Department of Information Technology

Digitala projekt

SuperKull™

Daniel Öhman <d03do@efd.lth.se>
Alexander Persson <d03ap@efd.lth.se>

Abstract

The purpose of this course was to design and construct an electronic circuit with a micro processor. We decided to construct a four player game, controlled by joysticks and displayed on a LCD screen. The work method used during the project was the waterfall model. The four phases of development were; Requirements Specification, Design, Construction and Implementation. We succeded in building a working prototype for our game.

Innehållsförteckning

1. Inledning

I moderna företag har projektarbeten fått en allt större betydelse, något som denna kurs försökt förbereda oss för. För att nå dit har vi självständigt, under viss handledning, fått välja och realisera en valfri konstruktion. Vi hadevid projektets början stora kunskaper inom programmering men bristfälliga inom elektronik. Därför valde vi en enklare processor för vår prototyp, ATmega16, men ganska höga krav på programmeringen. Programmet som körs på kontruktionen är ett spel för fyra spelare.

2. Metod och resultat

2.1 Spelet

Spelet heter SuperKull™ och bygger på en variant av känd lek, nämligen kull [6]. De fyra spelpjäserna börjar i varsitt hörn på spelplanen och kontrolleras av spelarna via joysticks. En av spelarna börjar att vara *han*, när denne spelare sedan kolliderar med en annan spelare blir den spelaren *han*. Den som är *han* efter en viss speltid har förlorat och får bestraffas av dem andra spelarna.

2.2 Hårdvaran

Först bestämde vi oss för att använda en enchipsdator, ATmega16 [1]. Denna fick vi färdigmonterad med ett JTAG-interface. För styrning valde vi fyra stycken joysticks [2], dessa uppfyllde våra krav och var de billigaste som fanns att tillgå. För att visa spelet användes en Batron 128x64 LCD-skärm [3]. Övrig hårdvara var en kondensator för att minska ingångsbrus samt virtråd.

I bilaga A finns ett kopplingsschema för konstruktionen. Vi kom fram till den designen genom att läsa de olika komponenternas datablad [1] [2] [3]. Ledare från spänningskällan är fastlödda men de andra ledarna är virade.

Under byggandet och testningen av konstruktionen råkade vi bränna en processor, detta p.g.a. ett icke jordat ben på processorn.

2.3 Mjukvaran

Utvecklingsmiljön vi använde var AVR Studio 4, detta program möjliggjorde kontakt med hårdvaran via JTAG. Programmet skrevs i språket C. Genom AVR Studio fick vi även tillgång till headerfiler tillhörande AVR ATMega16, dessa underlättade vårt arbete avsevärt. Filen *io.h* innehåller definitioner för alla register på processorn och *pgmspace.h* innehåller definitioner och funktioner för att spara och läsa från programminnet.

I bilaga B beskrivs programmets flödesschema.

Programmet består av fyra delar:

SuperKull.c
Denna fil innehåller main-funktionen, här startas skärmen samt spel-loopen.

Batron.c
Här hanteras allt som har med skärmen att göra, starta, rensa och rita. Först skickas en instruktion som väljer var man vill rita, sen skickas det som ska visas. Två specialiserade funktioner visar splashscreens, dels vid start av spelet under speletes gång, i form av en klocka och när spelet är slut en Game Over skärm.

Control.c
Startar och läser av D/A-omvandlaren samt byter kanal. En funktion som heter readResult() returnerar resultatet från senaste omvandlingen och startar nästa omvandling. Vi valde att jobba med 4 bitars upplösning per axel för våra kontroller.

Resultatet vi får används sedan för att beräkna den nya farten för den axeln enligt följande formel:

ny hastighet = gammal hastighet + (avläst resultat – 128) / 20 – konstant friktion

Friktionen används mest för att spelare inte ska studsa omkring i all oändlighet samt för att motverka bruspåverkan på D/A-omvandlaren. För att få en mer jämn friktion skulle en korrekt fysikalisk modell behöva användas istälelt för en konstant friktion.

Game.c
Här är huvudloopen i programmet. Spelarnas positioner och hastigheter uppdateras och kollisioner detekteras. Spelarnas avstånd beräknas med hjälp av Pythagoras sats och jämförs med spelarnas radie. När två spelare kolliderar så måste nya hastigheter beräknas, detta valde vi att hantera som två huvudfall, då spelarna åker i motsatt riktning och då de åker i samma riktning. När spelarna åker i motsatt riktning blir den nya hastigheten efter kollision:

ny egen hastighet = egen hastighet * -1 + motståndarens hastighet
motståndarens nya hastighet = motståndarens hastighet * -1 + egen gammal hastighet

När spelarna färdas i samma riktning blir nya hastigheten:

om spelare 1:s hastighet > spelare 2:s hastighet
spelare 1.s nya hastighet = spelare 1:s hastighet * -1 + spelare 2:s hastighet
spelare 2:s nya hastighet = spelare 1:s hastighet

Vi testade först att använda en mer fysikaliskt korrekt modell för kollisioner men då är man tvungen att ta hänsyn till elasticitet, eftersom de ofta har samma fart, vilket leder till att de skulle stanna vid kollisioner. Denna enklare modell ger ändå en bra spelkänsla.

Istället för att spara unden hela bildminnet inför varje skärmuppdatering sparade vi endast de byte på skärmen som faktiskt användes. När fler än en spelarsprite delar på samma bytes i skärmens minne, måste man se till att den som ritas ut först inte skrivs över av den som ritas ut sist. Detta löste vi genom att lägga in de använda byten i en

lista som sedan sorterades med hjälp av quicksort [4]. De byte som hade samma minnesadress använde vi logiskt OR [5] på, när de skickades till skärmen.

I denna fil finns även alla de sprites som används i spelet deklarerade. De sparas i programminnet då processorn har begränsat med RAM. För att sedan läsa från programminnet användes funktionen *pgm_read_byte_near( ).*

Under vår testperiod upptäckte vi att programmet hängde sig eller startade om, tillsynes slumpmässigt. Ibland direkt, ibland efter flera minuters speltid. Problemet visade sig vara vår sorteringsalgoritm, quicksort[4]. Vi tror att de många kontextbyten som den rekursiva algoritmen medförde gjorde att stacken inte räckte till. Vi löste detta genom att byta till en mindre effektiv linjär sortering och programmet uppvisade efter det inga problem alls.

3. Diskussion och slutsatser

Under kursens gång har vi lärt oss att datablad är en viktig, om än svåråtkomlig, informationskälla vid både konstruktion och implementation. Som de D-teknologer vi är var våra tidigare erfarenheter av lödning begränsade, något som vi nu fått lära oss.

Det största problemet under utveckligen av mjukvaran var det begränsade minnesutrymmet på processorn. Detta gjorde att vi fick optimera programmet ganska hårt med hänsyn till minnesanvändning.

Vi är nöjda med prototypen men vissa förbättringar skulle kunna göras, t.ex. ljudeffekter samt menysystem. Om vi skulle vetat vad vi vet idag, skulle vi nog ha valt en processor med mer minne.

På det hela taget är vi nöjda med kursen och kan rekommendera den till alla med ett intresse för inbyggda system.

Referenser

[1] AVR ATmega16 - http://www.it.lth.se/datablad/Processors/ATmega16_sum.pdf

[2] Joystick - http://www.elfa.se/pdf/64/06413538.pdf

[3] Batron LCD-skärm - http://www.it.lth.se/datablad/display/Batron128x64.pdf

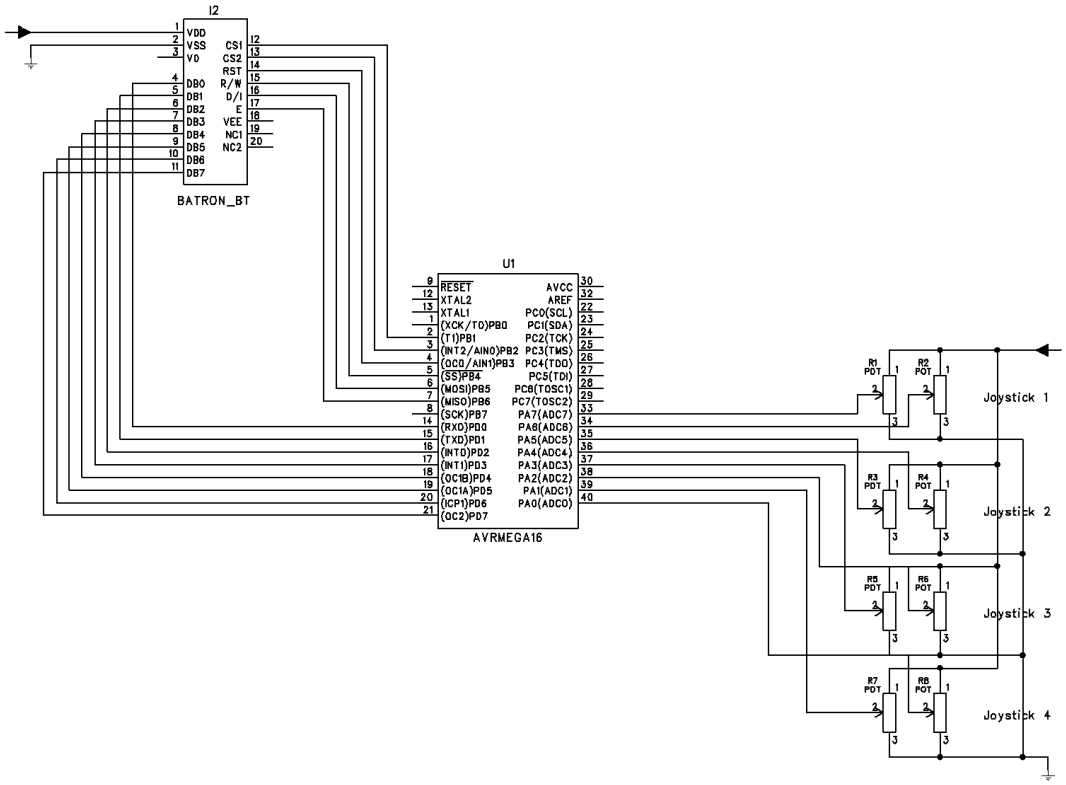
[4] Quicksort - http://en.wikipedia.org/wiki/Quicksort

[5] Logiskt OR - http://en.wikipedia.org/wiki/Logical_or

[6] Kull - http://susning.nu/Kull

Bilaga A

Kopplingsschema



Bilaga B

Flödesschema