        else
            _Audio_Track = _Audio_Byte dig 0
            if _Audio_Track <> 0 then
                PlayMP3(_Audio_Track,"[0to9]")
            endif
        endif
    endif
    return

DF_Player_End:
#Endregion

#Region "License"
#Rem

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

#EndRem
#EndRegion
```

Program Listing – “Voltage_X2.basinc”

```
#Region "Read Me"
#REM

           File: "Voltage_X2.basinc"
        License: MIT (See end of file)
 Change History: 2024/10/23 by Alan Hunt - Corrected typos of Denominator to Dividend.
                 2024/10/22 by Alan Hunt - 1st issue.


Description
===========
This module provides some macros that calculate the power supply voltage and ADC input voltages with the 10bit ADC channels on an PICAXE
X2 chip.  This is achieved with good accuracy and minimal program space and there's advice for a calibration method included in the
notes too.

Hopefully the macros are easy to use and understand, but sorry M2 chips are not supported.

The list of macros included:
    GetVdd                      A word variable symbolised by Voltage_Vdd is updated with the power supply voltage in mV.
    SetVdd(Val_mV)              If the power supply is accurately known and stable, Voltage_Vdd can be set manually.
    VoltsByVdd(Var, Channel)    Updates a word variable, Var, with the voltage in mV from the specified Channel number.


Limitations
-----------
M2 chips do not have the X2 operator "*/" that is used here for simplification.  It keeps the code simple and short by obtaining the
middle word of a word multiplication, which goes most of the way towards making a division of 1024 while achieving good accuracy on the
maths.

In the PICAXE Forum there are several solutions to calculate Vdd and ADC input voltages with some accuracy.  There are methods to
average ADC readings, or using large dividends and divisors to make the integer math easier, or a neat method of using the DAC to obtain
higher value ADC readings for more accuracy etc.  Sadly, there's no DAC on my 20X2 and I was extremely short on program space too, even
after swapping out a 20M2.

The key problem in calculating Vdd seems to be the accuracy of the PICAXE 1.024V FVR (Fixed Voltage Reference).  In my case it was 1.1%
high, which is expected as per Microchip application note AN1072 and datasheet PIC18(L)F1XK22.  If the power supply is known and stable
it's preferable to use a fixed constant for Vdd to calculate ADC input voltages, which is what the SetVdd macro does.  If the supply
voltage varies, perhaps being connected directly to a battery, another option to improve accuracy is to calibrate, as shown in a section
below.

Using the FVR to calculate Vdd presents another issue with the range of voltage uncertainty for each ADC step.  For example, each step
is 5mV with a 5V supply and there's perhaps 1 bit error making 10mV, then because the calculated Vdd is roughly 5 times larger this
value could be 50mV out.  A possible approach could be to reduce supply voltage making smaller ADC steps and less multiplication of
uncertainty too.  In terms of percentage accuracy it's also worth noting that you need lots of steps, so you can't measure small
voltages with 1% accuracy and you'd probably want a minimum ADC input of 1V when using a 5V supply.

The good news is that the ADC does seem to be reasonably accurate and linear.  Measured with two 20X2 chips, if anything the readings
start a little early perhaps by 1mV and finish early too by about 3mV.  At these voltage levels it's difficult to be accurate and noise
will be a factor, but the readings were taken with an OWON XDM1041 bench multimeter and verified with a handheld digital multimeter.
With 5.000V measured on the 20X2 PICAXE supply pins and using SetVdd(5000), the ADC was expected to read 1 between 2.4mV and 7.3mV but
this reading started from a 1mV input.  Also, an ADC reading of 1023 was expected after 4997mV but the reading started to be seen at
4993mV and was consistent at 4994mV.  The table of the results is below:

ADC mV | ADC Val | VoltsByVdd mV Value
---------------------------------------
5000     1023     4995
4994     1023     4995
4993     1022     4990 (occasional 1023, 4995mV)
4000      819     3999
3000      614     2998
2000      410     2001 (some 409, 1997mV)
1000      205     1000
0200       41      200
0100       21      102
0050       10       48
0027        6       29
0021        5       24
0016        4       19
0012        3       14
0007        2        9
0006        1        4 (occasional 2, 9mV)
0002        1        4
0001        0        0 (occasional 1, 4mV)
0000        0        0


Dithering around ADC step changes is a commonly reported issue. I found capacitance on the ADC input didn't help much when there's only
2 obvious ADC values at play.  If the readings are fluctuating more than this it probably points to the dynamics of your setup, like
varying supply loads and ADC inputs, or noise.


How it Works
------------
The maximum ADC value is 1023 and in basic configuration this approximately represents the PICAXE supply voltage, Vdd.  Actually, the
specification states this should be a half value, so 1023 represents a voltage over 2047/2048 of Vdd, whereas a value of 1 should
represent 1/2048 to 3/2048 of Vdd.  In testing the tipping point to 1023, it was found to be just under 1023/1024 of Vdd, which is a
good approximation.

To measure Vdd this module simply takes an ADC reading for the internal FVR of 1.024V and scales that up.  The math is
Vdd=1023*1.024/Val_ADC, but the ADC value is quite large divisor for integer maths.  Dependent on the chip and its supply voltage range
of perhaps 1.8V to 5.2V, the ADC reading will be between 581 (1023*1.024V/1.8V) and 201 (1023*1.024V/5.2V).  The dividend is therefore
scaled up to 10476, which is a rounding of 1023*1.024*10.  The remainder of this division is then scaled up by 100 for the final part of
the division.  The integer result is then 1,000 times bigger and accurate to the last digit, meaning the answer for Vdd is given in
millivolts.

To calculate an ADC input voltage, the ADC reading is just scaled according to Vdd with ADC_Volts=Vdd*ADC_Val/1024.  This formula gives
the lowest possible voltage at the input.  The math is performed with the X2 operator "*/", which returns the middle word of a
multiplication.  This effectively divides the result by 256 and this is then divided by 4 to get the overall division of 1024.
```

Calibration

The fixed voltage reference of 1.024V can vary by several percent between chips and it's also dependent on supply voltage and temperature. To halve the supply voltage errors across the possible range, Microchip AN1072 recommends calibrating at 4V. Coincidently, 4V calibration is good when monitoring charge voltages on Li-ion cells too.

To calibrate, set the supply voltage to 4000mV as monitored on the PICAXE supply pins. Uncomment the line "'#define VoltageDebug_On", which will output information to the PICAXE Editor Terminal, then repeatedly call GetVdd from your program with 1 second pauses. Note the ADC Value and adjust the supply up and down so you find voltages where the ADC value occasionally reads 1 higher and 1 lower. The midpoint between these voltages is used to calculate a constant named VoltageDividend, using the formula VoltageDividend=Midpoint*ADC_Value*10. If your FVR is accurate the reading should be 10476. In my case, two 20X2 chips gave an ADC reading of 265, the edge voltages were 3991mV and 4004mV, the chosen midpoint was 3998mV, so VoltageDividend was 10595. The FVR spec for the 20X2 chip is -8% to +6% and the error seen was +1.1%.

Programming Guide
=================
You should use the ADCSETUP command to disable digital circuitry on the ADC Channels that you plan to use, see Manual 2.

You must use the GetVdd or SetVdd macro before using the VoltsByVdd macro. If you are using a stable regulated power supply, then it's only necessary to do this once. The SetVdd macro is preferred if you have an accurate and stable supply to the PICAXE, the macro can be also be used for testing too. If the PICAXE is connected to a battery where the supply voltage will change over time, then the GetVDD macro should be used immediately before every VoltsByVdd reading.

To debug values to the PICAXE Editor Terminal window just uncomment this line in the module "'#define VoltageDebug_On".

When monitoring battery voltages be careful not to Phantom Power your circuit through voltage protection diodes on ADC channels. Ideally, batteries should be fully disconnected when your circuit is off to prevent battery drain, otherwise connect the ADC channel via a 1K to 10K resistor to protect the PICAXE chip. Microchip recommends a minimum source impedance of 10K for ADC inputs.

An example configuration is shown below and these statements should appear in your program before the macros are used:


```
#Region "Resources"
'Variables
    '@Voltage_X2.basinc Variables
    symbol Voltage_Vdd = w25                'Assign this symbol to any spare Word, it stores PICAXE supply voltage as mV.
    symbol _VoltageADCval = w26             'Assign this symbol to any spare Word, it's an internal variable for maths.
    symbol _VoltageModulus = w27            'Assign this symbol to any spare Word, it's an internal variable for maths.

#Constants
    '@Voltage_X2.basinc Constant
    symbol VoltageDividend = 10595       'Adjust value for FVR inaccuracy, see "Voltage_X2.basinc" Calibration.

'Modules
    #include "Voltage_X2.basinc"
#EndRegion

#EndREM
#EndRegion

#Region "Program"

goto VoltageEnd                             'There is no code to execute, just macro definitions for the compiler.

'Debugging
'=========
'Caution, debugging provides some nice info but causes serial transmission delays and consumes program space.
'#define VoltageDebug_On                    'Uncomment this line to turn debugging on.
#ifdef VoltageDebug_On
    #define _VoltageDebug sertxd
#else
    #define _VoltageDebug 'Do nothing
#endif

'Macros
'======
#macro GetVdd
    _VoltageDebug("@GetVdd")
    calibadc10 _VoltageADCval                                   'Read reference 1.024V (Value = 1023 * 1.024V / Vdd)
    Voltage_Vdd = VoltageDividend / _VoltageADCval * 100     'Obtain first 2 digits, nnxx. Max=5200(5211), Min=1800(1803)
    _VoltageModulus = VoltageDividend // _VoltageADCval      'Find remainder. Max=580, Min=0
    Voltage_Vdd = 100 * _VoltageModulus / _VoltageADCval + Voltage_Vdd  'Obtain last 2 digits, xxnn. Max=99(=100*580/581), Min=0
    _VoltageDebug("  (Ref_1.024V=",#_VoltageADCval,",  Vdd=",#Voltage_Vdd," mV)",cr,lf)
#endmacro

#macro SetVdd(VoltsW)
    _VoltageDebug("@SetVdd  (Val=",#VoltsW,")",cr,lf)
    Voltage_Vdd = VoltsW
#endmacro

#macro VoltsByVdd(VoltsW, Channel)
    'The measured voltage is Vdd(1800 mV to 5200 mV) * ADC_Reading(0 to 1023) / 1024(ADC_Range).  NB: ADC range is approximately 1024
    _VoltageDebug("@VoltsByVdd")
    readadc10 Channel, _VoltageADCval               'Read ADC Channel, the range is 0 to 1023 from Gnd to Vdd.
    VoltsW = Voltage_Vdd */ _VoltageADCval          'Get the middle word of Vdd * ADC10, so the result is divided by 256.
    VoltsW = VoltsW / 4                             'Then divide by 4, now we have Vdd * ADC_Reading / 1024.
    _VoltageDebug("  (ADC",#Channel,"=",#_VoltageADCval,",  Voltage=",#VoltsW," mV)",cr,lf)
#endmacro

VoltageEnd:
#Endregion

#Region "License"
#Rem

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
```

```
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

#EndRem
#EndRegion
```