### Preface

I will go over some practical examples that I came up with myself, and some that others have posted publicly that they have used in their hacks. You should almost already know everything about asm from the first few tutorials aside from some game specific things, this will just help solidify your foundation and explain how common functions/variables are used.

### Explained examples

Ex. 1 Play sound when mario touches obj

This will be a useful code to change music when entering an area. Something like this code is used in many hacks to have different music in different sections of levels.

* .orga 0x1201000
* .dw 0x00040000
* .dw 0x08000000
* .dd 0x0C00000080401014
* .dw 0x09000000
* ADDIU SP, SP, -0X18
* SW RA, 0X14 (SP)
* LUI A0, 0X8036
* LW A1, 0X1158 (A0)
* JAL 0X8029E2E8 //DIST 3D, RETURN F0
* LW A0, 0X1160 (A0)
* CVT.W.S F0, F0
* MFC1 T0, F0
* SUBIU T0, T0, 0X100
* BGEZ T0, END
* LUI T1, 0X8036
* LW T1, 0X1160 (T1)
* LW A1, 0X188 (T1) //SEQ ID
* SW T1, 0X18 (SP)
* JAL 0X80320544 //PLAY SEQUENCE
* OR A0, R0, R0
* LW T1, 0X18 (SP)
* SW R0, 0X74 (T1)
* END:
* LW RA, 0X14 (SP)
* JR RA
* ADDIU SP, SP, 0X18

First I set the behavior script of my object up. I use the compiler to write bytes to the ROM using “.dw” for words and “.dd” for doubles. The location is just in generic free space in the extended ROM, it can be anywhere you need it to be.

In the function, I load mario and my object and call a function that calculates the distance between them. A0 and A1 have to be the two objects, order does not matter. The function returns F0 as the distance. I convert F0 to an int, subtract 0x100 and check if its greater than zero. If I branch then mario is over 0x100 units away, else he is closer than that. I use a function to then play a new sequence ID and delete the object. I load the Bparam and use that as the ID number. In an object editor, you would want to use Bparam4. This is because each Bparam is a different byte of the same word. If you wanted to set a specific Bparam to be the seq ID you would have to load the corresponding byte. 0x188 is Bparam1, 0x189 is Bparam2 etc. A0 on that function tells the game whether or not to play the songs at the same time. This is used to play a star get or switch press song on top of normal music, we set it to zero to not use that functionality.

Before we call the function I store our object pointer in the stack so I don't have to load it again. I reload it afterwards so I can store zero in 0x74, this deactivates the object. You should do this or you will play the song every frame while mario is in range.

Ex. 2 Pay coins to make obj disappear

This is some code I’ve come up with for this tutorial. This object will be solid and block your way until you collect enough coins to remove it. You will push L to pay the coins for it to go away. You could easily use a thwomp anim, collision and model for a unique NPC in your level if you wanted to.

* .orga 0x1201000
* .dw 0x00090000
* .dw 0x11010011
* .dd 0x2A0000000E100000
* .dw 0x0E430500
* .dw 0x08000000
* .dd 0x0c000000803839CC
* .dd 0x0c00000080401050
* .dw 0x09000000
* .asciiz “Press L to pay” //0x102C
* .orga 0x1201050
* ADDIU SP, SP, -0X18
* SW RA, 0X18 (SP)
* LUI T0, 0X8034
* LH T1, 0XB218 (T0) //coins
* LUI T9, 0X8036
* LW T9, 0X1160 (T9)
* LW T8, 0X188 (T9)
* SLT T2, T8, T1
* BNE T2, R0, END
* ORI A0, R0, 0XA0
* ORI A1, R0, 0X60
* LI A2, 0x8040102C
* JAL 0X802D66C0
* NOP
* LUI T9, 0X8034
* LH T8, 0XAFA0 (T9)
* ANDI T8, T8, 0X20 // L
* BEQ T8, R0, END
* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LUI T9, 0X8034
* LH T1, 0XB218 (T9)
* LW T8, 0X188 (T0)
* SUBU T3, T1, T8
* SH T3, 0XB218 (T0) //COINS
* SW R0, 0X74 (T0)
* END:
* LW RA, 0X14 (SP)
* JR RA
* ADDIU SP, SP, 0X18

First I setup the behavior script as always. I set the object flags for 0x11, which just updates visuals to match the real position. I set the collision pointer with 0x2A. For this example I use some random place in bank 0xE. You will have to out your own collision there or maybe just a pushable metal box. I use 0x0E43xxxx to set the collision distance. This is important to reduce lag by not calculating collision a certain distance away. Finally I run the process collision and our custom function every frame.

Below the behavior I use “.asciiz” to insert a string that we will print with our object, I also wrote down its location.

In our function I first load our object, and the amount of coins mario has. I compare the Bparam to the coins and goto the end if Bparam is greater than our coin count.

* *SLT T2, T8, T1*

In the above line T2 will be set to one when T8 is less than T1, else T2 is set to zero. Since we load our Bparam to T8 and coins to T1 we have an expression that reads:

* If (Bparam<Coins)
* T2 = 1
* Else
* T2 = 0

We can now branch when T2 is zero to setup a condition that only executes when we have enough coins. Our condition is to print a string, then delete the object if we press L. We print a string using a function. A0 and A1 are locations on the screen and A2 is our string location. This will be the location we put our “.asciiz” line above. You can double check this in the function list on hack64.

Next we check the current button input (player 1 controller 0x8033AFA0), and see if L (value of 0x20) is pressed. If it is, we delete the object, else we do nothing.

This code has some issues. One is that there is no check for distance so it always prints a string when you have enough coins. It also does not tell you how many coins you need to destroy the block. These are simple issues that if you've been following along should be able to fix. I will leave them as an exercise for the reader.

Ex. 3 Text on 1 up collect (by aglab2)

This is code written by aglab2 to use 1ups as an easter egg in ZAR. If you have not played it, each level has a 1up that shows a unique message on collection.

* .orga 0x120AC00
* .dw 0x00040000
* .dw 0x08000000
* .dd 0x0C0000008040AD00
* .dw 0x09000000
* .orga 0x120AB00
* .asciiz "AHAHAHAHAHAAHAAHAHAHA"
* .orga 0x120AB20
* .asciiz "LITERALLY HANDLING GOD"
* .orga 0x120AB40
* .asciiz "I WISH IT WAS EASY"
* .orga 0x120AB60
* .asciiz "E\*PLORER IN BLOOD"
* .orga 0x120AB80
* .asciiz "SECRETS ARE FREE LOL"
* .orga 0x120ABA0
* .asciiz "PIRATE SHIP FROM MA"
* .orga 0x120ABC0
* .asciiz "WHO NEEDS ASSIST"
* .orga 0x120ABE0
* .asciiz "I WANNA SEE"
* .orga 0x120AC20
* .asciiz "IDIOT"
* .orga 0x120AC40
* .asciiz "NOTEBLOCKS ARE FUN"
* .orga 0x120AC60
* .asciiz "WIND IS SO DUMB"
* .orga 0x120AC80
* .asciiz "YOU FOUND A SECRET"
* .orga 0x120ACA0
* .asciiz "NO WAY IT IS POSSIBLE"
* .orga 0x120ACC0
* .asciiz "SHARP EYES ARE GOOD"
* .orga 0x120ACE0
* .asciiz "BREAKING COURSE FAIL"
* .orga 0x120AD00
* .area 0x100, 0x00
  + ADDIU SP, SP, -0x18
  + SW RA, 0x14(SP)
  + LW V1, 0x80361160
  + LW T0, 0x154(V1)
  + LI T1, 100
  + BEQ T0, T1, despawn
  + NOP
  + LI A0, 160
  + LI A1, 40
  + jal 0x802D66C0
  + LW A2, 0x188(V1)
  + B end
  + NOP
* despawn:
  + SW R0, 0x74(V1)
* end:
  + LW RA, 0x14(SP)
  + JR RA
  + ADDIU SP, SP, 0x18
* .endarea
* .orga 0x21DEA8
  + .dd 0x0C0000008040AA00
* .orga 0x120AA00
* .area 0x100, 0x00
  + ADDIU SP, SP, -0x18
  + SW RA, 0x14(SP)
  + JAL 0x802F45B8
  + NOP
  + LW V1, 0x80361160
  + LW T0, 0x74(V1)
  + LI T1, 0xFFFF0000
  + AND T2, T0, T1
  + BNE T2, R0, fini
  + NOP
  + ADD A0, R0, V1
  + LI A1, 0
  + LI A2, 0x0040AC00
  + JAL 0x8029EDCC
  + NOP
  + LW V1, 0x80361160
  + LW T0, 0x188(V1)
  + SW T0, 0x188(V0)
* fini:
  + LW RA, 0x14(SP)
  + JR RA
  + ADDIU SP, SP, 0x18
* .endarea

First the behavior is setup. The behavior calls 0x8040AC00 every frame then ends. Next aglab takes advantage of the compiler to easily add text. You should notice that each string is seperated in the ROM by 0x20 bytes, this is mostly to easily reference each string.

Next the asm function for 0x8040AC00 is written. The “.area” line just tells the compiler to limit the code to a certain length. First the object timer is loaded, 0x154. Then he checks if it is equal to 100 (not 0x100). Next he branches to the tag “despawn” when it is equal, otherwise he runs a piece of code that goes to the end. First on the “despawn” tag it is obvious that he despawns the object then ends the function. 0x74 controls whether or not an object is active. Set it to zero to despawn it, 0x0101 to be active. There may be other states I am unaware of as well used for specific objects.

The code he runs before despawning loads two decimal numbers and the Bparam, executes a function then goes to the end. The function he executes is called “printXY”. In our function reference we see that “printXY” takes A0 as an X position, A1 as a Y position and A2 as the location of the string in RAM. Here you can see that A2 is the Bparam, so each instance of the object can have a unique message. This message stays on screen for 100 frames then gets deleted.

The next function he injects into the normal 1up mushroom behavior script. He then runs the normal 1up mushroom function with the first *JAL*. Next he checks 0x74 and sees if it is zero. If it is not then the mushroom is exactly as vanilla, if it is, then he runs his piece of code before the object deactivates.

Note that since the branch condition is 0x74 == 0, it would normally never activate. This is because the functions called from a behavior of a deactivated object don't run. This one will run because the vanilla 1up mushroom function sets 0x74 to zero when mario touches it, then within the same function runs the rest of the code including the 0x74 branch condition. If he had set 0x74 to zero in a separate function, this code would never run.

The code he runs when 0x74 is zero simply spawns an object (0x8029EDCC). A0 is the object you are spawning from, A1 is the model ID and A2 is the behavior of the spawned object. You'll notice that A2 is set to the behavior he set at the beginning (0x8040AC00 is the same as 0x0040AC00 because bank 0 starts at RAM start). All together this will print a message of his choice for 100 frames once you collect a 1up mushroom, in addition to granting a life and playing the 1up SFX.

Ex. 4 Moving platform with with inertia

This is some code I’ve written for this tutorial specifically. It will be how to move a platform but when mario jumps off the platform he will gain the speed from it. Normally jumps do not have inertia so this will fix this. Objects use 0x116 and 0x11E to move mario while he stands on them so it will just be a matter of using these after mario jumps.

* .orga 0x1201000
* .dw 0x00 09 00 00,
* 0X11 01 00 11,
* 0x2A 00 00 00, 0x0E 01 00 00,
* 0x08 00 00 00,
* 0x0c 00 00 00, 0x80 38 39 CC,
* 0x0c 00 00 00, 0x80 40 10 40,
* 0X09 00 00 00
* ADDIU SP, SP, -0X18
* SW RA, 0X14 (SP)
* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LW T1, 0X188 (T0)
* MTC1 T1, F0
* CVT.S.W F0, F0
* JAL 0X802A1308 //VECTOR DISSECTION
* SWC1 F0, 0XB8 (T0)
* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LWC1 F2, 0XAC (T0)//X SPD
* LWC1 F4, 0XB4 (T0)//Z SPD
* CVT.W.S F2, F2
* CVT.W.S F4, F4
* MFC1 T4, F4
* MFC1 T2, F2
* SW T4, 0X116 (T0)
* JAL 0X802A2348 //MOVE OBJ
* SW T2, 0X11E (T0)
* JAL 0X802A3CFC//is mario on this obj (must be solid)
* NOP
* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LW T1, 0X14C (T0)
* ORI T5, R0, 0X2
* BEQ T1, T5, AIRDECAY
* NOP
* BNE T1, R0, INERTIA
* NOP
* BEQ R0, V0, END
* ORI T9, R0 ,0X1
* RESET:
* BEQ R0, R0, END
* SW T9, 0X14C (T0)
* INERTIA:
* BNE V0, R0, END
* //mario is no longer on block
* LWC1 F2, 0XAC (T0)
* LWC1 F4, 0XB4 (T0)
* LUI T9, 0X8034
* LWC1 F6, 0XB1AC (T9)
* ADD.S F2, F2, F6
* SWC1 F2, 0XB1AC (T9)
* LWC1 F6, 0XB1B4 (T9)
* ADD.S F4, F4, F6
* SWC1 F4, 0XB1B4 (T9)
* ORI T2, R0, 0X2
* BEQ R0, R0, END
* SW T2, 0X14C (T0)
* AIRDECAY:
* BNE V0, R0, RESET
* OR TO, R0, R0
* LW T5, 0X154 (T0)
* ORI T9, R0, 0X40 //~ 2 SEC
* BEQ T9, T5, ENDDECAY
* LUI T9, 0X8034
* LW T8, 0XB17C (T9)
* ANDI T7, T8, 0X1C0
* ORI T8, R0, 0X80
* BNE T8, T7, END
* MTC1 T5, F8
* CVT.S.W F8, F8
* LWC1 F0, 0XAC (T0)
* DIV.S F0, F0, F8
* LWC1 F2, 0XB4 (T0)
* DIV.S F2, F2, F8
* LWC1 F6, 0XB1AC (T9)
* ADD.S F2, F0, F6
* SWC1 F2, 0XB1AC (T9)
* LWC1 F6, 0XB1B4 (T9)
* ADD.S F4, F2, F6
* SWC1 F4, 0XB1B4 (T9)
* END:
* LW RA, 0X14 (SP)
* JR RA
* ADDIU SP, SP, 0X18

First thing that I do is set up the behavior. I use a dummy collision pointer for this example. Next in the function I take the Bparam and store it in this object's horizontal speed(hspd). The hspd is then converted into x and z speed using a vector dissection formula and then another function takes the x and z speed and processes the change in position.

Next I check if mario is on the object and check the objects action. The action starts at zero, and in while it is zero if mario isn't on the object then it stays zero. If mario is on the object then it gets set to one.

Next while the action is one I check if mario is still on it. If he is then nothing happens. If he gets off, I set the action to two and then increase his position by the objects x and z speeds. I do this instead of editing marios speeds because I don't want to interfere with the calculations the player input does. This will basically be an isolated speed and while it won't be affected by player input it should be fine.

Next when the action is two I just linearly decay the speed I add to mario. I divide it by the time and after 2 seconds I no longer add any. I also check if mario is still in the air, which is an important check to do. If mario lands on the block again before 2s I set the action to one again.

Ex. 5 Custom Bowser Battle (aglab2 code)

By controlling the action variable of bowser you can control him very easily. Unlike mario actions, bowser almost never bugs out when switching to whatever action you want and plays nice. Here is a reference of bowser action states that I have made.

* //0x14c:bowser action states
* //0=random attack, switches state to 11/F next frame
* //1=tail grab/in air
* //2=fell off arena/jumping back
* //3=bowser dance
* //4=defeated
* //5=talking to mario
* //6=walking forward for cutscene
* //7=running towards mario
* //8=bowser 3 blue fire eruption thing
* //9=single flame spit
* //A=teetering on edge
* //B=turn around
* //c=hit by bomb
* //D=jumping straight up
* //E=walking
* //F=breathing fire forwards
* //10=Teleport
* //11=jumping forward attack
* //12=idle
* //13=random attack, switches state to 11/F next frame
* //14=crash

I have used these for minor custom objects in some minihacks and in ASA, but not to that great an effect. A much more noticeable example is how aglab2 used this in ZAR for 3 custom bowser fights. I am sure that kaze also used these for Last Impact and for SM64XOOT but kaze code is not open source so I can't reference it. I would also not be surprised if Last Impact did some crazy thing that is completely un-intuitive since it was made during the Wild West of asm hacking. Aglab’s code is simple and efficient, so we will use his.

* .orga 0x120F600
  + .dw 0x00040000
  + .dd 0x0C0000008040F680
  + .dw 0x08000000
  + .dd 0x0C0000008040F700
  + .dw 0x09000000
* .orga 0x120F680
* .area 0x80
  + ADDIU SP, SP, -0x18
  + SW RA, 0x14(SP)
  + ; search bowser
  + LI A0, 0x13001850
  + JAL 0x8029F95C
  + NOP
  + LW V1, 0x80361160
  + SW V0, 0x100(V1)
  + LW RA, 0x14(SP)
  + JR RA
  + ADDIU SP, SP, 0x18
* .endarea
* .orga 0x120F700
* .area 0x100
  + ADDIU SP, SP, -0x18
  + SW RA, 0x14(SP)
  + LW V1, 0x80361160
  + ; let bowser start up
  + LW T0, 0x154(V1)
  + SLTI AT, T0, 10
  + BNEZ AT, end
  + NOP
  + ; load bowsey and his data
  + LW A0, 0x100(V1)
  + LW T0, 0x14C(A0)
  + ; init cs
  + LI T1, 0x6
  + BEQ T0, T1, end
  + NOP
  + LI T1, 0x5
  + BEQ T0, T1, end
  + NOP
  + ; reset timer on action end
  + BNE T0, R0, noresetcycle
  + NOP
  + SW R0, 0x154(V1)
* noresetcycle:
  + LW T0, 0x154(A0)
  + LI T1, 52 ; bowser drops on ground
  + BNE T0, T1, noswitchblocks
  + NOP
  + ;switch switchblocks
  + LUI T0, 0x8040
  + LB T1, 0x0(T0)
  + LI T2, 2
  + SUB T3, T2, T1
  + SB T3, 0x0(T0)
* noswitchblocks:
  + ; jump up straight
  + LI T0, 0xD
  + SW T0, 0x14C(A0)
  + LW A1, 0x80361158
  + ; look at mario
  + JAL 0x8029E694
  + NOP
  + LW A0, 0x100(V1)
  + SH V0, 0xCA(A0)
* end:
  + LW RA, 0x14(SP)
  + JR RA
  + ADDIU SP, SP, 0x18
* .endarea

First thing to note is that aglab slightly modified bowsers asm with some *NOP*s. Those are to remove effects like the stun on mario when bowser lands and other small things. You can find those on his github where I got this code from. They won't be too important when looking at this code though.

For his object he has just two functions: one that executes once on frame one, and the other that executes every frame. The frame one function gets a pointer to the bowser object and stores it in 0x100 of this object. One thing to notice is that aglab never checked for a null pointer so if you managed to unload bowser the game would definitely crash with this object here.

For the function that runs every frame he first checks the timer. He does nothing for the first 9 frames and on frame 10 he starts checking bowsers actions. If you look at my reference you see that action 0x5 and 0x6 are for the initial cutscene. If those are active he does nothing because he doesn't want to mess with the cutscene.

If the cutscene is over, he checks if the action is zero. If it is then he resets the timer goes into the main loop, if not he just goes into directly the main loop with the timer intact. In the main, he checks if the timer is 52. If it is, he changes a value in RAM. What you don't see is how that value in RAM interacts with his objects he labels “switchblocks” but you can imagine their function with the associated code. Afterwards the code does the same routine it would when timer is not 52.

When the timer is not 52, be stores 0xD in bowser's action which makes him jump straight up in the air. Then he uses a function that rotates the current object towards mario and stores the returned rotation value in bowser's rotation. Notice that A0 is bowser and A1 is mario for this function, the two arguments usually required for this type of calculation. What will happen after bowser lands is that the cycle will reset and bowser will continually jump in the air. The code will reset itself because bowsers action will change inside his own function and the timer will go back to zero automatically.

This code itself is not that confusing, but there are many hidden elements that can make it more confusing. For one, the switchblocks that we know nothing about. All we know is that a value in RAM is changed. Second is bowsers actions. We never reset the action so one would assume it stays as 0xD but because bowser has his own asm running it will change in the time between his asm and this object’s asm.

Bowser in this example is essentially a black box that we feed inputs too. We do not know what happens inside his code and do not need to. It is just a matter of experimentation to get the desired results. If you truly had an issue that would not resolve no matter how you tinkered with his actions you would have to debug bowsers code or wait for someone to hand you the code written in lovely C. This happens very often with much less complex things in mario 64 so you should be glad that such a complex object is so nice to work with.

Ex. 6 Beep Block

I wrote this code for SR 3.5 using caje asm so the syntax is a little different but the principle is the same. I also made this sort of inefficiently because at the time, there was no resource explaining geo layout switches so I switched using two models. I later have found you can use 0xF0 with a specific geo layout to switch function to change models much easier and have updated the wiki to display this. This code is inefficient but works so I will go through it anyway.

* .org 0x1203700 //80403700
* hex{00 09 00 00
* 11 01 00 01
* 2a 00 00 00 07 00 F4 F8
* 08 00 00 00
* 0c 00 00 00 80 40 37 20
* 09 00 00 00}
* //colpointer:120370c
* addiu sp, sp, $ffe8
* sw ra, $0014 (SP)
* lui t0, $8036
* lw t0, $1160 (T0)
* lw t1, $0154 (T0)
* !startcol:
* ori t2, r0, $0015 //this is the tempo
* div t1, t2
* mfhi t3 //timer mod 15
* bne t3, r0, !nosound
* nop
* lw t4, $01b0 (T0)
* addiu t4, t4, $0001
* sw t4, $01b0 (T0)
* subiu t5, t4, $0004
* blez t5, !nosound
* subiu t6, t4, $0008
* bne t6, r0, !tick
* lui a0, $301f //pound
* //blockswitchcode
* sw r0, $01b0 (T0)
* lw t9, $0188 (T0)
* beq t9, r0, !on
* nop
* //turnoff
* sw r0, $0188 (T0)
* jal $802a04c0 //change model ID
* ori a0, r0, $0014 //model ID
* beq r0, r0, !gunshot
* nop
* !on:
* ori t8, r0, $0001
* sw t8, $0188 (T0)
* jal $802a04c0 //change model ID
* ori a0, r0, $000f //model ID
* !gunshot:
* lui a0, $304d //gunshot
* !tick:
* jal $802CA144 //play sound
* ori a0, a0, $0081
* lui t0, $8036
* lw t0, $1160 (T0)
* lw t1, $0188 (T0)
* bne t1, r0, !hollowflash
* ori a0, r0, $0014 //hollow model
* ori a0, r0, $0000 //solid model ID
* !hollowflash:
* jal $802a04c0 //change model ID
* nop
* lui t0, $8036
* lw t0, $1160 (T0)
* lw t1, $0188 (T0)
* bne t1, r0, !solid
* nop
* beq r0, r0, !end
* !nosound:
* lui t0, $8036
* lw t0, $1160 (T0)
* lw t1, $0188 (T0)
* beq t1, r0, !solidflash
* ori a0, r0, $0014 //hollow model
* ori a0, r0, $000f //solid model ID
* !solidflash:
* jal $802a04c0 //change model ID
* nop
* lui t0, $8036
* lw t0, $1160 (T0)
* lw t1, $0188 (T0)
* beq t1, r0, !end
* nop
* !solid:
* jal $803839cc
* nop
* !end:
* lw ra, $0014 (SP)
* jr ra
* addiu sp, sp, $0018

The behavior is nothing special, just note that I don’t call the process solidity function in it.

First I load the timer and then get the %0x15 value. Modulo (% operator) just means the remainder and can be gotten using *MFHI* after a division. The quotient will be in the *LO* register.

If the time is not a multiple of 0x15, then the object is not on a beat. What happens is that I go near the end on that condition, check the Bparam and change the model to correspond to the Bparam. The labeled function just takes A0 as the model ID. Then I check the Bparam again and if it is zero, then I skip the process collision function effectively making the box non solid. So this means if the Bparam is zero, I use a hollow model and the object is not solid; if the Bparam is one then my object is solid and I use a solid model.

Now in the case our timer falls on a beat, we do something completely different. First we increment 0x1B0. Normally this value does nothing, I use it to keep track of which beat we are on. You can see for the first 4 beats I make no sound, then on the 8th beat I do something. So first measure no sound, second measure I play sounds and the last beat I switch. To play no sound we just use the same routine we use for offbeat frames. To play a sound we use a new routine.

I simply check if we are on beat 8, and if not I load the ID of the sound I want to play, and use the play sound effect function. Play sounds arguments always go: upper bytes == ID, lower bytes == 0x81. On top of playing a sound I also switch the model ID. I do this the same way I do the no sound routine. I check the Bparam and set the model ID argument to the corresponding one. It's important that we use 0 == solid and 1 == hollow so our model flashes on beats. Then I check the Bparam again and set solidity to match.

Finally we can go into our beat 8 code. This is labeled with the comment “//blockswitchcode”. The first thing to do is to reset our beat count to zero. Next we check our Bparam to see what we need to switch to. If its zero, we need to become solid; if it's one we need to become hollow. I do this by changing the model ID and storing the opposite value in 0x188. I could have used *XOR* here to be really efficient but I was calling a function with unique model IDs so it did not matter that much. Finally we end this switch section by loading our unique sound ID and merging back with our normal beat code. This means it does the usual solidity and checks.

While it sounds simple in words, typing it out as code is difficult, so do not worry if you understand the explanation but cannot produce code like this yourself. The key will be practice and to go through sample codes like these and trying to understand them. There is actually a lot to be optimized here in code so I’d rather you don't make something like this but something much better, I will leave that as an exercise to the reader. I am making a more optimized version myself so you can compare yours to mine and see how well you did once I release it.