?.? Analys av koden

I följande kapitel kommer väsentliga delar av koden att granskas och förklaras. För att kunna förstå vissa delar av följande kapitel måste ett fåtal saker förklaras:

* Displayen består av 8x16 lysdioder. När väggar och tak förekommer syftar det till gränserna på displayen, det vill säga är något utanför gränsen ryms det inte på displayen.
* Bilderna som representerar en spelplan illustrerar endast undre halvan av spelplanen (en av de båda 8x8 display).

?.1 Översikt av kodstruktur (processor 1)

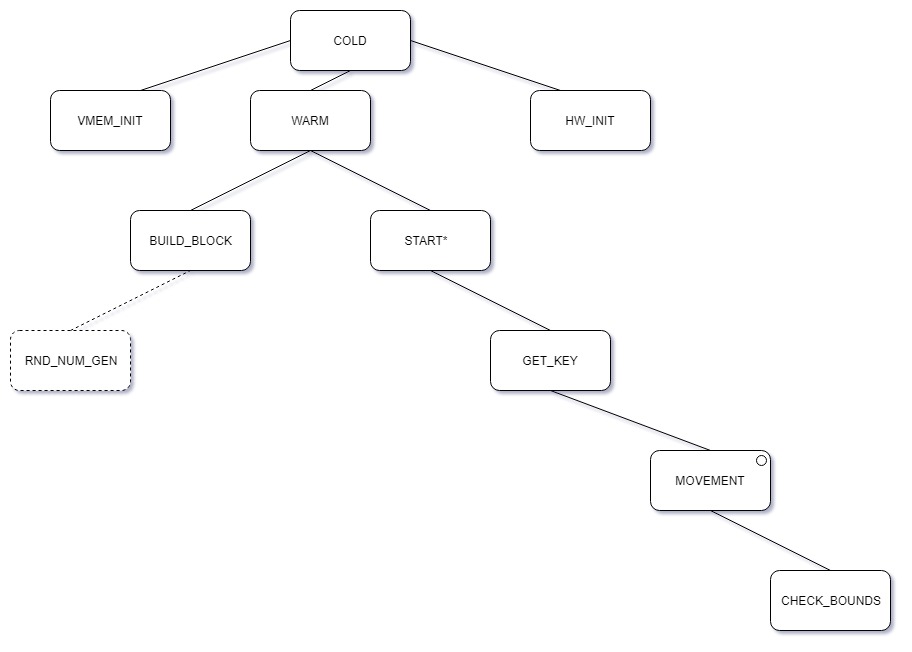
I bilden nedan, se bild 1, ses en representation över processerna som processor 1 hanterar, exklusive avbrottsrutiner vilka beskrivs nedan i bild 2. I startprocessen COLD hanteras vissa initieringar som att sätta stacken, därefter kallas VMEM\_INIT vilket initierar videominnet. Denna ordning är viktig för att videominnet inte ska bråka med gravitationen samtidigt som rensning sker. HW\_INIT initierar hårdvaran och initierar därmed avbrott via intern klocka och SPI:n. Från START kallas kontinuerligt subrutinen GET\_KEY som inväntar ett knapptryck för hantering av figurernas flyttrutiner.

Bild 1: JSP-diagram över processor 1 (exklusive avbrottsrutiner)

Vid ett specifikt knapptryck kallar rutinen GET\_KEY på ett specifikt MOVEMENT som i sin tur tillkallar CHECK\_BOUNDS för kollisionshantering innan MOVEMENT får lov att utföra sin flytt. Alltså granskas kollisionerna innan förflyttning har skett vilket underlättar vid minneshanteringen.

Avbrotten kommer kontinuerligt där GRAVITY avbrottet sköter hur figuren förflyttas nedåt medan MUX avbrottet sköter displayen. GRAVITY, se bild 2, består av ett flertal subrutiner och funktioner som måste tillkallas i ordning innan avbrottet är över.

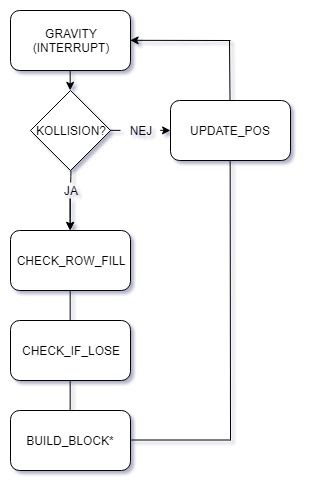


Bild 2: Diagram över avbrottsrutinen GRAVITY

Avbrottet sker i 5 steg:

1. Undersök om kollision. Uppfylls kollisionsvillkoret avbryts avbrottet.
2. Undersök om en rad är fylld. Är en rad fylld startas rutinen för radborttagning.
3. Undersök om förlust. Når figuren ”taket”? Om ja har förlust skett och spelet spelar upp ett förlustljud och stannar klockan för GRAVITY (spelet stannar och visar hur du förlorade).
4. Bygg ett nytt block om kollision har skett.
5. Uppdatera positionerna. Eftersom figuren faller ned genom att ändra dess Y-koordinater kommer figurens förra plats ligga kvar i minnet. På grund av detta måste den gamla platsen rensas ur minnet och den nya istället sparas. Detta sker oavsett om det uppstått kollision eller ej.

Det andra avbrottet som sker klockvis är MUX. Detta avbrott sköter utskrift till led-matrisen genom att skifta ut information via SPI. Sättet som muxningen sker på är inte koordinatvis utan istället radvis. Alltså sköts utskriften genom att skicka ut en rad per display för varje MUX avbrott. Informationen för utskrift sköts genom att:

1. Hämta MUXCOUNTER för att se vilken rad som är aktuell för utskrift.
2. Hämta information ur VMEM, som innehåller 16 rader med information, genom att hämta raden MUXCOUNTER och MUXCOUNTER + 7. Genom detta får båda displayerna information som kan skrivas ut samtidigt.
3. Skifta ut information till displayerna.
4. Öka MUXCOUNTER om MUXCOUNTER ≠ 8 annars sätt MUXCOUNTER = 0

Muxningsmetoden som används är väldigt snabb och resulterar därför i att muxningen inte stör andra rutiner som körs när avbrottet sker. Ljusstyrkan på dioderna blir också starkare av detta samt att spelhastigheten kan manipuleras utan hänsyn till muxningshastigheten.

?.2 Figurernas logik

Tetris består av figurer av 4 punkter vilket innebär att det på något sätt måste sparas koordinater för dessa prickar. Det sätt som valdes är att spara det som ett koordinatplan bestående av 3 rader med 8 kolumner. Detta plan definierar hur figuren kommer att se ut och går att direkt implementera i videominnet (VMEM). Därför sparas koordinaterna på ett mer sparsamt sätt med förlusten av individuell kontroll av punkter. Koordinaterna för y sparas simplare och mer hanterbart, nämligen genom att spara ett heltal som representerar ett radnummer (1–16). Koordinaterna för x sparas istället i 1 frigiven byte per x-koordinat, där det sparas 8 ”punkter” med information (se bild 1).

Bild 3: Minnet för en x-byte (sparar koordinaterna för en rad)

Denna minnesrad beskriver de punkter, i en av figurens rader, som är aktiva respektive inaktiva (nollor aktiva, ettor inaktiva). En figur byggs sedan upp av 3 stycken bytes där varje x beskriver en rad i figuren likt nedan:

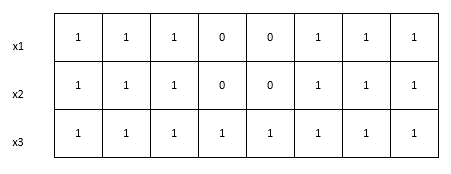


Bild 4: Tre stycken x-minnen som bygger upp ett koordinatplan och definierar en fyrkant.

Koordinatplanet ovan (bild 2) skulle alltså då rita ut en fyrkant baserat på var x-koordinaternas respektive y-värde är. Denna metod innebär att muxningen kan ske genom att muxa en hel rad åt gången vilket ökar muxningens hastighet avsevärt.

?.3 Rotationslogik

För att kunna rotera figurerna måste två olika problem överkommas: ska rotationerna vara hårdkodade eller generella? Hur ska rotationer hanteras när kollision sker efter rotation? Eftersom koordinaterna sparas som rader är det komplicerat att manipulera enskilda koordinater för att kunna skapa en generell algoritm för rotation. Detta medför att rotationerna sparas hårdkodade i minnet. Kollisionshanteringen genomförs i tre steg: Är det något direkt under figuren? Är figuren vid en vägg? Är figuren instängd av upptagna punkter i spelplanen?

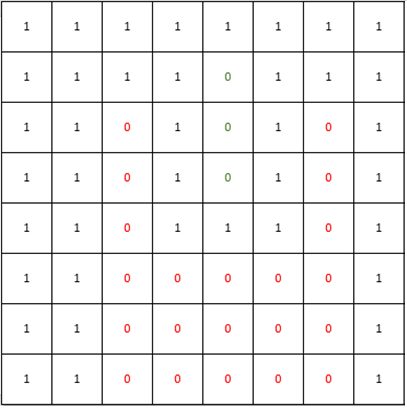
Första villkoret för rotation är väldigt simpelt, kolla om det är upptaget under figuren. Detta görs eftersom rotationspunkten resulterar i att den blir minst 1 bredare i x-led eller minst 1 högre i y-led vid rotation. Kollisionschecken undersöker alla punkter i figuren för att se om det är någon plats undertill som är upptagen.

Bild 5: Illustrerar hur kollision ses. Röda är upptagna, gröna är aktiva figuren

Ovan illustreras en halv spelplan (8x8) där grön representerar figuren som för närvarande spelas och röd representerar färdigspelade figurer. Rotationen sker alltid runt koordinaten i mitten eller x2 i vårt koordinatminne. Först görs kollisionscheck för alla punkter i figuren och om någon ger ett sant retur-värde, att figuren då befinner sig direkt ovanför en upptagen plats, avbryts rotationsprocessen och ingenting händer. Denna lösning för detta problem är väldigt primitiv men resultatet blir ett mer pålitligt rotationssystem.

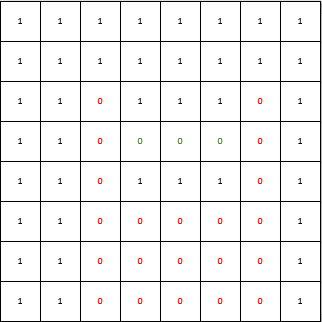
Liknande metod används vid rotation av en instängd figur likt nedan, se bild #, men istället görs först en kontroll för att se vilken rotation som är aktuell sedan om det är så att figuren kommer att ”lägga sig ned” kontrolleras om det är fritt 1 steg till höger respektive vänster innan rotationen genomförs. Om kontrollen returnerar ett false kommer rotationen avbrytas och inget kommer att hända. I bild 3 kommer rotation att genomföras eftersom längden kommer omvandlas till bredden likt nedan (bild 4)

Bild 6: Resultat efter att rotation har kallats i bild 3

Samma teknik används vid väggarna för att se om rotationen kommer att göra att vi hamnar utanför. Roteras figuren vid väggen kommer figuren, om det krävs, att flytta ett steg inåt (bort från väggen). För att detta ska fungera utan problem undersöks det om det finns plats att flytta in figuren efter rotation. Om plats inte finns kommer rotationen avbrytas och ingenting händer.

?.4 Radhantering för full rad

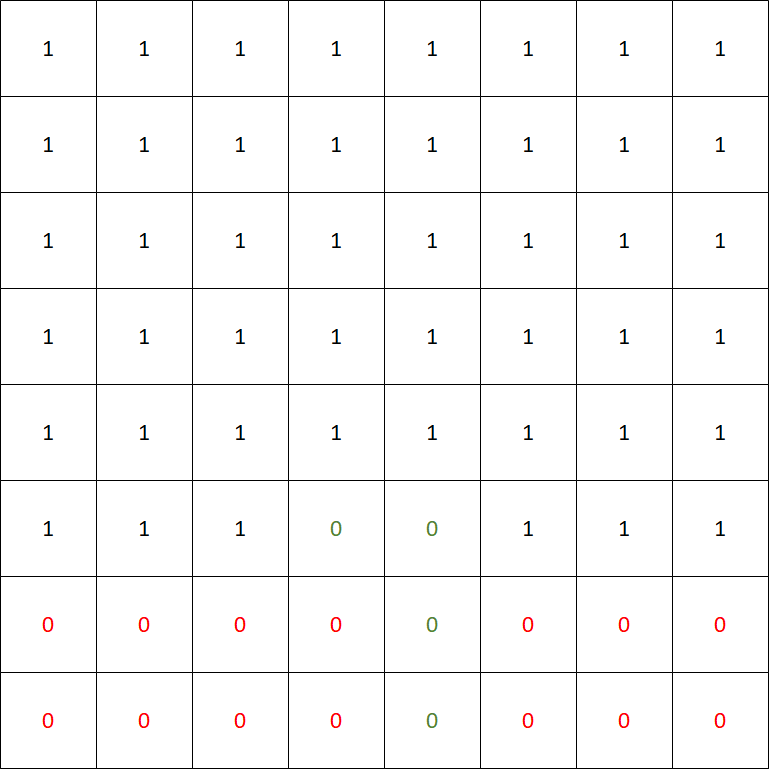
När ett block har landat kallas en subrutin som kontrollerar om en rad har blivit fylld. Om en rad har blivit fylld resulterar det i att den eller de rader som är fyllda kommer att försvinna ur videominnet VMEM. Enligt spelreglerna för Tetris ska detta resultera i poängökning, men det sker inte i denna versionen av Tetris på grund av varierande anledningar där tidsbrist är den främsta.

Bild 7: Figur (grön) landar och fyller upp 2 rader

Ovan illustreras hur en L – figur precis har landat. Vid nästa avbrott som hanterar GRAVITY kommer en kollisionscheck genomföras och i detta fall köra rutinen för radborttagning och skicka en signal till processor 2 som spelar upp ett ljud. Denna algoritm kör rensning från översta raden nedåt. Eftersom alla rader ovanför den borttagna måste tas bort kommer koden att loopa för så många rader som ska undersökas, se bild 8 nedan.

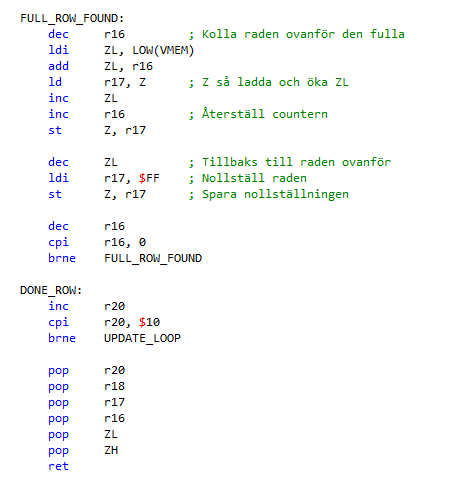


Bild 8: Kodsnutt som hanterar radflytt och borttagning

Rutinen sköter radhanteringen genom att ladda in raden ovanför den som kommer att försvinna följt av att spara över raden under med värdet från raden ovan. På detta sätt sparas tid då raderna som är fyllda inte behöver nollställas utan istället blir direkt överskrivna. Efter detta backas Z – pekaren tillbaka till den rad som nyligen ”flyttats” ned och nollställer därefter den nyligen flyttade radens gamla position. När detta är gjort minskas registret r16, som i detta fall räknas som en loopcounter, för att sedan flytta ned de rader som är kvar att flytta på. När detta är klart ramlar koden ner i DONE\_ROW som ser till att genomföra operationen för alla rader på spelplanen. Detta sker eftersom det kan vara fler än endast 1 rad som blivit fylld. Efter denna rutin kört klart kommer spelplanen att se ut likt följande:

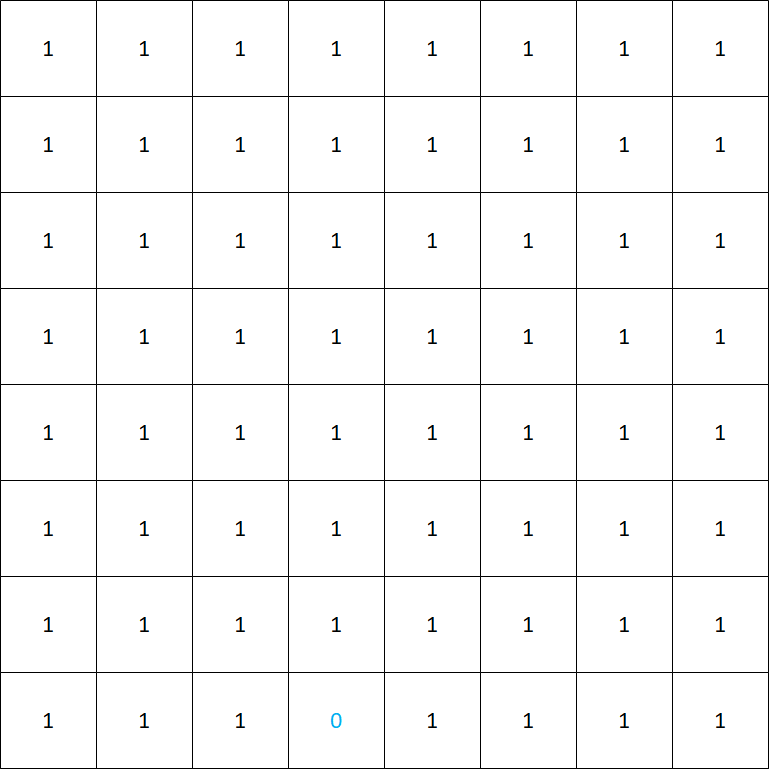
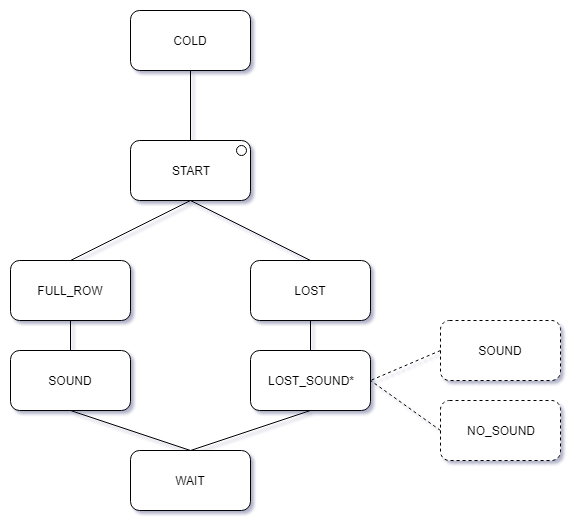


Bild 9: Resultat av bild 7 efter radborttagning. Blå representerar biten som är kvar av den spelade figuren efter rensning.

Eftersom två av raderna i bild 7 var fyllda försvinner dessa efter rutinen i bild 8 och kvar blir då den, i bild 8, blå biten som landade ovanpå raderna. När GRAVITY avbrottet är genomfört kommer sedan rutinen BUILD\_BLOCK köras så att spelet fortsätter.

?.1 Översikt av kodstruktur (processor 2)

Processor 2 sköter endast ljudhantering genom att få in signaler på två av sina ben. Dessa två insignaler bestämmer huruvida det ska spelas ett ljud för radborttagning eller om det ska spelas ett ljud för förlust. Processerna som används är väldigt simpla och använder sig endast av olika delayer för att sätta hög respektive låg signal på den pinne som är kopplad till högtalaren. Bild är ett JSP – diagram som tydliggör hur processorn hanterar sina requests. Cold initierar stacken och in- respektive utgångar. Rutinen START loopas och inväntar en signal från en av två pinnar. När signal uppfångats verifierar rutinen vilken pinne som skickade signal och väljer nästa subrutin beroende på vad som ska spelas. Valen som görs är att spela ett ljud för förlust eller att spela ett ljud när en full rad är funnen.