Università degli studi di Palermo

Scuola Politecnica Ingegneria Informatica LM-32

Sistemi embedded

SNAKE

Membri del gruppo:

Sciarrabba Enrico Maria

Solaro Giuliano

Sommario

1. STRUTTURA GENERALE 1

2. INSTALLAZIONE 3

2.1 HARDWARE 3
2.2 SOFTWARE 5

3. FILE GENERICI 6

3.1 PRINT_FUNCTIONS.F 6
3.2 TIMER.F 12
3.3 RECEIVER.F 16

4. FILE SPECIFICI 21

4.1 SNAKE.F 21
4.2 INTERFACES.F 31
4.3 MAIN.F 37

5. FILE ASSEMBY 41

5.1 PRINT_PIXEL.S 41
5.2 PRINT_CHAR.S 42

6. ADATTAMENTO 45

6.1 TELECOMANDO 45

1. Struttura generale

PREMESSA
Tutte le operazioni mostrate in questo capitolo e in quello successivo sono state eseguite su un sistema Unix.

Nella cartella del Progetto si trovano due sotto cartelle:

- Sistema operativo
- Snake

Nella prima sono contenuti i file:

- R3bp (Cartella)
- R4bp (Cartella)
- Bootcode.bin
- Config.txt
- Fixup.dat
- Start.elf

necessari per il corretto avvio del raspberry e per il caricamento del sistema operativo pijFrthos attraverso il quale è stato sviluppato il gioco.

Nella seconda cartella troviamo:

- Assembly
- Forth
- Crea_file.sh

Crea_file.sh è stato inserito per semplificare il caricamento del codice sulla macchina. Avendo infatti strutturato il codice in più file, questo permette di unificarli.

Assembly:

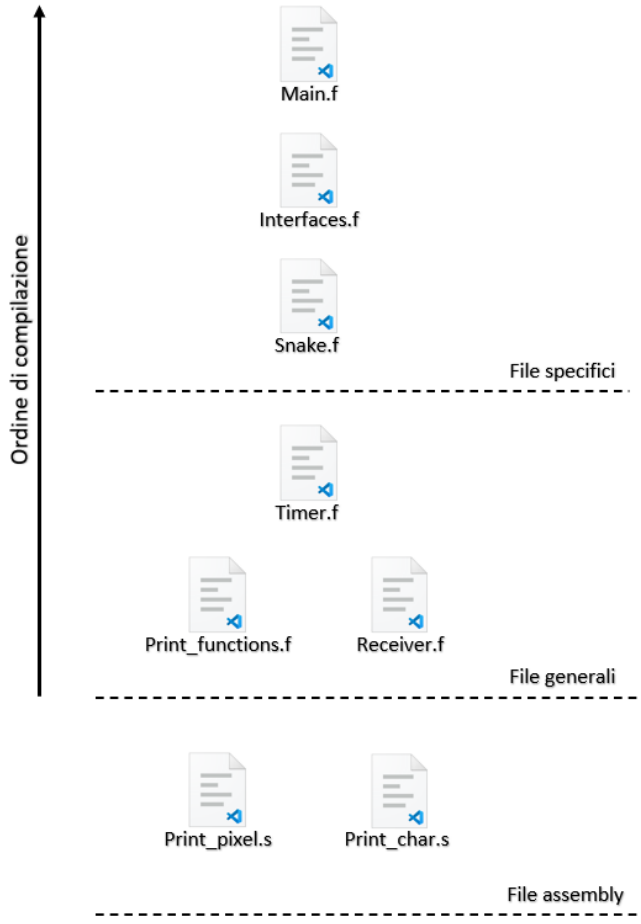
- Print_char.s
- Print_pixel.s

Forth:

- Interfaces.f
- Main.f
- Print_functions.f
- Receiver.f
- Snake.f
- Timer.f

Possiamo suddividere tutti questi file in 3 gruppi:

- **File generici**: sono da vedere come delle librerie che mettono a disposizione metodi di utilità. Questi file potrebbero quindi essere riutilizzati per altri scopi poiché risultano indipendenti da tutti gli altri codici del progetto corrente.
- **File specifici**: contengono le istruzioni specifiche del gioco.
- **File assembly**: contengono istruzioni assembly per l'ottimizzazione della stampa.



La foto sopra mostra la suddivisione dei file e l'ordine secondo il quale questi devono essere compilati.

2. Installazione

Per il corretto funzionamento del progetto saranno necessari i seguenti componenti hardware e software:

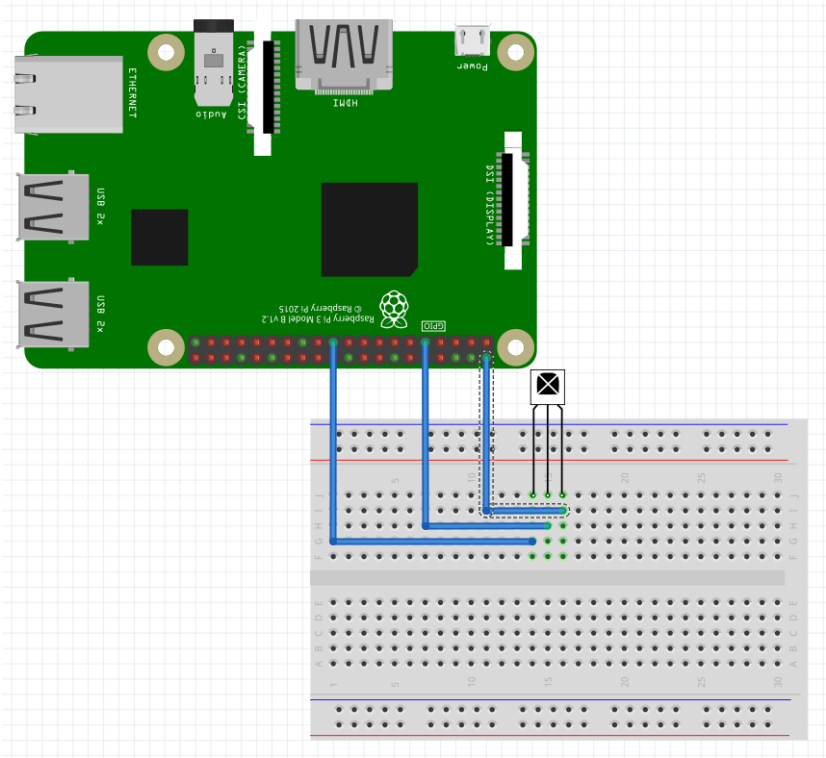
2.1 Hardware

Componenti:

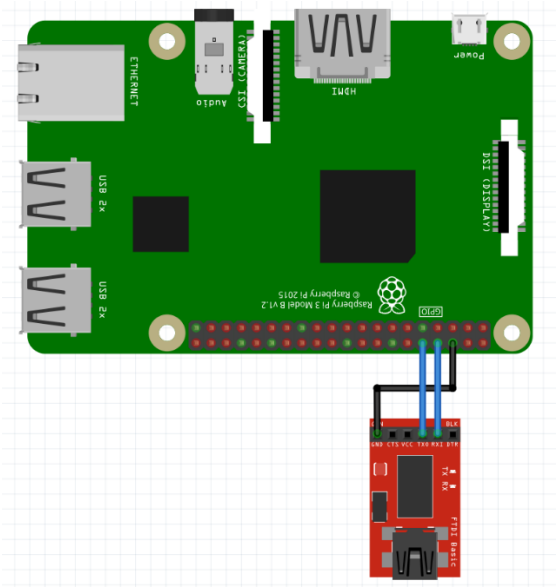
Raspberry Pi 3B+ o 4B+	
FT232RL FTDI USB To TTL Serial Converter	
Ricevitore infrarossi VS1838B + telecomando infrarossi	
Micro SD	
Jumper	

Dopo aver inserito i file necessari nella micro SD (vedere paragrafo successivo) inserirla nello slot dedicato del raspberry.

Tramite i jumper collegare il ricevitore infrarossi come indicato nell’immagine sottostante



Successivamente collegare il modulo FT232RL come indicato nell'immagine sottostante



Per completare l'installazione collegare il cavo dal modulo FT232RL al computer per la comunicazione seriale, il cavo di alimentazione e il cavo HDMI.

2.2 Software

Caricare nella microSD del raspberry il sistema operativo pijForthos. Troviamo i file nel percorso Progetto > Sistema operativo.
Scegliere il file Kernel7.img in base alla macchina utilizzata.

È possibile unificare i file della cartella Forth eseguendo il file Crea_file.sh attraverso il comando:

```
./Crea_file.sh
```

Il comando crea un file Snake.f contenente le istruzioni forth.

Altri due software fondamentali sono picocom e minicom che possono essere scaricati eseguendo le seguenti istruzioni:

```
sudo apt install picocom
sudo apt install minicom
```

Posizionarsi nella cartella Snake (Progetto > Snake), eseguire picocom con la seguente istruzione:

```
picocom --b 9600 /dev/tty0 --send "ascii-fr -sn -l100 -c10" --imap delbs
```

Sostituire il percorso “/dev/tty0” con la porta seriale con la quale è collegato il raspberry.

Infine collegare il cavo di alimentazione al raspberry.

Avviato il sistema, premendo la combinazione di tasti *ctrl + a + s* è possibile caricare sul raspberry le istruzioni contenute in un file. Digitare il comando “./Snake.f”

Attendere il completamento dell’operazione.

Per avviare il gioco utilizzare il comando GAME.

3. File generici

Il progetto, nella directory “Progetto > Forth”, include tre file “generici”:

- print_functions.f
- timer.f
- receiver.f

Sono così definiti in quanto potrebbero essere visti come delle librerie che mettono a disposizione dei metodi da chiamare dall’esterno.

Per garantire una certa autonomia questi possiedono delle specifiche (variabili e metodi) particolari che devono essere adattate al proprio ambiente.

Analizziamo questi tre file:

3.1 Print_functions.f

Tale file contiene i metodi che si occupano della stampa di numeri, lettere o simboli.

Possiamo vedere lo schermo come una grande griglia dove ogni cella (pixel) è composta da tre sub-pixel di colori differenti: rosso, verde e blu. Ad ognuno di questi possiamo attribuire un’intensità diversa che va da 0 a 255.
In realtà ad ogni pixel attribuiamo anche un quarto valore, la trasparenza, non utilizzato ai fini del progetto e, di conseguenza, impostato sempre a 0.

Per mostrare qualcosa nello schermo basta accendere i pixel col colore desiderato (regolando le tre intensità).

Dalla libreria print_functions.f è possibile richiamare i seguenti metodi:

- PRINT_PIXEL (a1 a2 --)
- PRINT_CHAR (a1 a2 --)
- PRINT_NUMBER (a1 --)

PRINT_PIXEL.f

Il primo metodo è **PRINT_PIXEL** che permette di creare un rettangolo di dimensione variabile:

```
\ Subroutine in assembly per stampare a
\ schermo un rettangolo
CREATE PRINT_PIXEL_ASSEMBLY
e92d5000 , e59f0060 , e5900000 , e59f105c    , e5911000 ,
e1a02001 , e59f3054 , e0011003 , e59f3050    , e0022003 ,
e1a02622 , e59f3048 , e5933000 , eb000001    , e8bd5000 ,
e12fff1e , e1a04002 , e1a02004 , e4803004    , e2522001 ,
1affffc , e1a02104 , e0500002 , e2800a01    , e2511001 ,
1affff6 , e12fff1e ,  PIXEL   , PIXEL_SIZE  , 00000fff ,
00fff000 , COLOR ,

\ Interfaccia Forth di PRINT_PIXEL_ASSEMBLY
: PRINT_PIXEL 1000 * + SPIXEL_SIZE PRINT_PIXEL_ASSEMBLY JSR DROP ;  \ ( a1 a2 -- )
```

Il metodo prende in input le dimensioni del rettangolo e le salva in un'unica variabile, PIXEL_SIZE, alla quale farà accesso PRINT_PIXEL_ASSEMBLY. In questo ritroviamo le istruzioni in linguaggio macchina e gli indirizzi delle variabili in cui trovare i valori da utilizzare. Quindi PRINT_PIXEL è soltanto un'interfaccia che permette di richiamare il codice assembly che stamperà il rettangolo. Per maggiori informazioni guardare i file assembly.

PRINT_PIXEL richiede l'inizializzazione di alcune variabili:

- PIXEL: memorizza l'indirizzo del pixel in alto a sinistra del rettangolo

```
\ Contiene l'indirizzo del registro del pixel di
\ partenza in una stampa
VARIABLE PIXEL
: GPIXEL PIXEL @ ;                          \ (    -- b1 )
: SPIXEL PIXEL ! ;                          \ ( a1 --    )
: INC_PIXEL_O  4    * GPIXEL + SPIXEL ;     \ ( a1 --    )
: INC_PIXEL_V  1000 * GPIXEL + SPIXEL ;     \ ( a1 --    )
```

- COLOR: memorizza il colore del rettangolo

```
\ Contiene il valore rgb del colore
\ scelto per la stampa
VARIABLE COLOR
: GCOLOR COLOR @ ;                  \ (    -- b1 )
: SCOLOR COLOR ! ;                  \ ( a1 --    )
```

Esempio:

```
HL SPIXEL WHITE SCOLOR
300 400 PRINT_PIXEL
```

Stampa, partendo dal pixel in alto a sinistra, un rettangolo bianco grande quanto tutto lo schermo.

PRINT_CHAR

Anche questo, come PRINT_PIXEL, è un'interfaccia di PRINT_CHAR_ASSAMBLY.

```
\ Subroutine in assembly per stampare a
\ schermo un carattere
CREATE PRINT_CHAR_ASSEMBLY
e92d4010 , e59f0144 , e5900000 , e59f1140 , e5911000 ,
e59f213c , e5922000 , e59f3138 , e5933000 , e59f4134 ,
e5944000 , e1833404 , eb000001 , e8bd4010 , e12fff1e ,
e92d0030 , e92d00c0 , e92d4100 , e1a04000 , e1a05001 ,
e1a06002 , e3a07000 , e3c380ff , e1a08428 , e20330ff ,
e3a02020 , e1a00006 , e3a01001 , e0000001 , e3500000 ,
0a000005 , e1a00004 , e1a01003 , e92d000c , e1a03008 ,
eb00001b , e8bd000c , e0844103 , e2422001 , e1a060a6 ,
e0820007 , e3a01005 , eb000022 , e3500002 , 1a000005 ,
e3a01014 , e0000391 , e0444000 , e3a01a01 , e0000391 ,
e0844000 , e1a00002 , e3500000 , 1affffe3 , e3570002 ,
0a000003 , e3a07002 , e3a02008 , e1a06005 , eaffffdd ,
e8bd4100 , e8bd00c0 , e8bd0030 , e12fff1e , e92d1010 ,
e1a02001 , e1a04002 , e1a02004 , e4803004 , e2522001 ,
1affffc , e1a02104 , e0500002 , e2800a01 , e2511001 ,
1affff6 , e8bd1010 , e12fff1e , e1500001 , ba000002 ,
e0400001 , e1500001 , aaffffc , e12fff1e , PIXEL    ,
CHAR1    , CHAR2    , CHAR_SIZE , COLOR   ,

\ Interfaccia Forth di PRINT_PIXEL_ASSEMBLY
: PRINT CHAR SCHAR2 SCHAR1 PRINT CHAR ASSEMBLY JSR DROP ;       \ ( a1 a2 -- )
```

Il metodo richiede in input la codifica del carattere che si vuole stampare e questa verrà inserita nelle variabili CHAR1 CHAR2 dalle quali il codice assembly andrà a leggere i valori. Per maggiori informazioni della codifica guardare PRINT_CHAR_ASSEMBLY.

PRINT_CHAR utilizza PRINT_PIXEL, per questo motivo prima di essere chiamato è necessario inizializzare le variabili PIXEL e COLOR, oltre a CHAR_SIZE:

- CHAR_SIZE: memorizza la dimensione della lettera.

```
\ Contiene la dimensione di un carattere
VARIABLE CHAR_SIZE
: GCHAR_SIZE CHAR_SIZE @ ;                    \ (     -- b1 )
: SCHAR_SIZE CHAR_SIZE ! ;                    \ ( a1 --     )
```

Esempio:

```
HL SPIXEL WHITE SCOLOR A SCHAR_SIZE
.A. PRINT_CHAR
```

Stampa, partendo dal pixel in alto a sinistra, una lettera A bianca.

PRINT_NUMBER

Stampa il numero in cima allo stack. Se il numero è espresso in una qualsiasi codifica viene prima convertito in decimale e poi stampato.

```
\ Stampa il numero in input
: PRINT_NUMBER                                          \ ( a1 -- )
    DUP SNUM_TEMP  -1 SWAP
    BEGIN
        SWAP 1 +  SWAP A /
    DUP 0 =  UNTIL
    DROP
    BEGIN
        DUP A SWAP EXP  GNUM_TEMP SWAP /
        CODE_NUMBER  PRINT_CHAR  GCHAR_SIZE 6 * INC_PIXEL_O
        SWAP DUP  A SWAP EXP  ROT * GNUM_TEMP SWAP - SNUM_TEMP  1 -
    DUP -1 =  UNTIL
    DROP
;
```

Essendo un metodo di stampa devono essere inizializzate le variabili: PIXEL, COLOR, CHAR_SIZE.
Esempio:

```
HL SPIXEL WHITE SCOLOR A SCHAR_SIZE
20 PRINT_NUMBER
```

Viene stampato il numero 20. Se, ad esempio, la codifica del numero 20 è esadecimale questo viene convertito in 32 decimale e successivamente stampato sullo schermo.

PRINT_NUMBER utilizza il metodo EXP che permette di effettuare l'elevazione a potenza.

```
\ Effettua l'elevazione a potenza
: EXP                                                    \ ( a1 a2 -- b1 )
    DUP 0 =  IF  DROP DROP 1  ELSE
    DUP 1 =  IF  DROP         ELSE
        SWAP DUP ROT
        BEGIN
            1 -  ROT ROT SWAP DUP ROT * ROT
        DUP 1 =  UNTIL
        DROP SWAP DROP
    THEN THEN
;
```

Esempio:

```
3 7 EXP
```

Lascia sullo stack il risultato di 3^7.

PRINT_WORD

Stampa la parola presente nello stack.

```
\ Stampa le lettere in input
: PRINT_WORD                                     \ ( a1 ... an an+1 -- )
    DUP 6 * GCHAR_SIZE * INC_PIXEL_O
    BEGIN
        GCHAR_SIZE -6 * INC_PIXEL_O
        ROT ROT PRINT_CHAR
        1 -
    DUP 0 =  UNTIL
    DROP
;
```

Sotto la codifica delle lettere.

```
\ ( -- b1 b2 )
: .A. 8C 7F18C62E ;
: .B. 7C 6317C62F ;
: .C. F0 4210843E ;
: .D. 7C 6318C62F ;
: .E. F8 4217843F ;
: .F. 08 4217843F ;
: .G. 74 6316843E ;
: .H. 8C 631FC631 ;
: .I. 71 0842108E ;
: .J. 32 5084211C ;
: .K. 8C 63149D31 ;
: .L. F8 42108421 ;
: .M. 8C 631AD771 ;
: .N. 8C 639ACE31 ;
: .O. 74 6318C62E ;
: .P. 08 42F8C62F ;
: .Q. 20 CA94A526 ;
: .R. 8A 4AF8C62F ;
: .S. 74 6107062E ;
: .T. 21 0842109F ;
: .U. 74 6318C631 ;
: .V. 21 14A54631 ;
: .W. 55 6B5AC631 ;
: .X. 8A 94422951 ;
: .Y. 21 08452A31 ;
: .Z. F8 4222221F ;
: .0. 74 6318C62E ;
: .1. F9 084210E4 ;
: .2. F8 4444462E ;
: .3. 74 6106422E ;
: .4. 42 3E952988 ;
: .5. 7C 2107843F ;
: .6. 74 6317862E ;
: .7. 21 0844421F ;
: .8. 74 6317462E ;
: .9. 74 610F462E ;
: .DP. 0 40008000 ;
: .SPACE. 0 0 ;
```

Print word permette di stampare una qualsiasi parola dopo aver inizializzato le variabili PIXEL, COLOR, CHAR_SIZE ed aver lasciato sullo stack le codifiche delle lettere e il numero di lettere utilizzato.

Esempio:

```
HL SPIXEL WHITE SCOLOR A SCHAR_SIZE
.H. .E. .L. .L. .O. 5 PRINT_WORD
```

Stampa, partendo dall'indirizzo indicato nella variabile PIXEL, la scritta "HELLO" di colore bianco.

3.2 Timer.f

Questo file mette a disposizione un timer che conta dai decimi di secondo fino ai minuti. Il suo limite è 99:59:9. Oltrepassandolo i numeri non verranno più stampati nell'ordine corretto.

Come prima operazione, poiché nel codice è necessario accedere al registro CLO, viene auto-configurato il suo indirizzo in memoria.

Si riconosce uno dei due modelli 3b+ e 4b+ di raspberry verificando l'indirizzo del registro CLO. Successivamente viene impostato l'indirizzo del registro di base dal quale è possibile calcolare quelli di tutti gli altri registri applicando degli offset.

```
\ Autoconfigurazione base_register
: AUTOCONFIG                                \ (  -- b1 )
    3F003004 @
    5 DROP
    3F003004 @ =  IF
        FE000000
    ELSE
        3F000000
    THEN
;
AUTOCONFIG CONSTANT BASE_REGISTER
```

Poiché il timer viene stampato a schermo è necessario impostare alcuni parametri:

- **TIMER_COLOR**: memorizza il colore che avrà il timer

```
\ Contiene il valore rgb del colore
\ scelto per il timer
VARIABLE TIMER_COLOR
: GTIMER_COLOR TIMER_COLOR @ ;                \ (    -- b1 )
: STIMER_COLOR TIMER_COLOR ! ;                \ ( a1 --    )
```

- **TIMER_PIXEL**: memorizza l'indirizzo di partenza dal quale stampare il timer

```
\ Contiene l'indirizzo del pixel di
\ partenza in cui stampare il timer
VARIABLE TIMER_PIXEL
: GTIMER_PIXEL TIMER_PIXEL @ ;                \ (    -- b1 )
: STIMER_PIXEL TIMER_PIXEL ! ;                \ ( a1 --    )
```

- **TIMER_SIZE**: memorizza la dimensione del timer

```
\ Contiene la dimensione del timer
VARIABLE TIMER_SIZE
: GTIMER_SIZE TIMER_SIZE @ ;                    \ (     -- b1 )
: STIMER_SIZE TIMER_SIZE ! ;                    \ ( a1 --     )
```

I metodi che possono essere richiamati sono SET_TIMER e CONTROL_TIMER.

SET_TIMER

Inizializza il contatore e lo stampa a schermo.

```
\ Inizializza il timer e lo stampa
: SET_TIMER                                     \ (     --     )
    SET_MINUTES
    SET_SECONDS
    SET_DECSECONDS
    GTIMER_COLOR  SCOLOR
    GTIMER_PIXEL  SPIXEL
    GTIMER_SIZE   SCHAR_SIZE
    GMINUTES     DUP PRINT_NUMBER PRINT_NUMBER  .DP. PRINT_CHAR  GCHAR_SIZE 2 * INC_PIXEL_O
    GSECONDS     DUP PRINT_NUMBER PRINT_NUMBER  .DP. PRINT_CHAR  GCHAR_SIZE 2 * INC_PIXEL_O
    GDECSECONDS      PRINT_NUMBER
;
```

Esempio:

```
HL STIME_PIXEL WHITE STIME_COLOR A STIMER_SIZE
SET_TIMER
```

Stampa, partendo dal pixel in alto a sinistra, un timer 00:00:0 di colore bianco.

CONTROL_TIMER

Il metodo controlla se sono passati più 100ms dall'ultimo aggiornamento e, in tal caso, incrementa i decimi di secondo.

```
\ Controlla se è il momento di
\ aggiornare il timer
: CONTROL_TIMER                                  \ (    --    )
    GCLOCK 186A0 /
    DUP GTIMER <>  IF
        STIMER
        GTIMER_COLOR SCOLOR
        GTIMER_SIZE SCHAR_SIZE
        UPDATE_DECSECONDS
    ELSE
        DROP
    THEN
;
```

L'incremento del timer avviene tramite una reazione a catena: vengono infatti incrementati direttamente soltanto i decimi di secondo tramite UPDATE_DECSECONDS.

```
\ Contiene i decimi di secondo
VARIABLE DECSECONDS
: GDECSECONDS DECSECONDS @ ;                     \ (    -- b1 )
: SET_DECSECONDS 0 DECSECONDS ! ;                \ (    --    )
: UPDATE_DECSECONDS                              \ (    --    )
    GDECSECONDS 1 +
    DUP A =  IF
        SET_DECSECONDS
        UPDATE_SECONDS
        DROP GDECSECONDS
    ELSE
        DUP DECSECONDS !
    THEN

    GTIMER_PIXEL SPIXEL
    1C GTIMER_SIZE * INC_PIXEL_O
    BLACK SCOLOR
    GCHAR_SIZE 8 *  GCHAR_SIZE 5 *  PRINT_PIXEL
    WHITE SCOLOR  PRINT_NUMBER
;
```

Dopo l'incremento lo stesso metodo verifica se tale valore ha raggiunto A (10 in decimale) e in tal caso incrementa i secondi richiamando UPDATE_SECONDS.

```
\ Contiene i secondi
VARIABLE SECONDS
: GSECONDS SECONDS @ ;                          \ (    -- b1 )
: SET_SECONDS 0 SECONDS ! ;                     \ (    --    )
: UPDATE_SECONDS                                \ (    --    )
    GSECONDS 1 +
    DUP 3C =  IF
        SET_SECONDS
        UPDATE_MINUTES
        DROP GSECONDS
    ELSE
        DUP SECONDS !
    THEN

    GTIMER_PIXEL SPIXEL
    E GCHAR_SIZE * INC_PIXEL_O
    BLACK SCOLOR
    GCHAR_SIZE 8 *  GCHAR_SIZE C *  PRINT_PIXEL
    WHITE SCOLOR
    DUP  A / 0 =  IF  0 PRINT_NUMBER  THEN
    PRINT_NUMBER
;
```

Analogo ragionamento viene fatto per i minuti che vengono incrementati quando i secondi diventano 60 chiamando UPDATE_MINUTES.

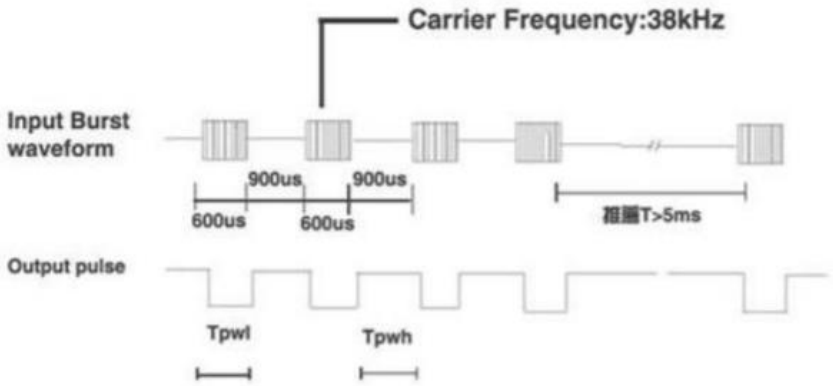
```
\ Contiene i minuti
VARIABLE MINUTES
: GMINUTES MINUTES @ ;                          \ (    -- b1 )
: SET_MINUTES 0 MINUTES ! ;                     \ (    --    )
: UPDATE_MINUTES                                \ (    --    )
    GMINUTES 1 +  DUP MINUTES !
    GTIMER_PIXEL SPIXEL
    BLACK SCOLOR
    GCHAR_SIZE 8 *  GCHAR_SIZE C *  PRINT_PIXEL
    WHITE SCOLOR
    DUP  A /  0 =  IF  0 PRINT_NUMBER  THEN
    PRINT_NUMBER
;
```

3.3 Receiver.f

Il file contiene i metodi che gestiscono i segnali provenienti dal telecomando e letti dall'output del ricevitore infrarossi.

Per iniziare è utile mostrare come avviene una comunicazione tra telecomando e ricevitore.

Quando viene premuto un pulsante sul telecomando, questo invia dei bit al ricevitore. L'invio avviene tramite la generazione di un treno di impulsi di frequenza 38kHz. Se viene trasmesso un bit 1 il telecomando genera l'onda, altrimenti no.
La foto sotto descrive il funzionamento del ricevitore utilizzato nel progetto (VS1838).



Il ricevitore lavora in logica negata: quando il telecomando genera l'onda il ricevitore fornisce 0 in output. Invece quando il ricevitore non riceve nessun segnale restituisce 1.

Come nel file timer.f si ritrova il metodo AUTOCONFIG, inserito per rispettare la logica di libreria secondo la quale i file risultano indipendenti.
Tutti gli indirizzi dei registri sono calcolati applicando degli offset al registro di base, il cui indirizzo è stato impostato nel metodo AUTOCONFIG.

```
\ Legge il bit del pin 9 (output del ricevitore)
BASE_REGISTER 200034 +  CONSTANT GPLEV0
: INPUT GPLEV0 @ 400000 * 80000000 / ;          \ (    -- b1 )

\ Indirizzo del registro dove sono
\ memorizzati i microsendi passati dall'avvio della macchina
BASE_REGISTER 3004 +  CONSTANT CLO
: GCLOCK CLO @ ;                                \ (    -- b1 )
```

Fondamentale è il metodo INPUT.
Ricordando che il ricevitore manda il proprio output al pin 9, questo metodo accede al registro GPLEV0 e tramite una serie di shift lascia in cima allo stack soltanto il bit che ci interessa ovvero l'output del ricevitore. Il pin 9 viene configurato come input di default di conseguenza non si ha necessità di modificare il registro GPSET0.

I bit inviati dal telecomando sono tanti. In questo progetto poiché vengono utilizzati soltanto i pulsanti 2, 4, 6, 8, PLAY sono stati campionati soltanto 3 specifici punti della sequenza di bit ricevuta in modo da ottimizzare il campionamento.
Per sapere come scegliere questi tre punti vedere il capitolo Adattamento.

Configurazione telecomando

Per il progetto sono stati utilizzati due telecomandi differenti e, di conseguenza, la sequenza di bit è diversa tra questi. Per semplificare la fase di adattamento del codice in base al tipo di telecomando utilizzato sono state utilizzate delle variabili che contengono i punti di campionamento (SAMPLE_POINT1, SAMPLE_POINT2, SAMPLE_POINT3) e le codifiche dei pulsanti da riconoscere in base ai bit campionati (P2 -> pulsante 2, P4 -> pulsante 4, P6 -> pulsante 6, P8 -> pulsante 8, PP, pulsante play).

Queste variabili devono essere inizializzate prima dell'avvio del gioco in modo da consentire al sistema di riconoscere il telecomando. Per farlo sono forniti i metodi ENRICO e GIULIANO che lasciano sullo stack tutti i valori per inizializzare le variabili sopra indicate.

```
\ Caricano sullo stack i valori di campionamento
\ personali
: ENRICO                        \ (     -- b1 b2 b3 b4 b5 b6 b7 b8 )
    A410  A85C  B0F4
    0 3 4 2 5
;

: GIULIANO                      \ (     -- b1 b2 b3 b4 b5 b6 b7 b8 )
    B284  BB99  BFFE
    1 6 2 0 5
;
```

Infine il metodo SET_RECEIVER legge questi valori dallo stack e li memorizza nelle variabili.

```
111   \ Imposta i punti di campionamento e le
112   \ codifiche dei campioni
113   : SET_RECEIVER                    \ ( a1 a2 a3 a4 a5 a6 a7 a8 --     )
114       SP8  SP6  SPP  SP4  SP2
115       SSAMPLE_POINT3
116       SSAMPLE_POINT2
117       SSAMPLE_POINT1
118   ;
```

Prima di avviare il gioco basta quindi inserire il comando ENRICO SET_RECEIVER o GIULIANO SET_RECIVER per configurare uno dei due telecomandi.

CONTROL_RECEIVER1

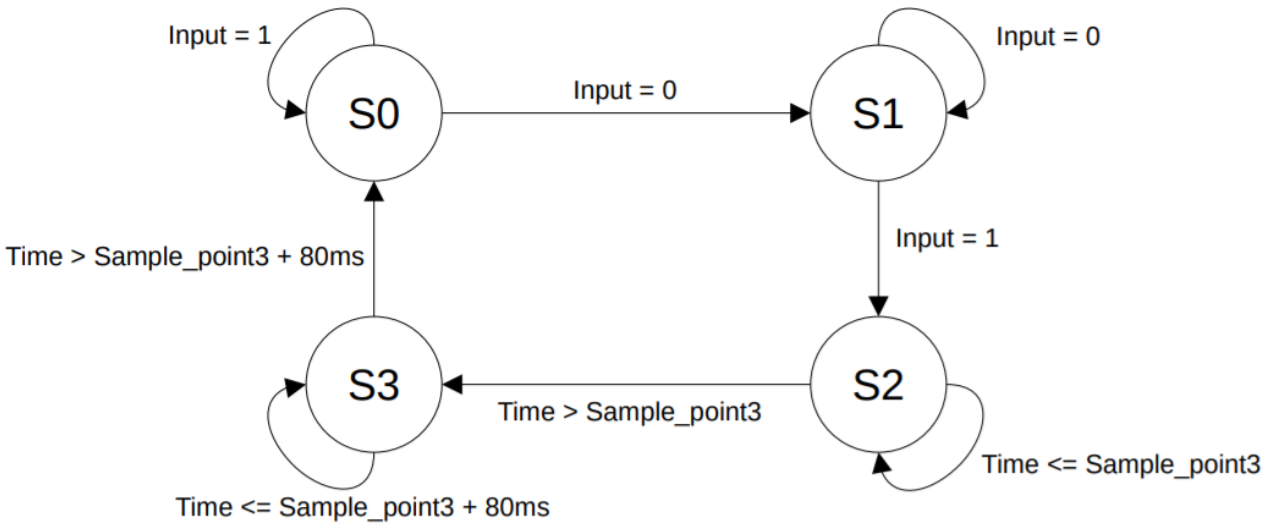
È il primo metodo che consente di leggere la sequenza di bit ricevuta.

```
134   \ Secondo metodo che permette la ricezione
135   \ di un valore tramite infrarossi
136   : CONTROL_RECEIVER1                                          \ (    --    )
137       GSTATE_RECEIVER
138       DUP   0 =  IF
139           INPUT  0 =  IF  1 SSTATE_RECEIVER  THEN
140       ELSE
141           DUP 2 =  IF
142               GCLOCK GTIMESAMPLE - GSAMPLE_POINT1 1388 - >  IF
143                   BEGIN  GCLOCK GTIMESAMPLE - GSAMPLE_POINT1 >  UNTIL  INPUT
144                   BEGIN  GCLOCK GTIMESAMPLE - GSAMPLE_POINT2 >  UNTIL  INPUT
145                   BEGIN  GCLOCK GTIMESAMPLE - GSAMPLE_POINT3 >  UNTIL  INPUT
146
147                   DECODE SSAMPLE 3 SSTATE_RECEIVER  GCLOCK STIMESAMPLE
148               THEN
149           ELSE
150               DUP 1 =  IF
151                   BEGIN  INPUT  1 =  UNTIL
152                   GCLOCK STIMESAMPLE  2 SSTATE_RECEIVER
153               ELSE
154                   GCLOCK GTIMESAMPLE - 13880 >  IF  0 SSTATE_RECEIVER  THEN
155       THEN THEN THEN
156       DROP
157   ;
```

Nelle righe 148 149 e 150 i metodi GSAMPLE_POINTi restituiscono i punti di campionamento memorizzati nelle variabili SAMPLE_POINTi.

Per evitare che il metodo blocchi il flusso di esecuzione è stato progettato come una macchina a stati. Infatti il metodo non contiene nessun ciclo.

Sotto è mostrato il funzionamento della macchina:



- **S0**: si controlla il bit in input dal ricevitore. Se è 1 vuol dire che non sta ricevendo e quindi si rimane in S0 altrimenti si passa in S1.
- **S1**: si attende che il bit in input diventi 1 e si memorizza il tempo in cui cambia. Questa operazione deriva dal fatto che con i particolari telecomandi utilizzati per svolgere il progetto il primo bit che viene ricevuto è sempre 0 per 9.1ms. Il meccanismo è sfruttato per sincronizzare il ricevitore. Una volta ricevuto il bit 1 si passa in S2.
- **S2**: si attende l'arrivo degli istanti in cui campionare il segnale.
  Una volta finito si passa allo stato S3.
- **S3**: si attende l'arrivo di tutti i bit. Il telecomando infatti trasmette il segnale per più tempo rispetto a quanto necessario ai fini del progetto: se si passasse subito in S0 il metodo comincerebbe di nuovo a campionare ricevendo la restante parte dei bit trasmessi.
  Passato un certo intervallo di tempo si ritorna in S0.

Lo stato viene memorizzato nella variabile STATE_RECEIVER.

```
\ Contiene lo stato del metodo
\ CONTROL_RECEIVER
VARIABLE STATE_RECEIVER
: GSTATE_RECEIVER STATE_RECEIVER @ ;            \ (     -- b1 )
: SSTATE_RECEIVER STATE_RECEIVER ! ;            \ ( a1 --     )
```

Una volta ricevuti i bit questi vengono codificati tramite il metodo DECODE.

```
\ Decodifica i bit ricevuti
: DECODE                          \ ( a1 a2 a3 --  b1 )
    SWAP 2 * +
    SWAP 4 * +
    DUP GP2 =  IF  2  ELSE
    DUP GP4 =  IF  4  ELSE
    DUP GPP =  IF  5  ELSE
    DUP GP6 =  IF  6  ELSE
    DUP GP8 =  IF  8  ELSE
                  -1
    THEN THEN THEN THEN THEN
    SWAP DROP
;
```

Questo legge i tre bit campionati e restituisce il valore corrispondente. I metodi GPi restituiscono la codifica del pulsante i-esimo contenuta nella variabile Pi.

CONTROL_RECEIVER2

Secondo metodo che permette di ricevere i bit.

```
\ Interfaccia che sfrutta CONTROL_RECEIVER1
\ e blocca il flusso di esecuzione
: CONTROL_RECEIVER2                          \ (  --  )
    1 SSTATE_RECEIVER
    BEGIN  INPUT 0 =  UNTIL
    BEGIN
        CONTROL_RECEIVER1
    GSTATE_RECEIVER 0 =  UNTIL
;
```

Questo sfrutta CONTROL_RECEIVER1. Rispetto a questo blocca il flusso di esecuzione fin quando non viene campionato un valore.

In entrambi i casi il valore campionato viene memorizzato dentro la variabile SAMPLE che può essere utilizzata esternamente per leggere il valore ricevuto.

```
\ Contiene il valore campionato
VARIABLE SAMPLE
: GSAMPLE SAMPLE @ ;            \ (     -- b1 )
: SSAMPLE SAMPLE ! ;            \ ( a1 --     )
```

4. File specifici

In questo capitolo vengono trattati i file:

- Snake.f
- Interfaces.f
- Main.f

Questi contengono le istruzioni specifiche per il corretto funzionamento del gioco. Sfruttano i file visti nel precedente capitolo.

4.1 Snake.f

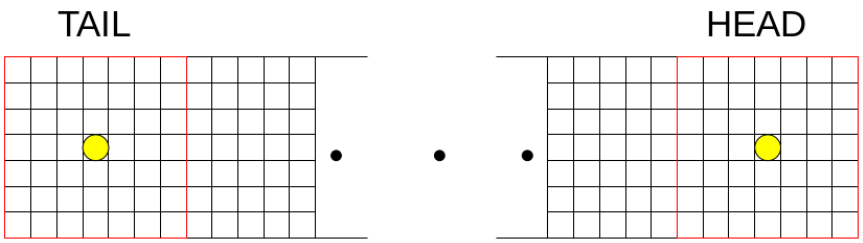
Nel file sono contenuti tutti i metodi di gestione del serpente.

Il primo ad essere eseguito è PRINT_SNAKE:

```
\ Stampa il serpente sullo schermo
: PRINT_SNAKE                                        \ (  --  )
    WHITE1 SCOLOR
    GFRAMEBUFFER 1BF670 +  SPIXEL  7 C8 PRINT_PIXEL
    GFRAMEBUFFER 1C2980 +  SHEAD
    GFRAMEBUFFER 1C267C +  STAIL
    0 SDEAD
    0 SSNAKE_DIR
    0 SSTATE_APPLE
;
```

Questo stampa a schermo un serpente di dimensione 7x200 pixel di colore WHITE1, ovvero una tonalità di bianco (0x00FFFFFE) usata per distinguere il serpente dai bordi dello schermo.

Vengono inoltre inizializzate le variabili REF_HEAD e REF_TAIL contenenti rispettivamente gli indirizzi di testa e coda del serpente:



Come illustrato nella figura si considera il serpente come una matrice di pixel. Testa e coda sono visti come dei quadrati di dimensione 7x7 pixel e i loro indirizzi di riferimento si trovano al centro dei due quadrati. Questo tipo di rappresentazione consente una maggiore semplicità nella gestione del movimento.
Testa e coda sono inoltre considerate come entità distinte.

Prima di mostrare il modo in cui viene gestito il gioco è utile descrivere alcuni metodi generali utilizzati all'interno del codice:

- **INC_O**: incrementa orizzontalmente un indirizzo del numero di pixel dati in input

```
\ Incrementa orizzontalmente un indirizzo
\ del numero di pixel passati in input
: INC_O                                          \ ( a1 a2 -- b1 )
    4 * +
    DUP   FFFFF000 AND
    SWAP  00000FFF AND
    DUP  FEC >  IF  FE0 -  ELSE
    DUP  010 <  IF  FE0 +  THEN THEN +
;
```

  Il metodo effettua anche il controllo in modo che, quando un indirizzo incrementato ha oltrepassato i bordi sinistro o destro, questo viene riportato dall'atro lato dello schermo. Si crea una sorta di effetto modulare in cui il serpente uscito da destra nello schermo rientra da sinistra, e viceversa.

- **INC_V**: incrementa verticalmente un indirizzo del numero di pixel dati in input

```
\ Incrementa verticalmente un indirizzo
\ del numero di pixel passati in input
: INC_V                                          \ ( a1 a2 -- b1 )
    1000 * +
    DUP   FF000FFF AND
    SWAP  00FFF000 AND
    DUP   BF5000 >  IF  278000 - ELSE
    DUP   97E000 <  IF  278000 + THEN THEN +
;
```

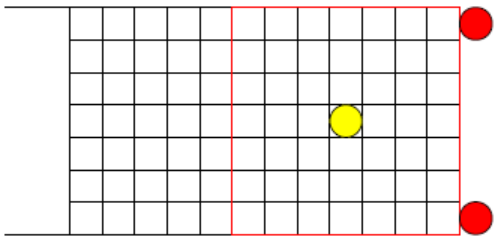
  Effettua lo stesso controllo del metodo INC_O, con l'unica differenza che questo viene fatto sui bordi superiore e inferiore.

- **CHECK_FORWARD**: Controlla se il colore di due pixel davanti al serpente è uguale a quello impostato alla chiamata del metodo. Lascia sullo stack un valore pari a 0 (false) o -1 (true) in base al verificarsi della precedente condizione.

```
\ Controlla il colore dei pixel di fronte al serpente
: CHECK_FORWARD                                                    \ (  -- b1 )
    GHEAD  GHEAD_DIR
    DUP  0 =  IF  DROP  4 INC_O  DUP  -3 INC_V @  SWAP  3 INC_V @  ELSE
    DUP  1 =  IF  DROP  4 INC_V  DUP  -3 INC_O @  SWAP  3 INC_O @  ELSE
    DUP  2 =  IF  DROP -4 INC_O  DUP  -3 INC_V @  SWAP  3 INC_V @  ELSE
                  DROP -4 INC_V  DUP  -3 INC_O @  SWAP  3 INC_O @
    THEN  THEN  THEN
    GCOLOR =  SWAP  GCOLOR =  OR
;
```

Il colore da controllare viene sempre impostato esternamente in modo che il metodo possa essere utilizzato per diversi scopi come il controllo sulla mela o sulla morte del serpente.

Il modo in cui lavora può essere rappresentato attraverso il seguente schema:



Basta controllare solamente il colore dei due pixel evidenziati in rosso. Infatti gli unici ostacoli che potrebbe incontrare il serpente sono la mela e sé stesso e, in entrambi i casi, hanno una dimensione maggiore di 5 pixel (distanza tra i punti rossi). Sarà sufficiente controllare solo questi due punti per essere sicuri di individuare un ostacolo.

La parte principale del file è costituita dal metodo CONTROL_SNAKE che si comporta come una interfaccia e consente di chiamare tutti i metodi necessari per il corretto funzionamento del gioco:

```
\ Svolge tutti i controlli necessari
\ e muove il serpente
: CONTROL_SNAKE                                    \ (  --  )
    CONTROL_DIRECTION
    CHECK_APPLE
    CHECK_DEAD
    HEAD MOVE
    TAIL MOVE
;
```

Ognuno dei metodi richiamati da CONTROL_SNAKE ha un grande numero di funzionalità:

CONTROL_DIRECTION

```
\ Controlla il valore contenuto nella variabile SAMPLE
\ e in base a questo modifica la direzione del serpente
: CONTROL_DIRECTION                                                    \ (  --  )
    GHEAD_DIR  GSAMPLE
    DUP 2 = IF  DROP DUP  0 =  SWAP  2 =  OR IF  3 SHEAD_DIR  THEN  ELSE
    DUP 4 = IF  DROP DUP  1 =  SWAP  3 =  OR IF  2 SHEAD_DIR  THEN  ELSE
    DUP 6 = IF  DROP DUP  1 =  SWAP  3 =  OR IF  0 SHEAD_DIR  THEN  ELSE
    DUP 8 = IF  DROP DUP  0 =  SWAP  2 =  OR IF  1 SHEAD_DIR  THEN  ELSE
                2DROP
    THEN THEN THEN THEN
;
```

Il compito del metodo è quello di leggere l'ultimo valore campionato che si trova nella variabile SAMPLE (scritta dal ricevitore) e gestire di conseguenza la direzione della testa del serpente.
Il ricevitore è da considerarsi come un programma a sé stante che riceve segnali in input e decodifica ciò che è stato ricevuto. Il dato viene poi interpretato da CONTROL_DIRECTION per muovere il serpente. In questo modo è possibile evitare direzioni non consentite.

ES. Se il serpente si muove verso NORD e viene ricevuto il comando per girare a SUD, questo viene ignorato.

Per quanto riguarda le direzioni di testa e coda, queste sono mantenute in un'unica variabile SNAKE_DIR:

```
\ Contiene le direzioni di testa e coda
VARIABLE SNAKE_DIR
: GSNAKE_DIR   SNAKE_DIR @ ;                          \ (    -- b1 )
: SSNAKE_DIR   SNAKE_DIR ! ;                          \ ( a1 --    )

\ Consentono di accedere in maniera diretta
\ ai bit memorizzati nella variabile SNAKE_DIR
: GHEAD_DIR   GSNAKE_DIR 3 AND ;                      \ (    -- b1 )
: GTAIL_DIR   GSNAKE_DIR C AND 4 / ;                  \ (    -- b1 )
: SHEAD_DIR   GSNAKE_DIR C AND +  SSNAKE_DIR ;        \ ( a1 --    )
: STAIL_DIR   4 * GSNAKE_DIR 3 AND +  SSNAKE_DIR ;    \ ( a1 --    )
```

Le direzioni sono 4 e sono codificate in 2 bit:

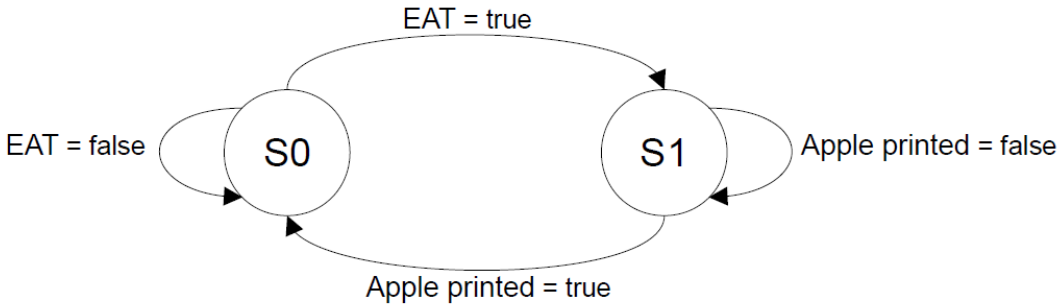
- EST → 00
- SUD → 01
- OVEST → 10
- NORD → 11

Nella variabile i 2 bit meno significativi contengono la direzione della testa, i 2 successivi quella della coda.
Sono riportati anche alcuni metodi che consentono di avere un accesso diretto ai bit corretti.

CHECK_APPLE

Il metodo si occupa di verificare se di fronte al serpente c'è una mela, mangiarla incrementare il punteggio e stamparne una nuova.

È stato progettato come una macchina a stati:



- **S0**: controlla se il serpente deve mangiare una mela e in tal caso la mangia e incrementa il punteggio (metodi EAT + UPDATE_SCORE) passando allo stato S1.
- **S1**: stampa una nuova mela (metodo PRINT_APPLE) e passa allo stato S0

Il motivo per cui è stato necessario adottare questo ragionamento è perché il tempo di stampa della mela è molto variabile e dipende dal numero di pixel liberi. Se infatti lo schermo fosse quasi interamente occupato dal serpente il metodo perderebbe molti millisecondi per terminare.

Attraverso questa implementazione viene assegnato a PRINT_APPLE un tempo massimo di 700us. In questo modo CONTROL_SNAKE non passa mai il set da 2ms assegnatogli (maggiori dettagli nel capitolo main.f).
Infatti nel caso peggiore CHECK_APPLE non riesce a stampare una mela entro quell'intervallo di tempo e resta nello stato S1. Si prosegue con la normale

esecuzione del gioco per poi, dopo 4ms, tornare ad eseguirlo. Alla ripresa dell'esecuzione, essendo ancora nello stato S1, il metodo prova nuovamente a stampare la mela per altri 700us. Si procede in questo modo fin quando questa non è stata stampata.

Analizziamo ora il codice:

CHECK_APPLE controlla il proprio stato leggendo dalla variabile STATE_APPLE:

```
\ In base al suo stato consente al serpente di
\ mangiare la mela e incrementare il punteggio
\ o stampa una nuova mela
: CHECK_APPLE                                   \ (  --  )
    GSTATE_APPLE  0 = IF
        RED SCOLOR  CHECK_FORWARD  IF
            EAT  UPDATE_SCORE  1 SSTATE_APPLE
        THEN
    ELSE
        PRINT_APPLE  SSTATE_APPLE
    THEN
;
```

Se si trova nello stato S0 controlla che il serpente abbia di fronte una mela. In tal caso chiama prima il metodo EAT che si occupa di incrementare la lunghezza del serpente e di cancellare la mela appena mangiata

```
\ Incrementa la lunghezza del serpente e cancella
\ la mela appena mangiata
: EAT   GTAIL GTAIL_DIR                                         \ (  --  )
    DUP  0 =  IF  DROP -7 INC_O DUP  -3 INC_O -3 INC_V SPIXEL  ELSE
    DUP  1 =  IF  DROP -7 INC_V DUP  -3 INC_V -3 INC_O SPIXEL  ELSE
    DUP  2 =  IF  DROP  7 INC_O DUP  -4 INC_O -3 INC_V SPIXEL  ELSE
                  DROP  7 INC_V DUP  -4 INC_V -3 INC_O SPIXEL
    THEN THEN THEN
    STAIL
    WHITE1 SCOLOR  7 7 PRINT_PIXEL

    GAPPLE SPIXEL  BLACK SCOLOR  7 7 PRINT_PIXEL
;
```

Successivamente chiama il metodo UPDATE_SCORE che si occupa di incrementare il punteggio e stamparlo

```
\ Contiene il punteggio
VARIABLE SCORE
: GSCORE   SCORE @ ;                                  \ (  -- b1 )
: SET_SCORE   0 SCORE ! ;                             \ (  --    )
: UPDATE_SCORE                                        \ (  --    )
    3E9142A8 SPIXEL  BLACK SCOLOR  GSCORE DUP PRINT_NUMBER
    3E9142A8 SPIXEL  WHITE SCOLOR  1 +    DUP PRINT_NUMBER
    SCORE !
;
```

Viceversa se il metodo si trova nello stato S1 viene chiamato il metodo PRINT_APPLE che si occupa di stampare la mela entro 700us.

```
\ Stampa la mela 7x7 in un punto casuale libero dello schermo
: PRINT_APPLE                                                          \ (  -- b1)
    GCLOCK
    BEGIN
        GHEAD GCLOCK DUP INC_O -1 * INC_V  BLR BHL -  MOD  BHL +  DUP SAPPLE
        DUP @                   BLACK <>                  IF  0  ELSE
        DUP 1C   + @            BLACK <>                  IF  0  ELSE
        DUP 7000 + DUP @        BLACK <>  SWAP BLR >  OR  IF  0  ELSE
        DUP 1C +  7000 + DUP @  BLACK <>  SWAP BLR >  OR  IF  0  ELSE
                                                                 -1
        THEN  THEN  THEN  THEN
    NIP DUP ROT  GCLOCK SWAP -  2BC >  OR  UNTIL

    IF GAPPLE SPIXEL  RED SCOLOR  7 7 PRINT_PIXEL  0  ELSE  1  THEN
;
```

CHECK_DEAD

```
\ Controlla se il serpente ha sbattuto su se stesso
: CHECK_DEAD   WHITE1 SCOLOR  CHECK_FORWARD  IF  1 SDEAD  THEN ;      \ (  --  )
```

Il metodo utilizza COLOR_FORWARD spiegato prima. Se almeno uno dei due pixel controllati è dello stesso colore del serpente significa che ha sbattuto su sé stesso. In caso affermativo la variabile DEAD viene impostata a 1.

```
\ Contiene lo stato del serpente
\ 0 -> vivo ,  1 -> morto
VARIABLE DEAD
: GDEAD   DEAD @ ;                    \ (     -- b1 )
: SDEAD   DEAD ! ;                    \ ( a1 --     )
```

Questo cambiamento causa la conclusione della partita. Infatti la variabile viene controllata nel metodo PLAY del file main.f all'inizio di ogni ciclo di esecuzione: un suo valore pari ad 1 causa l'uscita dal ciclo e l'inizializzazione della schermata di game over (ulteriori dettagli nel file main.f).

MOVE

Si occupa del movimento del serpente.

```
\ Muove la testa o la coda del serpente stampando
\ delle righe o colonne bianche o nere
: MOVE                                                  \ ( a1 a2 a3 a4 --   )
    0 = IF  SHEAD  ELSE  STAIL  THEN
    SWAP
    DUP  0 =  SWAP  2 =  OR  IF
        -4 INC_V  7
        BEGIN
            SWAP 1 INC_V DUP GCOLOR SWAP !
            SWAP 1 - DUP
        0 =  UNTIL  DROP DROP
    ELSE
        -4 INC_O  7
        BEGIN
            SWAP 1 INC_O DUP GCOLOR SWAP !
            SWAP 1 - DUP
        0 =  UNTIL  DROP DROP
    THEN
;
```

Necessita di alcuni valori in input per comprendere come muovere testa e coda. Questi valori vengono forniti dai due metodi HEAD e TAIL che devono essere eseguiti prima di MOVE.
Il metodo consente di impostare il colore della colonna esternamente; saranno i due metodi a gestirlo (bianco per avanzare la testa, nero per cancellare la coda).

HEAD

```
\ Lascia sullo stack tutti i dati necessari a MOVE per
\ muovere la testa e stampare una riga o colonna bianca
: HEAD                                                  \ (  -- b1 b2 b3 b4 )
    GHEAD GHEAD_DIR
    DUP 0 =  IF  SWAP  1 INC_O  DUP  3 INC_O   ELSE
    DUP 1 =  IF  SWAP  1 INC_V  DUP  3 INC_V   ELSE
    DUP 2 =  IF  SWAP -1 INC_O  DUP -3 INC_O   ELSE
                 SWAP -1 INC_V  DUP -3 INC_V
    THEN THEN THEN

    WHITE1 SCOLOR  SWAP  0
;
```

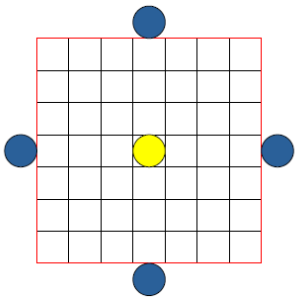
Il metodo controlla la direzione della testa del serpente e calcola la sua nuova posizione.
Al termine della sua esecuzione lascia quattro valori sullo stack:

- la direzione verso la quale la testa si sta muovendo
- l'indirizzo a partire dal quale stampare una colonna o riga bianca di lunghezza 7 pixel in modo da simulare un movimento della testa
- la nuova posizione che avrà la testa dopo il movimento
- un flag per comunicare a MOVE di lavorare sulla testa

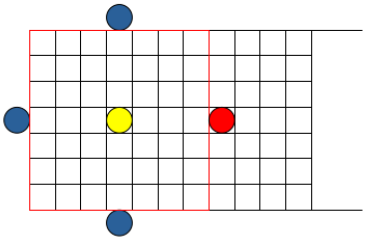
TAIL

Prima di mostrare il codice è necessario spiegare la logica utilizzata per gestire il movimento.

Per muovere la coda è necessario replicare tutti i movimenti svolti dalla testa.
Il grafico sottostante rappresenta il quadrato 7x7 pixel della coda e il puntino giallo corrisponde all'indirizzo scelto come riferimento.



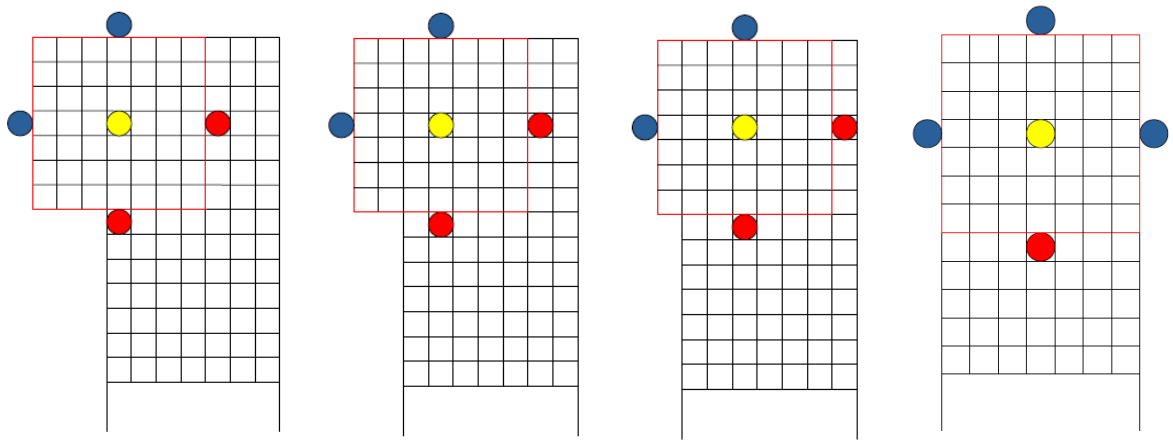
L'idea è quella di controllare, ad ogni passo, tutti i 4 pixel indicati dai puntini blu nello schema.



Nell'esempio sopra il corpo del serpente è a EST. Controllando i punti blu si nota che solamente quello a destra rispetto al riferimento ha un colore uguale a quello del serpente, quindi la coda viene spostata verso destra.

Questo ragionamento risulta corretto solamente quando il serpente va dritto.

Nel caso in cui la coda arriva in prossimità di una curva vengono trovati due pixel dello stesso colore del serpente. In questo caso la coda deve continuare a seguire la sua direzione precedente in quanto dovrà ancora spostarsi di un certo numero di pixel per poter essere nuovamente allineata col resto del serpente. La sequenza sottostante illustra quanto detto.



Questo sistema di movimento porterebbe ad un errore nel caso in cui venissero effettuati due cambi di direzione in 2 pixel successivi. In questo caso la coda seguirebbe la direzione sbagliata.
Per questo progetto il problema non si pone in quanto, come spiegato nella sezione del main.f, i controlli sulla direzione e il successivo movimento del serpente vengono svolti ogni 4ms. Un utente dovrebbe quindi modificare due volte la direzione in 8ms, ma per campionare ogni segnale in input sono necessari almeno 100ms. Il tempo minimo per cambiare la direzione due volte è quindi di almeno 200ms, intervallo molto grande rispetto agli 8ms che causerebbero l'errore.

Il codice che implementa quanto detto è:

```
\ Lascia sullo stack tutti i dati necessari a MOVE per
\ muovere la coda e stampare una riga o colonna nera
: TAIL                                                    \ (  -- b1 b2 b3 b4 )
    0
    GTAIL  4 INC_O @  WHITE1 =  IF  1 +  0 SWAP  THEN
    GTAIL  4 INC_V @  WHITE1 =  IF  1 +  1 SWAP  THEN
    GTAIL -4 INC_O @  WHITE1 =  IF  1 +  2 SWAP  THEN
    GTAIL -4 INC_V @  WHITE1 =  IF  1 +  3 SWAP  THEN

    2 =  IF  2DROP  GTAIL_DIR  THEN

    GTAIL SWAP
    DUP 0 = IF  SWAP  DUP -3 INC_O  SWAP  1 INC_O  ELSE
    DUP 1 = IF  SWAP  DUP -3 INC_V  SWAP  1 INC_V  ELSE
    DUP 2 = IF  SWAP  DUP  3 INC_O  SWAP -1 INC_O  ELSE
                SWAP  DUP  3 INC_V  SWAP -1 INC_V
    THEN  THEN  THEN

    ROT DUP STAIL_DIR  ROT ROT  BLACK SCOLOR  1
;
```

La prima parte conta il numero di pixel (puntini blu) dello stesso colore del serpente.

La successiva, in base al risultato dei controlli precedenti, lascia sullo stack quattro valori:

- la direzione verso la quale la coda si sta muovendo
- l'indirizzo a partire dal quale stampare una colonna o riga nera di lunghezza 7 pixel in modo da simulare un movimento della coda
- la nuova posizione che avrà la coda dopo il movimento
- un flag per comunicare a MOVE di lavorare sulla coda

4.2 Interfaces.f

Il file contiene i metodi per stampare tutte le tre interfacce di gioco:

- INITIALIZE_SELECTION
- INITIALIZE_PLAY
- INITIALIZE_GAME_OVER

Nessuna delle tre svolge alcuna operazione logica: lo scopo del file è soltanto quello di inizializzare i valori di partenza e mostrare le semplici schermate.

Prima di spiegare come lavorano le interfacce è utile mostrare alcuni metodi:

- **PRINT_BUTTONS**: stampa sullo schermo i due pulsanti PLAY e EXIT le cui funzioni sono rispettivamente quella di avviare il gioco passando alla schermata successiva, e uscire dal gioco causando l'uscita dal ciclo di esecuzione (ulteriori dettagli nel file main.f).

```
\ Stampa pulsanti PLAY ed EXIT
: PRINT_BUTTONS
    6 SCHAR_SIZE
    GFRAMEBUFFER 2126EC +  SPIXEL  .P. .L. .A. .Y. 4 PRINT_WORD
    GFRAMEBUFFER 27C6EC +  SPIXEL  .E. .X. .I. .T. 4 PRINT_WORD
```

- **CONTROL_BUTTONS**: consente di selezionare i pulsanti PLAY e EXIT. Attende la ricezione di un segnale in input. Se il valore ricevuto è 2 seleziona il pulsante PLAY, se è 8 seleziona il pulsante EXIT. Di default viene selezionato il pulsante PLAY.

```
\ Permette la selezione del pulsante PLAY o EXIT
: CONTROL_BUTTONS                                                \ (    --    )
    0
    2 SBORDER_S  59 SBORDER_H  C0 SBORDER_W
    WHITE SCOLOR  GFRAMEBUFFER 1FE674 +  SPIXEL BORDER
    BEGIN
        CONTROL_RECEIVER2  GSAMPLE
        DUP 2 =  IF
            SWAP DUP  1 =  IF
                1 - BLACK SCOLOR  GFRAMEBUFFER 268674 +  SPIXEL BORDER
                    WHITE SCOLOR  GFRAMEBUFFER 1FE674 +  SPIXEL BORDER
            THEN
            SWAP
        ELSE
            DUP 8 =  IF
                SWAP DUP  0 =  IF
                    1 + BLACK SCOLOR  GFRAMEBUFFER 1FE674 +  SPIXEL BORDER
                        WHITE SCOLOR  GFRAMEBUFFER 268674 +  SPIXEL BORDER
                THEN
                SWAP
            THEN
        THEN
        GCLOCK STIME
    5 = UNTIL
;
```

Per quanto detto prima, il file in questione non utilizza mai direttamente questo metodo. La logica di selezione è gestita in main.f

- **PAUSE_INTERFACE:** consente di fermare l'esecuzione del gioco per un tempo indefinito.

```
\ Blocca il flusso di esecuzione e attende il comando
\ per ricominciare
: PAUSE_INTERFACE                                                    \ (    --    )
    GFRAMEBUFFER 1A5E0 +  SPIXEL  WHITE SCOLOR  A SCHAR_SIZE
    .P. .A. .U. .S. .E. 5 PRINT_WORD

    BEGIN  CONTROL_RECEIVER2 GSAMPLE 5 =  UNTIL

    GFRAMEBUFFER 1A5E0 +  SPIXEL  BLACK SCOLOR
    .P. .A. .U. .S. .E. 5 PRINT_WORD

    GHEAD_DIR
    DUP 0 =  IF 6 SSAMPLE  ELSE
    DUP 1 =  IF 8 SSAMPLE  ELSE
    DUP 2 =  IF 4 SSAMPLE  ELSE
                   2 SSAMPLE
    THEN THEN THEN
    DROP
;
```

Il metodo viene eseguito ogni volta che l'utente preme il pulsante PLAY sul telecomando e consiste nella stampa della scritta PAUSE nella parte superiore dello schermo.
Successivamente il metodo rimane in attesa di ricevere il segnale PLAY in modo da poter riprendere la normale esecuzione del gioco.

INITIALIZE_SELECTION

```
\ Inizializazione di SELECTION
: INITIALIZE_SELECTION                                          \ (    --    )
    CLEAR   WHITE SCOLOR  GFRAMEBUFFER SPIXEL
    300 SBORDER_H  400 SBORDER_W  4 SBORDER_S  BORDER
    18 SCHAR_SIZE  GFRAMEBUFFER BC290 +  SPIXEL
    .S. .N. .A. .K. .E. 5 PRINT_WORD
    PRINT_BUTTONS
;
```

Il metodo mostra la schermata principale del gioco in cui sono contenuti il nome del gioco e i due pulsanti PLAY e EXIT, stampati attraverso il metodo PRINT_BUTTONS precedentemente spiegato.

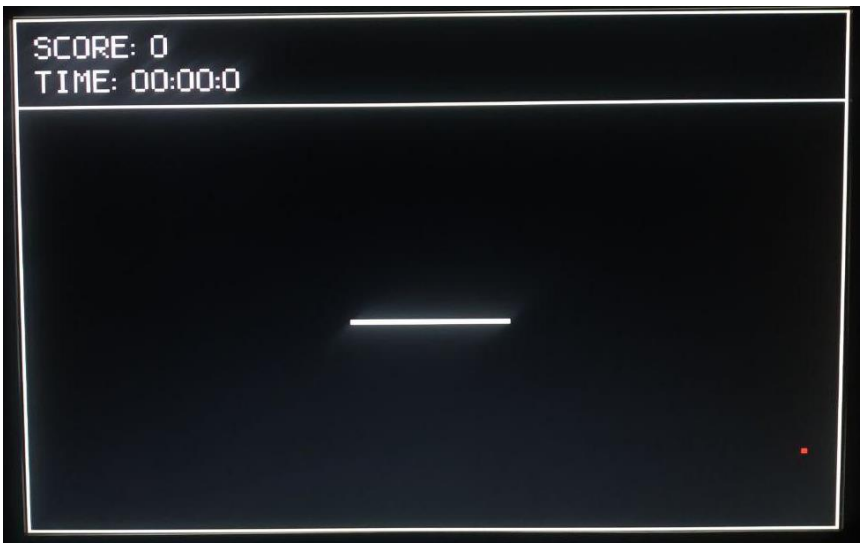
La schermata si presenta nel seguente modo:

INITIALIZE_PLAY

```
\ Inizializzazione di PLAY                                    \ (    --    )
: INITIALIZE_PLAY
    CLEAR   WHITE SCOLOR   GFRAMEBUFFER SPIXEL
    300 SBORDER_H  400 SBORDER_W  4 SBORDER_S  BORDER

    3E97A000 SPIXEL  4 400 PRINT_PIXEL

    4 SCHAR_SIZE
    GFRAMEBUFFER 1A068 +  SPIXEL  .S. .C. .O. .R. .E. .DP.  6 PRINT_WORD
    GFRAMEBUFFER 1A2A8 +  SPIXEL   SET_SCORE   0 PRINT_NUMBER

    GFRAMEBUFFER 4C068 +  SPIXEL  .T. .I. .M. .E. .DP. 5 PRINT_WORD
    GFRAMEBUFFER 4C248 +  STIMER_PIXEL
    WHITE STIMER_COLOR   4 STIMER_SIZE   SET_TIMER

    PRINT_SNAKE  PRINT_APPLE  DROP

    0 SSTATE_RECEIVER
    BEGIN  CONTROL_RECEIVER2  GSAMPLE 5 =  UNTIL
    6 SSAMPLE
    SAVE_TIMER
;
```

È la schermata di gioco in cui vengono mostrati il tempo di gioco (espresso in minuti, secondi e decimi di secondo), il punteggio, e un rettangolo di gioco contenente il serpente e la mela. Questi ultimi vengono stampati attraverso i metodi PRINT_SNAKE e PRINT_APPLE analizzati nella sezione Snake.f

La schermata si presenta nel seguente modo:



INITIALIZE_GAME_OVER

Viene mostrata al termine della fase di gioco quando il serpente ha sbattuto su sé stesso. Indica la fine della partita.

```
\ Inizializzazione di GAME_OVER
: INITIALIZE_GAME_OVER                                                    \ (    --    )
    CLEAR  WHITE SCOLOR  GFRAMEBUFFER SPIXEL
    300 SBORDER_H  400 SBORDER_W  4 SBORDER_S  BORDER

    GFRAMEBUFFER 79234 +  SPIXEL  E SCHAR_SIZE
    .G. .A. .M. .E. .SPACE. .O. .V. .E. .R.  9 PRINT_WORD

    GFRAMEBUFFER 13E604 + SPIXEL  4 SCHAR_SIZE   .T. .I. .M. .E. .DP.  5 PRINT_WORD
    GFRAMEBUFFER 13E7E4 + SPIXEL

    GMINUTES  DUP A / 0 =  IF  0 PRINT_NUMBER  THEN
    PRINT_NUMBER  .DP. PRINT_CHAR  8 INC_PIXEL_O

    GSECONDS  DUP A / 0 =  IF  0 PRINT_NUMBER  THEN
    PRINT_NUMBER  .DP. PRINT_CHAR  8 INC_PIXEL_O

    GDECSECONDS PRINT_NUMBER

    GFRAMEBUFFER 182604 +  SPIXEL  .S. .C. .O. .R. .E. .DP. 6 PRINT_WORD
    GFRAMEBUFFER 182844 +  SPIXEL  GSCORE PRINT_NUMBER

    PRINT_BUTTONS
;
```

La schermata è composta dalla scritta GAME OVER, dalla durata della partita (espresso in minuti, secondi e decimi di secondo) e dal punteggio.
Include inoltre i pulsanti PLAY e EXIT tramite i quali un utente può decidere se cominciare una nuova partita o terminare gioco.

La schermata si presenta nel seguente modo:

4.3 Main.f

Questo file contiene la struttura del gioco e il comando GAME per avviarlo.

```
\ Struttura GAME
: GAME                          \ (    --    )
    SELECTION
    0 =  IF  PLAY  THEN
    END_GAME
;
```

GAME contiene le sezioni principali del gioco le quali richiamano i metodi spiegati nei capitoli precedenti.
Queste sono:

- SELECTION

```
\ Struttura SELECTION
: SELECTION                          \ (    --  b1 )
    INITIALIZE_SELECTION
    CONTROL_BUTTONS
;
```

  Mostra la schermata iniziale che dà la possibilità di avviare il gioco o di uscire. Lascia un valore sullo stack indicante la scelta del giocatore.

- PLAY
  Il cuore del gioco. Viene mostrato successivamente per una spiegazione più approfondita.

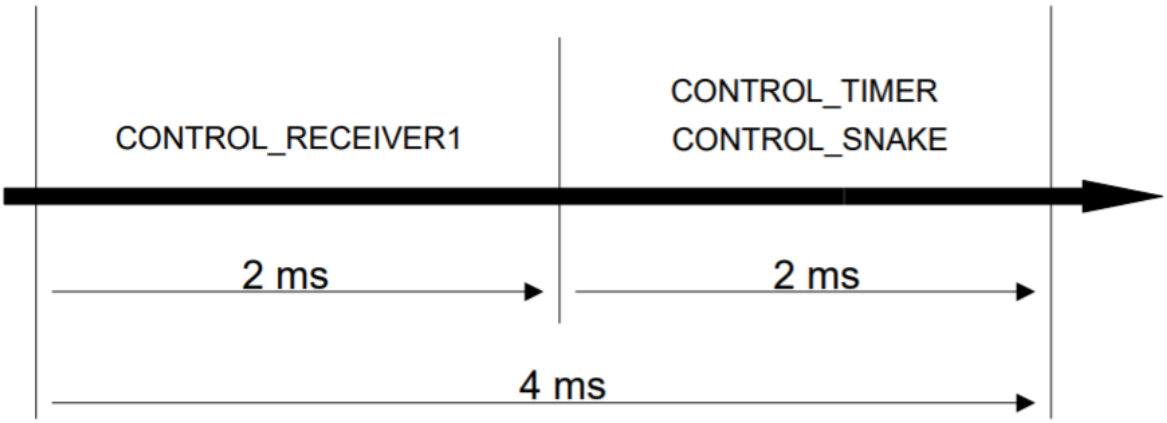
- END_GAME

```
\ Struttura END_GAME
: END_GAME                          \ (    --    )
    CLEAR
;
```

  Richiama il metodo CLEAR presente nel file interfaces.f che colora di nero tutti i pixel dello schermo. È stato creato per motivi strutturali.

PLAY

Per spiegare con maggiore semplicità PLAY si osservi il grafico sotto, che rappresenta le tempistiche e i metodi richiamati in una sola esecuzione del ciclo (parte principale del metodo).



Tale approccio è stato adottato per evitare il blocco del serpente a causa del campionamento. Quest'ultima operazione richiederebbe almeno 100ms e per mantenere la velocità stabilita, si ha la necessità di far avanzare il serpente di 1 pixel ogni 4ms. Per questo motivo si è scelto di creare un intervallo temporale di tale lunghezza e di dividerlo in 2 slot da 2ms. In ognuno di questi verranno richiamati i vari metodi cercando di rispettare le tempistiche imposte. Questa suddivisione permette così di eseguire ogni 4ms CONTROL_RECEIVER1 e CONTROL_SNAKE.

Si ricorda:

- CONTROL_RECEIVER1: gestisce l'output del ricevitore decodificando i bit ricevuti.
- CONTROL_SNAKE: gestisce tutto ciò che è strettamente legato al serpente come la mela, il punteggio, il suo movimento e l'incremento delle sue dimensioni.

La tabella sotto mostra per ogni metodo e condizione di esecuzione il tempo (nel caso peggiore) che questo occupa nello slot assegnato:

METODO	INDICAZIONI	TEMPO DI ESECUZIONE (µs)
CONTROL_RECEIVER1	Stato S0	100
	Stato S1	5000
	Stato S2 Flusso di esecuzione non bloccato	100
	Stato S2. Flusso di esecuzione bloccato	8300
	Stato S3	100
CONTROL_SNAKE	Movimento	525
	Movimento, mangia mela e aggiornamento punteggio	1000
	Movimento e stampa mela	1225
CONTROL_TIMER	Incremento decimi di secondo, secondi e minuti	580

Dalla tabella si nota che il primo slot, occupato solo da CONTROL_RECEIVER1, negli stati S1 e S2 con flusso di esecuzione bloccato sforerà i 2ms assegnati. In questi casi il serpente ritarda la stampa di un tempo massimo di 6ms, intervallo troppo piccolo da percepire nel caso di questo progetto. Inoltre, la maggior parte delle volte, CONTROL_RECIVER1 si troverà in stati differenti rendendo il problema ancora più impercettibile.

Compresa tale logica viene sotto riportato il codice.

```
18  \ Struttura PLAY
19  : PLAY                                          \ (    --    )
20      INITIALIZE_PLAY
21      BEGIN
22
23          GCLOCK
24          CONTROL_RECEIVER1
25          BEGIN  DUP GCLOCK SWAP - 7D0 >  UNTIL
26
27          GSAMPLE 5 = IF  PAUSE_INTERFACE  THEN
28
29          CONTROL_SNAKE
30          CONTROL_TIMER
31          GDEAD 1 = IF  GAME_OVER  THEN
32          BEGIN  DUP GCLOCK SWAP - FA0 >  UNTIL
33
34          DROP
35      GDEAD 1 =  UNTIL
36  ;
```

Le righe 23-25 rappresentano il primo slot. In questo viene eseguito CONTROL_RECEIVER e un ciclo che consente attendere la fine dello slot nel caso in cui questo termini prima di 2ms.
Riga 27 controlla se l'utente ha premuto il pulsante PLAY per mettere in pausa il gioco. In tal caso viene chiamato PAUSE_INTERFACE che blocca il flusso di esecuzione.
Righe 29-30 vengono richiamati i due metodi CONTROL_SNAKE e CONTROL_TIMER.
Riga 31 controlla se il giocatore ha terminato la partita e, in tal caso, viene chiamato GAME_OVER.
Infine il loop che permette di attendere la fine dei 2ms del secondo slot.

Sotto è riportata la struttura di GAME_OVER.

```
5   \ Struttura GAME_OVER
6   : GAME_OVER                                 \ (    --    )
7       INITIALIZE_GAME_OVER
8       CONTROL_BUTTONS
9       0 = IF  INITIALIZE_PLAY  THEN
10  ;
```

Tale metodo mostra l'interfaccia di INITIALIZE_GAME_OVER che consente ad un utente di cominciare una nuova partita o terminare il gioco. In quest'ultimo caso viene successivamente eseguito il metodo END_GAME.

5. File assemby

In questo capitolo vengono analizzati i file assembly della directory "Progetto > Assembly". Sono due e si occupano della stampa su schermo:

- print_pixel.s
- print_char.s

5.1 Print_pixel.s

Il file si occupa della stampa di un semplice rettangolo le cui caratteristiche sono state memorizzate in delle variabili in Forth.
Dati:

```
.data
    PIXEL:      .word 0
    PIXEL_SIZE: .word 1
    COLOR:      .word 2

```

L'assemblatore assegna a questi, in fase di compilazione, degli indirizzi in cui tramite Forth, vengono memorizzati i puntatori alle variabili contenenti i parametri di input. Questi vengono caricati nei registri nella prima parte del codice.

```
_start:
    push {ip, lr}

    ldr a1, =PIXEL
    ldr a1, [a1]
    ldr a2, =PIXEL_SIZE
    ldr a2, [a2]
    mov a3, a2
    ldr a4, =#0x00000FFF
    and a2, a2, a4
    ldr a4, =#0x00FFF000
    and a3, a3, a4
    lsr a3, #0xC
    ldr a4, =COLOR
    ldr a4, [a4]
    bl print_pixel

    pop {ip, lr}
    bx lr
```

Infine viene chiamata la subroutine print_pixel che stampa il rettangolo secondo i parametri scelti.

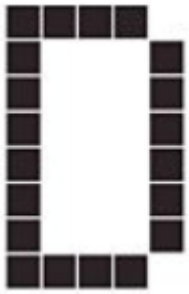
```
print_pixel:
    mov v1, a3
pp1:
    mov a3, v1
pp2:
    str a4, [a1], #0x4
    subs a3, a3, #0x1
    bne pp2

    mov a3, v1, lsl #0x2
    subs a1, a1, a3
    add a1, a1, #0x1000
    subs a2, a2, #0x1
    bne pp1

    bx lr
```

5.2 Print_char.s

In questo file vi sono le istruzioni Assembly per la stampa a schermo di un qualsiasi carattere rappresentato in una di cella 8x5 pixel.

Prima di vedere il codice è utile mostrare la logica utilizzata.

Si immagini di voler stampare la lettera D. Sotto la sua rappresentazione in una matrice 8x5:



Per stampare un carattere basta creare due cicli, uno per le righe ed uno per le colonne, e sapere se ogni pixel deve essere colorato o meno.

Il carattere viene rappresentato in memoria in 40 bit (8x5): ognuno assume valore 1 se il pixel corrispondente deve essere colorato o 0 se questo deve essere lasciato spento.

Sotto sono riportati i 40 bit che codificano la lettera D (little endian):
0111110001100011000110001100011000101111

In esadecimale: 7C6318C62F

Poiché è possibile caricare soltanto 32 bit alla volta sullo stack, prima viene eseguito il load degli 8 più significativi e poi degli altri 32. Qualsiasi carattere presente nel codice è stato codificato secondo questa logica.

Sotto tutte le codifiche dei caratteri.

```
\ ( -- b1 b2 )
: .A. 8C 7F18C62E ;
: .B. 7C 6317C62F ;
: .C. F0 4210843E ;
: .D. 7C 6318C62F ;
: .E. F8 4217843F ;
: .F. 08 4217843F ;
: .G. 74 6316843E ;
: .H. 8C 631FC631 ;
: .I. 71 0842108E ;
: .J. 32 5084211C ;
: .K. 8C 63149D31 ;
: .L. F8 42108421 ;
: .M. 8C 631AD771 ;
: .N. 8C 639ACE31 ;
: .O. 74 6318C62E ;
: .P. 08 42F8C62F ;
: .Q. 20 CA94A526 ;
: .R. 8A 4AF8C62F ;
: .S. 74 6107062E ;
: .T. 21 0842109F ;
: .U. 74 6318C631 ;
: .V. 21 14A54631 ;
: .W. 55 6B5AC631 ;
: .X. 8A 94422951 ;
: .Y. 21 08452A31 ;
: .Z. F8 4222221F ;
: .0. 74 6318C62E ;
: .1. F9 084210E4 ;
: .2. F8 4444462E ;
: .3. 74 6106422E ;
: .4. 42 3E952988 ;
: .5. 7C 2107843F ;
: .6. 74 6317862E ;
: .7. 21 0844421F ;
: .8. 74 6317462E ;
: .9. 74 610F462E ;
: .DP. 0 40008000 ;
: .SPACE. 0 0 ;
```

Come in print_pixel.s vengono create delle variabili che conterranno i puntatori ai dati passati in input.

```
.data
    PIXEL:      .word 0
    CHAR1:      .word 1
    CHAR2:      .word 2
    CHAR_SIZE:  .word 3
    COLOR:      .word 4
```

Le prime istruzioni caricano tali valori di input nei registri.

```
_start:
    push {v1, lr}

    ldr a1, =PIXEL
    ldr a1, [a1]
    ldr a2, =CHAR1
    ldr a2, [a2]
    ldr a3, =CHAR2
    ldr a3, [a3]
    ldr a4, =CHAR_SIZE
    ldr a4, [a4]
    ldr v1, =COLOR
    ldr v1, [v1]
    orr a4, a4, v1, lsl #0x8
    bl print_char

    pop {v1, lr}
    bx lr
```

Per rispettare lo standard ARM secondo il quale i valori passati tra le funzioni devono essere memorizzati nei registri a1 a2 a3 e a4, in a4 viene memorizzato sia il colore che la dimensione.

Caricati i valori viene chiamata la subroutine print_char.
Questa utilizza due cicli, uno itera per le righe, l'altro per le colonne e ad ogni iterazione, leggendo la codifica del carattere a partire dal bit meno significativo, colora (bit 1) o salta (bit 0) il pixel.

In realtà il metodo non lavora su un singolo pixel, ma fa uso di PRINT_PIXEL (spiegato precedentemente) per poter stampare un rettangolo. In questo modo è possibile gestire la dimensione di ogni singolo pixel che compone il carattere e, di conseguenza, la dimensione del carattere.

6. Adattamento

In questa sezione viene mostrato come adattare il codice al proprio ambiente.

6.1 Telecomando

In questo progetto è stato usato un telecomando del tipo mostrato sotto:

È necessario utilizzare un modello simile in quanto ha una particolare codifica di tasti che consente di sfruttare il bit di start di 9,1ms.

All'interno del file receiver.f è stato inserito il metodo SAMPLES che può essere utilizzato per leggere la sequenza di bit trasmessa da una qualsiasi sorgente infrarossi.

```
\ Campiona i bit del range selezionato
: SAMPLES                                        \ (     --     )
    0 SCOUNT
    0 SSAMPLE
    BEGIN  INPUT 1 <>  UNTIL
    GCLOCK STIME  2710 GRANGE *
    BEGIN  DUP  GCLOCK GTIME -  <  UNTIL
    DROP

    BEGIN
        INPUT
        DUP GSAMPLE <>  IF
            DUP SSAMPLE  INC_COUNT  GCLOCK GTIME -
        ELSE
            DROP
        THEN
    GCLOCK GTIME - 2710 GRANGE 1 + * >  UNTIL

    GCOUNT 2 * SCOUNT
    BEGIN
        DECIMAL . HEX
        GCOUNT 2 MOD 1 =  IF  CR  THEN
        GCOUNT 0 <>  IF  GCOUNT  1 - SCOUNT  THEN
    GCOUNT 0 = UNTIL
;
```

Prima di utilizzarlo è necessario collegare il ricevitore infrarossi come descritto nella sezione 1 e inizializzare la variabile RANGE:

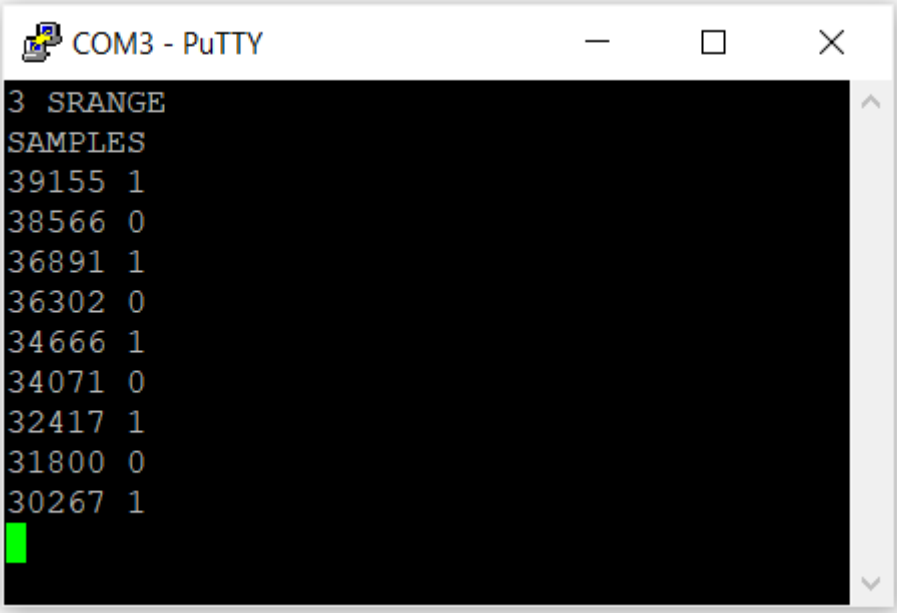
```
\ Contiene il range selezionato
\ per il campionamento
VARIABLE RANGE
: GRANGE RANGE @ ;                          \ (    -- b1 )
: SRANGE RANGE ! ;                          \ ( a1 --    )
```

Questa contiene un valore indicante il range all'interno del quale si vuole campionare. Ogni telecomando infatti trasmette bit per un certo intervallo di tempo. Per semplicità si è scelto di dividere gli intervalli in gruppi di 10ms.

RANGE 0: 0 -> 10 ms
RANGE 1: 10 -> 20 ms
RANGE 2: 20 -> 30 ms
…

Tornando al codice di SAMPLES, si attende l'arrivo dell'istante indicato dal range e da questo momento in poi legge i bit ricevuti dal segnale campionando gli instanti di tempo in cui varia.

ES. Usando un RANGE pari a 3 il terminale mostrata il seguente risultato quando viene ricevuto un segnale (in questo caso è stato premuto il pulsante 2):



Per fare in modo che il sistema riconosca i tasti premuti è necessario scegliere 3 istanti in cui campionare il segnale in modo che la combinazione di bit riscontrabile in questi istanti identifichi univocamente ogni singolo tasto.

Vediamo sotto, lo schema costruito tramite la funzione SAMPLES per il ritrovamento dei tre istanti di campionamento.

Intervalli

Pulsanti	50,0	50,3	50,9	51,5	52,0	52,6	53,1	53,7	54,2	54,8
2	0	1	0	1	0	1	1	1	0	
4	0	1	0	1	1	1	0	1	1	
6	0	1	1	1	1	1	0	1	0	
8	0	1	1	1	0	1	0	1	1	
PLAY	0	1	1	1	0	1	0	1	0	
			↑		↑				↑	

Le tre frecce indicano i tre intervalli dove campionare. Infatti in tali intervalli le codifiche dei tasti sono:

- 2: 000
- 4: 011
- 6: 110
- 8: 101
- PLAY: 100

Questi hanno tutti valori diversi che quindi identificano univocamente il tasto. Inoltre è utile non utilizzare la sequenza 111 che viene inviata dal telecomando quando si tiene premuto un pulsante.
Adesso è possibile modificare la funzione CONTROL_RECEIVER1 nel file receiver.f per poter consentire al sistema di riconoscere i segnali.

Una volta ottenuti gli istanti di campionamento decrementarli di 9.1ms (dovuto al modo in cui lavora il telecomando), convertire i risultati in esadecimale e creare un nuovo metodo il cui compito è lasciare sullo stack questi valori e le codifiche dei tasti, come visto nella trattazione del file receiver.f per i metodi ENRICO e GIULIANO. Ricordare che prima dell'avvio del gioco è necessario impostare questi valori attraverso il metodo SET_RECEIVER.