



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	32
4.3 MAIN.F	38
5. FILE ASSEMBY.....	42
5.1 PRINT_PIXEL.S.....	42
5.2 PRINT_CHAR.S.....	44
6. ADATTAMENTO.....	47
6.1 TELECOMANDO.....	47

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:

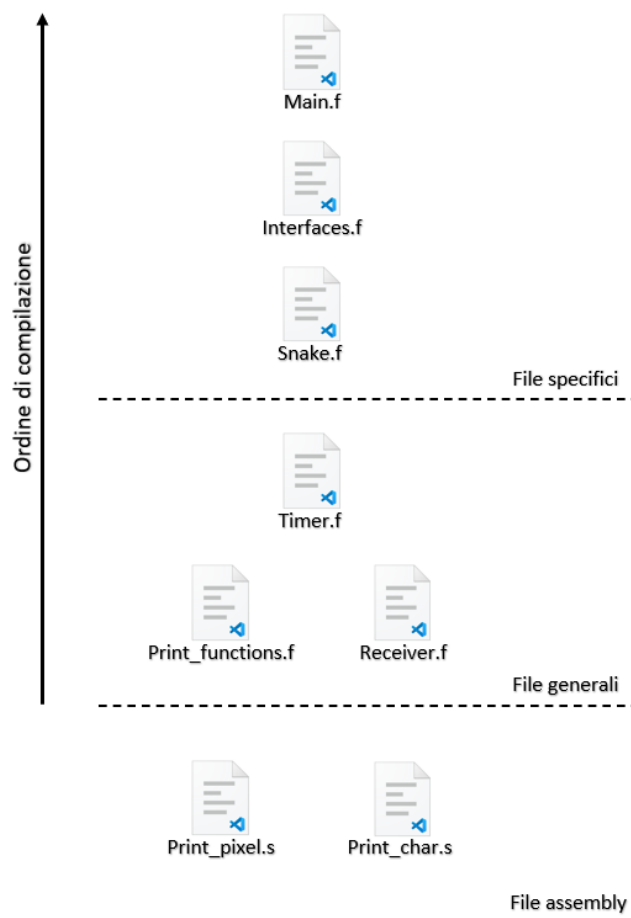
- Print_char.s
- Print_pixel.s

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.





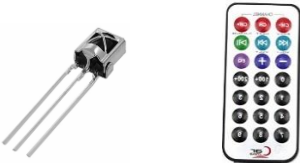


La foto sopra mostra la suddivisione dei file e l'ordine secondo il quale questi devono essere compilati.

2. Installazione

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

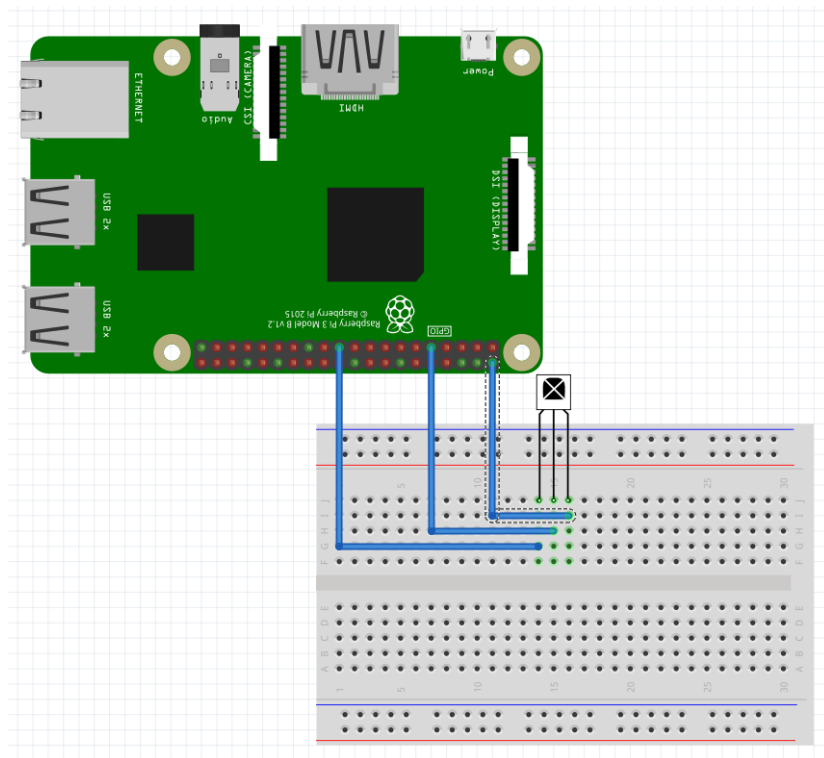
2.1 Hardware

Componenti:

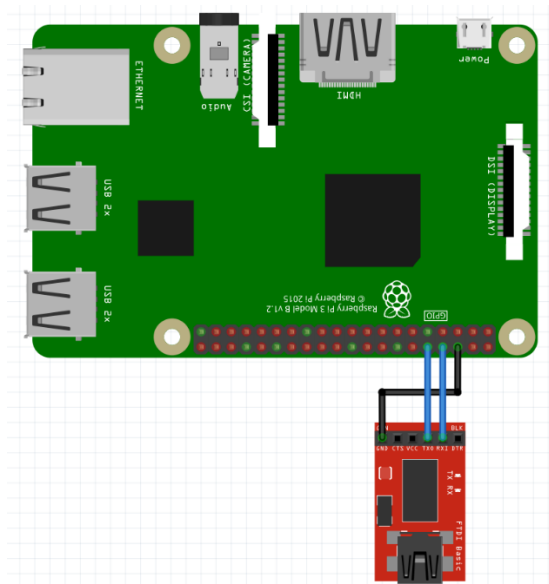
Raspberry Pi 3 B+	
FT232RL FTDI USB To TTL Serial Converter	
Ricevitore infrarossi VS1838B + telecomando infrarossi	
Micro SD	
Jumper	

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:

```
59 \ Subroutine in assembly per stampare a
60 \ schermo un rettangolo
61 CREATE PRINT_PIXEL_ASSEMBLY
62 e92d5000 , e59f0060 , e5900000 , e59f105c , e5911000 ,
63 e1a02001 , e59f3054 , e0011003 , e59f3050 , e0022003 ,
64 e1a02622 , e59f3048 , e5933000 , eb000001 , e8bd5000 ,
65 e12fff1e , e1a04002 , e1a02004 , e4803004 , e2522001 ,
66 1affffffc , e1a02104 , e0500002 , e2800a01 , e2511001 ,
67 1affffff6 , e12fff1e , 0001e4e0 , 0001e5f0 , 00000fff ,
68 00fff000 , 0001e598 ,
69
70 \ Interfaccia Forth di PRINT_PIXEL_ASSEMBLY
71 : 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**. 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

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

- **COLOR**: memorizza il colore del rettangolo

```
19 \ Contiene il valore rgb del colore
20 \ scelto per la stampa
21 VARIABLE COLOR
22 : GCOLOR COLOR @ ; \ ( -- b1 )
23 : 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_ASSEMBLY.

```
73 \ Subroutine in assembly per stampare a
74 \ schermo un carattere
75 CREATE PRINT_CHAR_ASSEMBLY
76 e92d4010 , e59f0144 , e5900000 , e59f1140 , e5911000 ,
77 e59f213c , e5922000 , e59f3138 , e5933000 , e59f4134 ,
78 e5944000 , e1833404 , eb000001 , e8bd4010 , e12fff1e ,
79 e92d0030 , e92d00c0 , e92d4100 , e1a04000 , e1a05001 ,
80 e1a06002 , e3a07000 , e3c380ff , e1a08428 , e20330ff ,
81 e3a02020 , e1a00006 , e3a01001 , e0000001 , e3500000 ,
82 0a000005 , e1a00004 , e1a01003 , e92d000c , e1a03008 ,
83 eb00001b , e8bd000c , e0844103 , e2422001 , e1a060a6 ,
84 e0820007 , e3a01005 , eb000022 , e3500002 , 1a000005 ,
85 e3a01014 , e0000391 , e0444000 , e3a01a01 , e0000391 ,
86 e0844000 , e1a00002 , e3500000 , 1affffe3 , e3570002 ,
87 0a000003 , e3a07002 , e3a02008 , e1a06005 , eaffffdd ,
88 e8bd4100 , e8bd00c0 , e8bd0030 , e12fff1e , e92d1010 ,
89 e1a02001 , e1a04002 , e1a02004 , e4803004 , e2522001 ,
90 1affffffc , e1a02104 , e0500002 , e2800a01 , e2511001 ,
91 1affffff6 , e8bd1010 , e12fff1e , e1500001 , ba000002 ,
92 e0400001 , e1500001 , aaffffffc , e12fff1e , 0001e4e0 ,
93 0001e6b8 , 0001e710 , 0001e654 , 0001e598 ,
94
95 \ Interfaccia Forth di PRINT_PIXEL_ASSEMBLY
96 : PRINT_CHAR SCHAR2 SCHAR1 PRINT_CHAR_ASSEMBLY JSR DROP ;
```

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.

```
30 \ Contiene la dimensione di un carattere
31 VARIABLE CHAR_SIZE
32 : GCHAR_SIZE CHAR_SIZE @ ; \ ( -- b1 )
33 : 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.

```
125 \ Stampa il numero in input
126 : PRINT_NUMBER \ ( a1 -- )
127 |   DUP SNUM_TEMP -1 SWAP
128 |   BEGIN
129 |     SWAP 1 + SWAP A /
130 |   DUP 0 = UNTIL
131 |   DROP
132 |   BEGIN
133 |     DUP A SWAP EXP GNUM_TEMP SWAP /
134 |     CODE_NUMBER PRINT_CHAR GCHAR_SIZE 6 * INC_PIXEL_0
135 |     SWAP DUP A SWAP EXP ROT * GNUM_TEMP SWAP - SNUM_TEMP 1 -
136 |   DUP -1 = UNTIL
137 |   DROP
138 ;
```

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.

```
98   \ Effettua l'elevazione a potenza
99   : EXP                                     \ ( a1 a2 -- b1 )
100   |   DUP 0 = IF DROP DROP 1 ELSE
101   |   DUP 1 = IF DROP           ELSE
102   |   |   SWAP DUP ROT
103   |   |   BEGIN
104   |   |   |   1 - ROT ROT SWAP DUP ROT * ROT
105   |   |   DUP 1 = UNTIL
106   |   |   DROP SWAP DROP
107   |   THEN THEN
108   ;
```

Esempio:

```
3 7 EXP
```

Lascia sullo stack il risultato di 3^7 .

PRINT_WORD

Stampa la parola presente nello stack.

```
140   \ Stampa le lettere in input
141   : PRINT_WORD                               \ ( a1 ... an an+1 -- )
142   |   DUP 6 * GCHAR_SIZE * INC_PIXEL_O
143   |   BEGIN
144   |   |   GCHAR_SIZE -6 * INC_PIXEL_O
145   |   |   ROT ROT PRINT_CHAR
146   |   |   1 -
147   |   DUP 0 = UNTIL
148   |   DROP
149   ;
```

Sotto la codifica delle lettere.

166	\ (-- b1 b2)	186	: .T. 21 0842109F ;
167	: .A. 8C 7F18C62E ;	187	: .U. 74 6318C631 ;
168	: .B. 7C 6317C62F ;	188	: .V. 21 14A54631 ;
169	: .C. F0 4210843E ;	189	: .W. 55 6B5AC631 ;
170	: .D. 7C 6318C62F ;	190	: .X. 8A 94422951 ;
171	: .E. F8 4217843F ;	191	: .Y. 21 08452A31 ;
172	: .F. 08 4217843F ;	192	: .Z. F8 4222221F ;
173	: .G. 74 6316843E ;	193	: .0. 74 6318C62E ;
174	: .H. 8C 631FC631 ;	194	: .1. F9 084210E4 ;
175	: .I. 71 0842108E ;	195	: .2. F8 4444462E ;
176	: .J. 32 5084211C ;	196	: .3. 74 6106422E ;
177	: .K. 8C 63149D31 ;	197	: .4. 42 3E952988 ;
178	: .L. F8 42108421 ;	198	: .5. 7C 2107843F ;
179	: .M. 8C 631AD771 ;	199	: .6. 74 6317862E ;
180	: .N. 8C 639ACE31 ;	200	: .7. 21 0844421F ;
181	: .O. 74 6318C62E ;	201	: .8. 74 6317462E ;
182	: .P. 08 42F8C62F ;	202	: .9. 74 610F462E ;
183	: .Q. 20 CA94A526 ;	203	: .DP. 0 40008000 ;
184	: .R. 8A 4AF8C62F ;	204	: .SPACE. 0 0 ;
185	: .S. 74 6107062E ;		

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.

IMPORTANTE!

Il file print_functions.f deve obbligatoriamente essere compilato per primo in quanto il codice in assembly agisce su determinati indirizzi di memoria.

3.2 Timer.f

Questo file mette a disposizione un timer che conta dai decimi di secondo fino ai minuti. Il suo limite è 99:99:9.

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.

```
7  \ Autoconfigurazione base_register
8  : AUTOCONFIG                                \ ( -- b1 )
9  | 3F003004 @
10 | 5 DROP
11 | 3F003004 @ = IF
12 | | FE000000
13 | ELSE
14 | | 3F000000
15 | THEN
16 ;
17 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

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

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

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

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

```

45  \ Contiene la dimensione del timer
46  VARIABLE TIMER_SIZE
47  : GTIMER_SIZE TIMER_SIZE @ ;
48  : STIMER_SIZE TIMER_SIZE ! ;

```

I metodi che possono essere richiamati sono SET_TIMER e CONTROL_TIMER.

SET_TIMER

Inizializza il contatore e lo stampa a schermo.

```

120  \ Inizializza il timer e lo stampa
121  : SET_TIMER
122  |   SET_MINUTES
123  |   SET_SECONDS
124  |   SET_DECSECONDS
125  |   GTIMER_COLOR SCOLOR
126  |   GTIMER_PIXEL SPIXEL
127  |   GTIMER_SIZE SCHAR_SIZE
128  |   GMINUTES     DUP PRINT_NUMBER PRINT_NUMBER .DP. PRINT_CHAR GCHAR_SIZE 2 * INC_PIXEL_O
129  |   GSECONDS     DUP PRINT_NUMBER PRINT_NUMBER .DP. PRINT_CHAR GCHAR_SIZE 2 * INC_PIXEL_O
130  |   GDECSECONDS  PRINT_NUMBER
131  ;

```

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.

```
133 \ Controlla se è il momento di
134 \ aggiornare il timer
135 ∨ : CONTROL_TIMER ∖ ( -- )
136     GCLOCK 186A0 /
137 ∨     DUP GTIMER <> IF
138         STIMER
139         GTIMER_COLOR SCOLOR
140         GTIMER_SIZE SCHAR_SIZE
141         UPDATE_DECSECONDS
142 ∨     ELSE
143         DROP
144     THEN
145 ;
```

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

```
94 \ Contiene i decimi di secondo
95 VARIABLE DECSECONDS
96 : GDECSECONDS DECSECONDS @ ; ∖ ( -- b1 )
97 : SET_DECSECONDS 0 DECSECONDS ! ; ∖ ( -- )
98 : UPDATE_DECSECONDS ∖ ( -- )
99     GDECSECONDS 1 +
100     DUP A = IF
101         SET_DECSECONDS
102         UPDATE_SECONDS
103         DROP GDECSECONDS
104     ELSE
105         DUP DECSECONDS !
106     THEN
107
108     GTIMER_PIXEL SPIXEL
109     1C GTIMER_SIZE * INC_PIXEL_0
110     BLACK SCOLOR
111     GCHAR_SIZE 8 * GCHAR_SIZE 5 * PRINT_PIXEL
112     WHITE SCOLOR PRINT_NUMBER
113 ;
```


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.

```

71  \ Contiene i secondi
72  VARIABLE SECONDS
73  : GSECONDS SECONDS @ ;           \ (    -- b1 )
74  : SET_SECONDS 0 SECONDS ! ;      \ (    --    )
75  : UPDATE_SECONDS                 \ (    --    )
76      GSECONDS 1 +
77  \ DUP 3C = IF
78      SET_SECONDS
79      UPDATE_MINUTES
80      DROP GSECONDS
81  ELSE
82      DUP SECONDS !
83  THEN
84
85      GTIMER_PIXEL SPIXEL
86      E GCHAR_SIZE * INC_PIXEL_O
87      BLACK SCOLOR
88      GCHAR_SIZE 8 * GCHAR_SIZE C * PRINT_PIXEL
89      WHITE SCOLOR
90      DUP A / 0 = IF 0 PRINT_NUMBER THEN
91      PRINT_NUMBER
92  ;

```

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

```

57  \ Contiene i minuti
58  VARIABLE MINUTES
59  : GMINUTES MINUTES @ ;           \ (    -- b1 )
60  : SET_MINUTES 0 MINUTES ! ;      \ (    --    )
61  : UPDATE_MINUTES                 \ (    --    )
62      GMINUTES 1 + DUP MINUTES !
63      GTIMER_PIXEL SPIXEL
64      BLACK SCOLOR
65      GCHAR_SIZE 8 * GCHAR_SIZE C * PRINT_PIXEL
66      WHITE SCOLOR
67      DUP A / 0 = IF 0 PRINT_NUMBER THEN
68      PRINT_NUMBER
69  ;

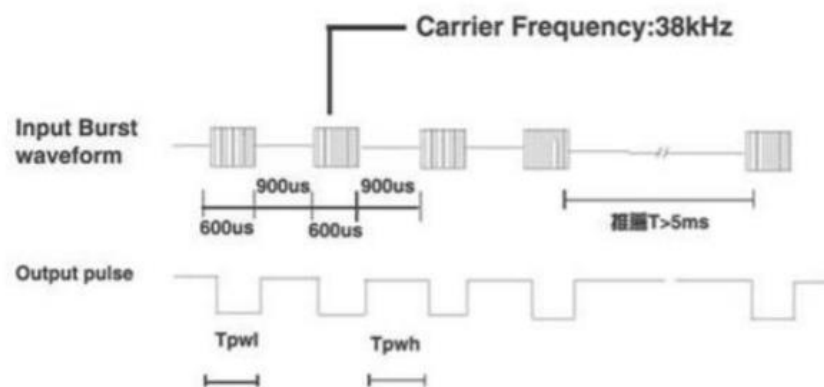
```

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.

```
23 \ Legge il bit del pin 9 (output del ricevitore)
24 BASE_REGISTER 200034 + CONSTANT GPLEV0
25 : INPUT GPLEV0 @ 400000 * 80000000 / ; \ ( -- b1 )
26
27 \ Indirizzo del registro dove sono
28 \ memorizzati i microsendi passati dall'avvio della macchina
29 BASE_REGISTER 3004 + CONSTANT CLO
30 : 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.

```
99  \ Caricano sullo stack i valori di campionamento
100 \ personali
101 : ENRICO                                \ (      -- b1 b2 b3 b4 b5 b6 b7 b8 )
102 |   A410  A85C  B0F4
103 |   0 3 4 2 5
104 | ;
105
106 : GIULIANO                             \ (      -- b1 b2 b3 b4 b5 b6 b7 b8 )
107 |   B284  BB99  BFFE
108 |   1 6 2 0 5
109 | ;
```

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

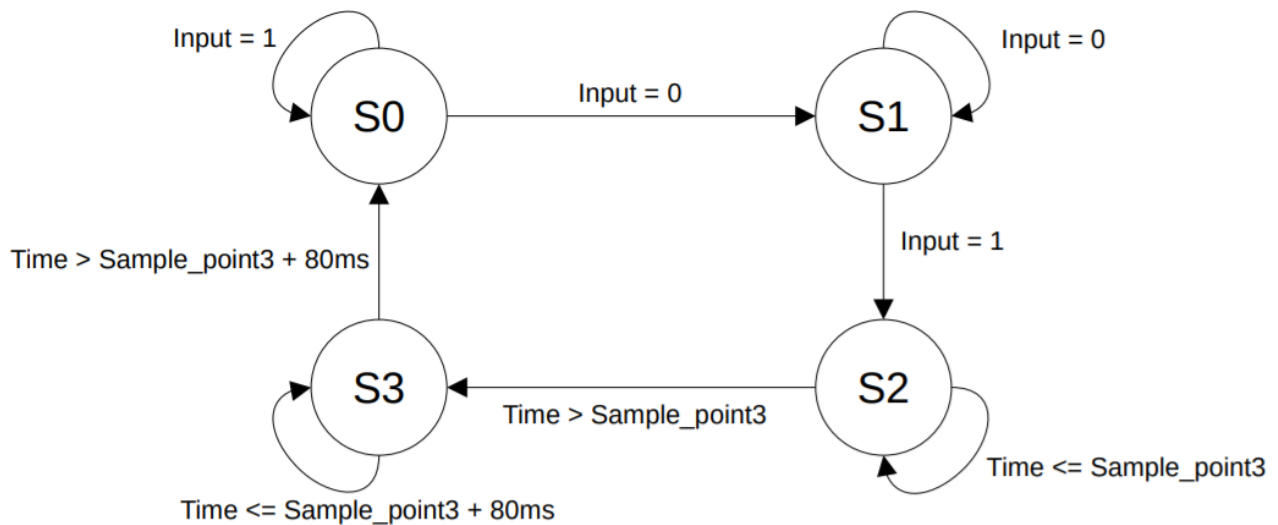
È il primo metodo che consente di leggere la sequenza di bit ricevuta.

```
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.

```

77 \ Contiene lo stato del metodo
78 \ CONTROL_RECEIVER
79 VARIABLE STATE_RECEIVER
80 : GSTATE_RECEIVER STATE_RECEIVER @ ; \ ( -- b1 )
81 : SSTATE_RECEIVER STATE_RECEIVER ! ; \ ( a1 -- )

```

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

```
120 \ Decodifica i bit ricevuti
121 : DECODE                                \ ( a1 a2 a3 -- b1 )
122     SWAP 2 * +
123     SWAP 4 * +
124     DUP GP2 = IF 2 ELSE
125     DUP GP4 = IF 4 ELSE
126     DUP GPP = IF 5 ELSE
127     DUP GP6 = IF 6 ELSE
128     DUP GP8 = IF 8 ELSE
129     | | | | -1
130     THEN THEN THEN THEN THEN
131     SWAP DROP
132 ;
```

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.

```
159 \ Interfaccia che sfrutta CONTROL_RECEIVER1
160 \ e blocca il flusso di esecuzione
161 : CONTROL_RECEIVER2                        \ ( -- )
162     1 SSTATE_RECEIVER
163     BEGIN INPUT 0 = UNTIL
164     BEGIN
165         CONTROL_RECEIVER1
166     GSTATE_RECEIVER 0 = UNTIL
167 ;
```

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.

```
88 \ Contiene il valore campionato
89 VARIABLE SAMPLE
90 : GSAMPLE SAMPLE @ ;                \ ( -- b1 )
91 : 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

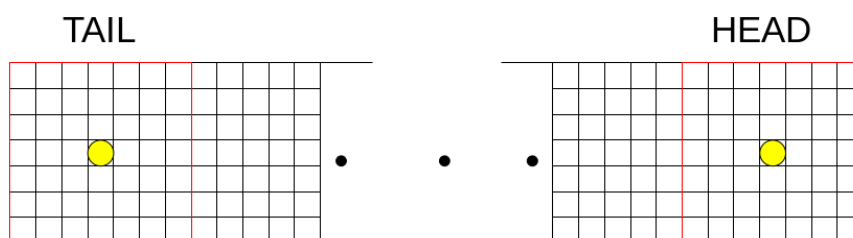
Nel file sono contenuti tutti i metodi di gestione del serpente.

Il primo ad essere eseguito è PRINT_SNAKE:

```
154 \ Stampa il serpente sullo schermo
155 : PRINT_SNAKE                                \ ( -- )
156     WHITE1 SCOLOR
157     3EAB9670 SPIXEL  7 C8 PRINT_PIXEL
158     3EABC980 SHEAD
159     3EABC67C STAIL
160     0 SDEAD
161     0 SSNAKE_DIR
162     0 SSTATE_APPLE
163 ;
```

Questo stampa a schermo un serpente di dimensione 7x200 pixel di colore WHITE1, ovvero una tonalità di bianco (0x00FFFFFFE) usata per distinguere il serpente dai bordi dello schermo.

Vengono inoltre inizializzate le variabili HEAD e 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

```

71  \ Incrementa orizzontalmente un indirizzo
72  \ del numero di pixel passati in input
73  : INC_O                                \ ( a1 a2 -- b1 )
74  |    4 * +
75  |    DUP  FFFFF000 AND
76  |    SWAP  00000FFF AND
77  |    DUP  FEC > IF FE0 - ELSE
78  |    DUP  010 < IF FE0 + THEN THEN +
79  ;

```

Il metodo effettua anche il controllo in modo che, quando un indirizzo incrementato ha oltrepassato i bordi sinistro o destro, questo viene riportato dall'altro 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

```

81  \ Incrementa verticalmente un indirizzo
82  \ del numero di pixel passati in input
83  : INC_V                                \ ( a1 a2 -- b1 )
84  |    1000 * +
85  |    DUP  FF000FFF AND
86  |    SWAP  00FFF000 AND
87  |    DUP  BF5000 > IF 278000 - ELSE
88  |    DUP  97E000 < IF 278000 + THEN THEN +
89  ;

```

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.

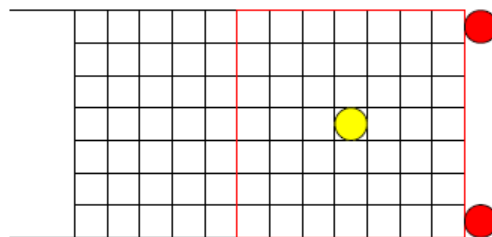
```

91 \ Controlla il colore dei pixel di fronte al serpente
92 : CHECK_FORWARD                                \ ( -- b1 )
93   GHEAD GHEAD_DIR
94   DUP 0 = IF DROP 4 INC_O DUP -3 INC_V @ SWAP 3 INC_V @ ELSE
95   DUP 1 = IF DROP 4 INC_V DUP -3 INC_O @ SWAP 3 INC_O @ ELSE
96   DUP 2 = IF DROP -4 INC_O DUP -3 INC_V @ SWAP 3 INC_V @ ELSE
97   | | | DROP -4 INC_V DUP -3 INC_O @ SWAP 3 INC_O @
98   THEN THEN THEN
99   GCOLOR = SWAP GCOLOR = OR
100 ;

```

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:

```

235 \ Svolge tutti i controlli necessari
236 \ e muove il serpente
237 : CONTROL_SNAKE                                \ ( -- )
238   CONTROL_DIRECTION
239   CHECK_APPLE
240   CHECK_DEAD
241   HEAD MOVE
242   TAIL MOVE
243 ;

```

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

CONTROL_DIRECTION

```

223 \ Controlla il valore contenuto nella variabile SAMPLE
224 \ e in base a questo modifica la direzione del serpente
225 : CONTROL_DIRECTION                                     \ ( -- )
226     GHEAD_DIR GSAMPLE
227     DUP 2 = IF DROP DUP 0 = SWAP 2 = OR IF 3 SHEAD_DIR THEN ELSE
228     DUP 4 = IF DROP DUP 1 = SWAP 3 = OR IF 2 SHEAD_DIR THEN ELSE
229     DUP 6 = IF DROP DUP 1 = SWAP 3 = OR IF 0 SHEAD_DIR THEN ELSE
230     DUP 8 = IF DROP DUP 0 = SWAP 2 = OR IF 1 SHEAD_DIR THEN ELSE
231     | | | 2DROP
232     THEN THEN THEN THEN
233 ;

```

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:

```

33 \ Contiene le direzioni di testa e coda
34 VARIABLE SNAKE_DIR
35 : GSNAKE_DIR SNAKE_DIR @ ;                                     \ ( -- b1 )
36 : SSNAKE_DIR SNAKE_DIR ! ;                                     \ ( a1 -- )
37
38 \ Consentono di accedere in maniera diretta
39 \ ai bit memorizzati nella variabile SNAKE_DIR
40 : GHEAD_DIR GSNAKE_DIR 3 AND ;                                 \ ( -- b1 )
41 : GTAIL_DIR GSNAKE_DIR C AND 4 / ;                             \ ( -- b1 )
42 : SHEAD_DIR GSNAKE_DIR C AND + SSNAKE_DIR ;                 \ ( a1 -- )
43 : 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

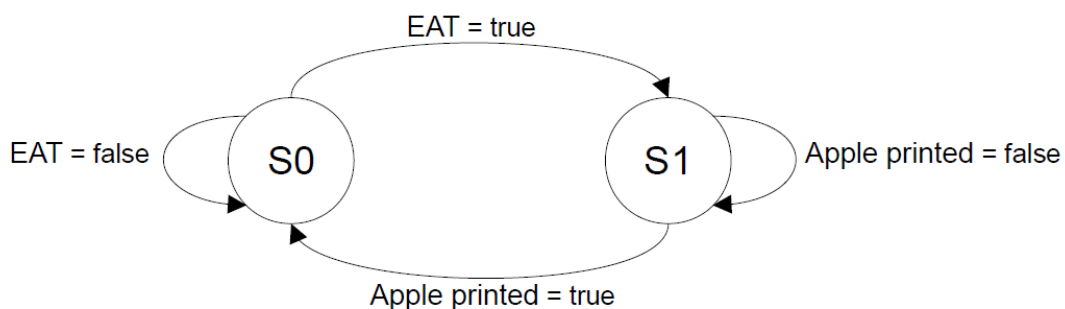
Nella variabile i 2 bit meno significativi contengono la direzione della testa, i 2 successivi quella della coda.

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:

```

137 \ In base al suo stato consente al serpente di
138 \ mangiare la mela e incrementare il punteggio
139 \ o stampa una nuova mela
140 : CHECK_APPLE                                \ ( -- )
141     GSTATE_APPLE 0 = IF
142     | RED SCOLOR CHECK_FORWARD IF
143     | | EAT UPDATE_SCORE 1 SSTATE_APPLE
144     | THEN
145     ELSE
146     | PRINT_APPLE SSTATE_APPLE
147     THEN
148 ;

```

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

```

123 \ Incrementa la lunghezza del serpente e cancella
124 \ la mela appena mangiata
125 : EAT GTAIL GTAIL_DIR                                \ ( -- )
126 | DUP 0 = IF DROP -7 INC_O DUP -3 INC_O -3 INC_V SPIXEL ELSE
127 | DUP 1 = IF DROP -7 INC_V DUP -3 INC_V -3 INC_O SPIXEL ELSE
128 | DUP 2 = IF DROP 7 INC_O DUP -4 INC_O -3 INC_V SPIXEL ELSE
129 | | | | DROP 7 INC_V DUP -4 INC_V -3 INC_O SPIXEL
130 THEN THEN THEN
131 STAIL
132 WHITE1 SCOLOR 7 7 PRINT_PIXEL
133
134 GAPPLE SPIXEL BLACK SCOLOR 7 7 PRINT_PIXEL
135 ;

```

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

```

46 \ Contiene il punteggio
47 VARIABLE SCORE
48 : GSCORE SCORE @ ;                                \ ( -- b1 )
49 : SET_SCORE 0 SCORE ! ;                            \ ( -- )
50 : UPDATE_SCORE                                     \ ( -- )
51 | 3E9142A8 SPIXEL BLACK SCOLOR GSCORE DUP PRINT_NUMBER
52 | 3E9142A8 SPIXEL WHITE SCOLOR 1 + DUP PRINT_NUMBER
53 | SCORE !
54 ;

```

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

```

107 \ Stampa la mela 7x7 in un punto casuale libero dello schermo
108 : PRINT_APPLE \ ( -- b1)
109 GCLOCK
110 BEGIN
111 GHEAD GCLOCK DUP INC_O -1 * INC_V BLR BHL - MOD BHL + DUP SAPPLE
112 DUP @ BLACK <> IF 0 ELSE
113 DUP 1C + @ BLACK <> IF 0 ELSE
114 DUP 7000 + DUP @ BLACK <> SWAP BLR > OR IF 0 ELSE
115 DUP 1C + 7000 + DUP @ BLACK <> SWAP BLR > OR IF 0 ELSE
116 | -1
117 THEN THEN THEN THEN
118 NIP DUP ROT GCLOCK SWAP - 2BC > OR UNTIL
119
120 IF GAPPLE SPIXEL RED SCOLOR 7 7 PRINT_PIXEL 0 ELSE 1 THEN
121 ;

```

CHECK_DEAD

```

165 \ Controlla se il serpente ha sbattuto su se stesso
166 : 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.

```

27 \ Contiene lo stato del serpente
28 \ 0 -> vivo , 1 -> morto
29 VARIABLE DEAD
30 : GDEAD DEAD @ ; \ ( -- b1 )
31 : 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.

```
168 \ Muove la testa o la coda del serpente stampando
169 \ delle righe o colonne bianche o nere
170 : MOVE                                     \ ( a1 a2 a3 a4 -- )
171     0 = IF SHEAD ELSE STAIL THEN
172     SWAP
173     DUP 0 = SWAP 2 = OR IF
174     -4 INC_V 7
175     BEGIN
176         SWAP 1 INC_V DUP GCOLOR SWAP !
177         SWAP 1 - DUP
178     0 = UNTIL DROP DROP
179 ELSE
180     -4 INC_O 7
181     BEGIN
182         SWAP 1 INC_O DUP GCOLOR SWAP !
183         SWAP 1 - DUP
184     0 = UNTIL DROP DROP
185 THEN
186 ;
```

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

```
188 \ Lascia sullo stack tutti i dati necessari a MOVE per
189 \ muovere la testa e stampare una riga o colonna bianca
190 : HEAD                                     \ ( -- b1 b2 b3 b4 )
191     GHEAD GHEAD_DIR
192     DUP 0 = IF SWAP 1 INC_O DUP 3 INC_O ELSE
193     DUP 1 = IF SWAP 1 INC_V DUP 3 INC_V ELSE
194     DUP 2 = IF SWAP -1 INC_O DUP -3 INC_O ELSE
195     | | | | SWAP -1 INC_V DUP -3 INC_V
196     THEN THEN THEN
197
198     WHITE1 SCOLOR SWAP 0
199 ;
```

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:

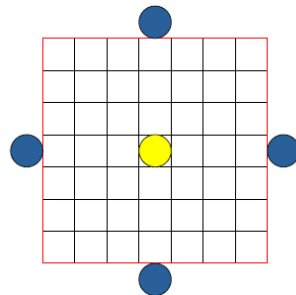
- la direzione verso la quale la testa si sta muovendo
- l'indirizzo a partire dal quale stampare una colonna o riga bianca di lunghezza 7 pixel in modo da simulare un movimento della testa
- la nuova posizione che avrà la testa dopo il movimento
- un flag per comunicare a MOVE di lavorare sulla testa

TAIL

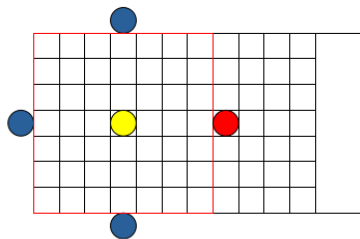
Prima di mostrare il codice è necessario spiegare la logica utilizzata per gestire il movimento.

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.



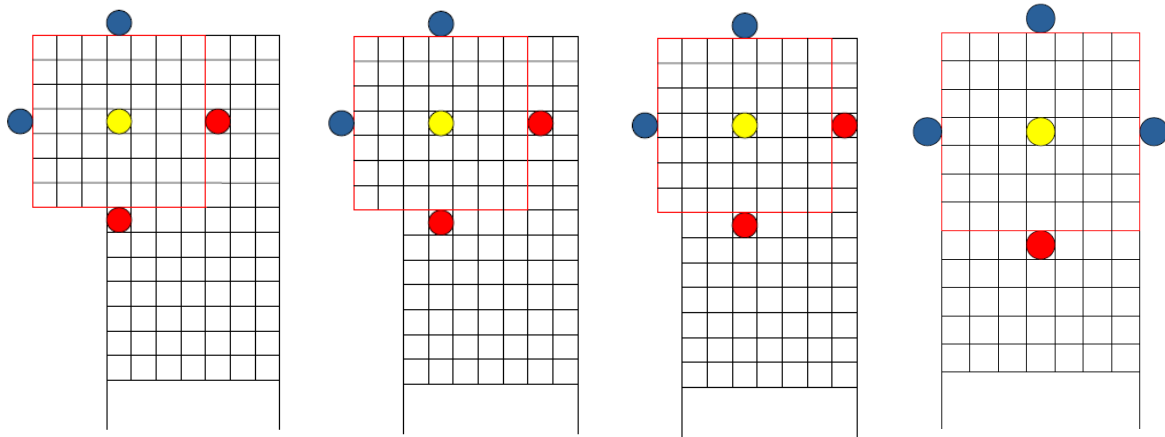
L'idea è quella di controllare, ad ogni passo, tutti i 4 pixel indicati dai puntini blu nello schema.



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:

```

201 \ Lascia sullo stack tutti i dati necessari a MOVE per
202 \ muovere la coda e stampare una riga o colonna nera
203 : TAIL \ ( -- b1 b2 b3 b4 )
204 0
205 GTAIL 4 INC_O @ WHITE1 = IF 1 + 0 SWAP THEN
206 GTAIL 4 INC_V @ WHITE1 = IF 1 + 1 SWAP THEN
207 GTAIL -4 INC_O @ WHITE1 = IF 1 + 2 SWAP THEN
208 GTAIL -4 INC_V @ WHITE1 = IF 1 + 3 SWAP THEN
209
210 2 = IF 2DROP GTAIL_DIR THEN
211
212 GTAIL SWAP
213 DUP 0 = IF SWAP DUP -3 INC_O SWAP 1 INC_O ELSE
214 DUP 1 = IF SWAP DUP -3 INC_V SWAP 1 INC_V ELSE
215 DUP 2 = IF SWAP DUP 3 INC_O SWAP -1 INC_O ELSE
216 | | | SWAP DUP 3 INC_V SWAP -1 INC_V
217 THEN THEN THEN
218
219 ROT DUP STAIL_DIR ROT ROT BLACK SCOLOR 1
220 ;

```

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 del gioco (ulteriori dettagli nel file main.f). Di default viene sempre selezionato il pulsante PLAY.

```
35 \ Stampa pulsanti PLAY ed EXIT
36 : PRINT_BUTTONS \ ( -- )
37 | 6 SCHAR_SIZE
38 | 3EB0C6EC SPIXEL .P. .L. .A. .Y. 4 PRINT_WORD
39 | 3EB766EC SPIXEL .E. .X. .I. .T. 4 PRINT_WORD
40 | SEL_PLAY
41 ;
```

Per selezionare i pulsanti si usano due ulteriori metodi:

- SEL_PLAY che crea un rettangolo intorno alla scritta PLAY

```
18 \ Crea un contorno al pulsante PLAY
19 : SEL_PLAY \ ( -- )
20 | 3EAF8674 SPIXEL 2 C0 PRINT_PIXEL
21 | 3EAF8674 SPIXEL 59 2 PRINT_PIXEL
22 | 3EB4F67C SPIXEL 2 C0 PRINT_PIXEL
23 | 3EAF8974 SPIXEL 59 2 PRINT_PIXEL
24 ;
```

- SEL_EXIT che crea un rettangolo attorno alla scritta EXIT

```

26  \ Crea un contorno al pulsante EXIT
27  : SEL_EXIT                                \ (    --    )
28      3EB62674 SPIXEL  2  C0 PRINT_PIXEL
29      3EB62674 SPIXEL  59 2  PRINT_PIXEL
30      3EBB967C SPIXEL  2  C0 PRINT_PIXEL
31      3EB62974 SPIXEL  59 2  PRINT_PIXEL
32  ;

```

Entrambi i metodi sopra indicati consentono di impostare il colore dall'esterno in modo che possano essere utilizzati per stampare e cancellare il bordo.

- **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.

```

43  \ Permette la selezione del pulsante PLAY o EXIT
44  : CONTROL_BUTTONS                        \ (    --    )
45      0
46      BEGIN
47          CONTROL_RECEIVER2 GSAMPLE
48          DUP 2 = IF
49              SWAP DUP 1 = IF
50                  1 - BLACK SCOLOR SEL_EXIT
51                  WHITE SCOLOR SEL_PLAY
52              THEN
53              SWAP
54          ELSE
55              DUP 8 = IF
56                  SWAP DUP 0 = IF
57                      1 + BLACK SCOLOR SEL_PLAY
58                      WHITE SCOLOR SEL_EXIT
59                  THEN
60                  SWAP
61              THEN
62          THEN
63          GCLOCK STIME
64          BEGIN 13880 GCLOCK GTIME - < UNTIL
65      5 = UNTIL
66  ;

```

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.

```

98  \ Blocca il flusso di esecuzione e attende il comando
99  \ per ricominciare
100 : PAUSE_INTERFACE                                \ (  --  )
101   3E9145E0 SPIXEL  WHITE SCOLOR  A SCHAR_SIZE
102   .P. .A. .U. .S. .E. 5 PRINT_WORD
103
104   BEGIN  CONTROL_RECEIVER2 GSAMPLE 5 = UNTIL
105
106   3E9145E0 SPIXEL  BLACK SCOLOR
107   .P. .A. .U. .S. .E. 5 PRINT_WORD
108
109   GHEAD_DIR
110   DUP 0 = IF 6 SSAMPLE ELSE
111   DUP 1 = IF 8 SSAMPLE ELSE
112   DUP 2 = IF 4 SSAMPLE ELSE
113   |   |   | 2 SSAMPLE
114   THEN THEN THEN
115   DROP
116 ;

```

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

```
68  \ Inizializzazione di SELECTION
69  : INITIALIZE_SELECTION                                \ (  --  )
70      CLEAR  WHITE SCOLOR  BORDER
71      18 SCHAR_SIZE  3E9B6290 SPIXEL
72      .S. .N. .A. .K. .E. 5 PRINT_WORD
73      PRINT_BUTTONS
74  ;
```

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:

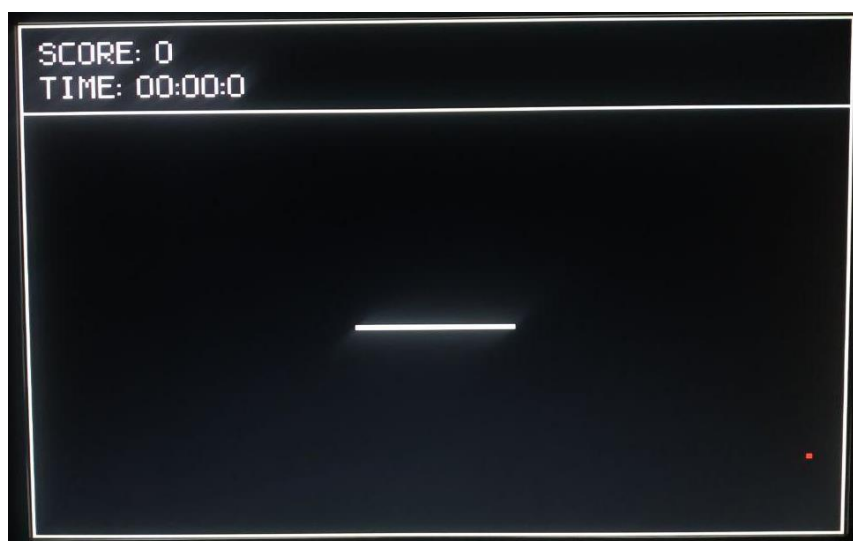


INITIALIZE_PLAY

```
76 \ Inizializzazione di PLAY \ ( -- )
77 : INITIALIZE_PLAY
78   CLEAR   WHITE SCOLOR   BORDER
79
80   3E97A000 SPIXEL  4 400 PRINT_PIXEL
81
82   4 SCHAR_SIZE
83   3E914068 SPIXEL  .S. .C. .O. .R. .E. .DP.  6 PRINT_WORD
84   3E9142A8 SPIXEL  SET_SCORE   0 PRINT_NUMBER
85
86   3E946068 SPIXEL  .T. .I. .M. .E. .DP.  5 PRINT_WORD
87   3E946248 STIMER_PIXEL
88   WHITE STIMER_COLOR  4 STIMER_SIZE   SET_TIMER
89
90   PRINT_SNAKE PRINT_APPLE
91
92   0 SSTATE_RECEIVER
93   BEGIN CONTROL_RECEIVER2 GSAMPLE 5 = UNTIL
94   6 SSAMPLE
95   SAVE_TIMER
96 ;
```

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.

```
118 \ Inizializzazione di GAME_OVER
119 : INITIALIZE_GAME_OVER \ ( -- )
120   CLEAR WHITE SCOLOR BORDER
121
122   3E973234 SPIXEL E SCHAR_SIZE
123   .G. .A. .M. .E. .SPACE. .O. .V. .E. .R. 9 PRINT_WORD
124
125   3EA38604 SPIXEL 4 SCHAR_SIZE .T. .I. .M. .E. .DP. 5 PRINT_WORD
126   3EA387E4 SPIXEL
127
128   GMINUTES DUP A / 0 = IF 0 PRINT_NUMBER THEN
129   PRINT_NUMBER .DP. PRINT_CHAR 8 INC_PIXEL_O
130
131   GSECONDS DUP A / 0 = IF 0 PRINT_NUMBER THEN
132   PRINT_NUMBER .DP. PRINT_CHAR 8 INC_PIXEL_O
133
134   GDECSECONDS PRINT_NUMBER
135
136   3EA7C604 SPIXEL .S. .C. .O. .R. .E. .DP. 6 PRINT_WORD
137   3EA7C844 SPIXEL GSCORE PRINT_NUMBER
138
139   PRINT_BUTTONS
140 ;
```

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.

```
42  \ Struttura GAME
43  : GAME                                \ (  --  )
44  |   SELECTION
45  |   0 = IF PLAY THEN
46  |   END_GAME
47  ;
```

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

Queste sono:

- SELECTION

```
12  \ Struttura SELECTION
13  : SELECTION                            \ (  -- b1 )
14  |   INITIALIZE_SELECTION
15  |   CONTROL_BUTTONS
16  ;
```

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.

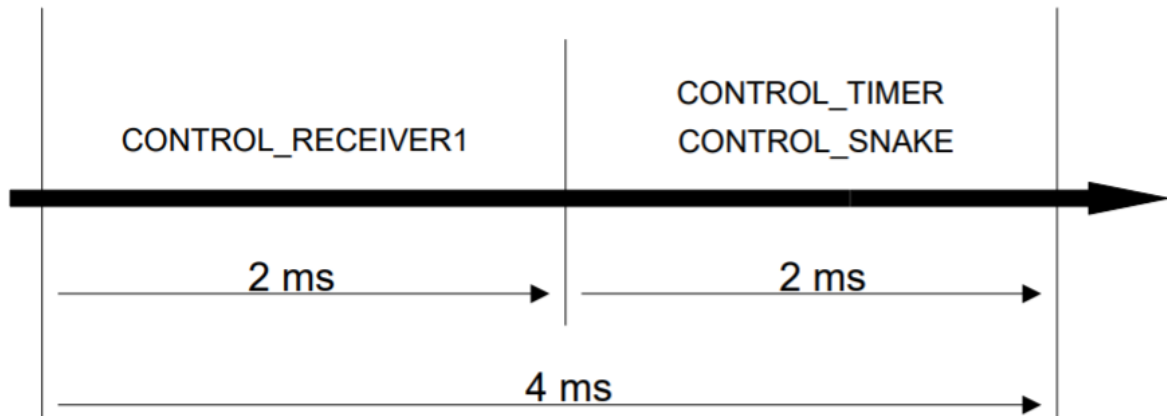
- END_GAME

```
37  \ Struttura END_GAME
38  : END_GAME                            \ (  --  )
39  |   CLEAR
40  ;
```

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 sfiorerà 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_RECEIVER1 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 assembly

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.

Attraverso i loro indirizzi è possibile risalire dal codice assembly alle variabili definite in forth. Sotto sono riportate le corrispondenze nel caso della normale esecuzione del gioco nel formato ASSEMBLY -> FORTH:

POS → PIXEL
DIM → PIXEL_SIZE
COL → COLOR

Gli indirizzi che delle variabili sono:

```
1  POS = 0x1E4E0
2  DIM = 0x1E5F0
3  COL = 0x1E598
```

A tali costanti viene assegnato un indirizzo fisso. Nel caso in cui il file print_functions.f non venisse compilato per primo gli indirizzi potrebbero variare.

Per quanto riguarda il codice, vengono inizialmente caricati i valori delle variabili nei registri.

La posizione:

```
9  | ldr a1, =POS
10 | ldr a1, [a1]
```

La dimensione:

```

11      |      ldr a2, =DIM
12      |      ldr a2, [a2]
13      |      mov a3, a2
14      |      ldr a4, =#0x00000FFF
15      |      and a2, a2, a4
16      |      ldr a4, =#0x00FFF000
17      |      and a3, a3, a4
18      |      lsr a3, #0xC

```

Nel registro a4 vengono memorizzate sia l'altezza, nei 12 bit meno significativi, che la larghezza, nei successivi 12.

Il colore:

```

19      |      ldr a4, =COL
20      |      ldr a4, [a4]

```

Fatto ciò viene chiamata la subroutine print_pixel.

```

21      |      bl print_pixel

```

Questa attraverso due cicli accende tutti i pixel indicati.

Ricordando che:

- a1 = posizione
- a2 = altezza
- a3 = larghezza
- a4 = colore

```

33      |      print_pixel:
34      |      mov v1, a3
35      |      pp1:
36      |      mov a3, v1
37      |      pp2:
38      |      str a4, [a1], #0x4
39      |      subs a3, a3, #0x1
40      |      bne pp2
41      |
42      |      mov a3, v1, lsl #0x2
43      |      subs a1, a1, a3
44      |      add a1, a1, #0x1000
45      |      subs a2, a2, #0x1
46      |      bne pp1
47      |
48      |      bx lr

```

Tale metodo è stato implementato in assembly per motivi di efficienza.

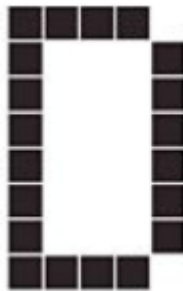
Utilizzando la stessa logica, ma con il linguaggio Forth, il tempo impiegato nella stampa sarebbe molto maggiore rispetto al corrispondente in Assembly.

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.

Immaginiamo 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.

```

166 \ ( -- b1 b2 )
167 : .A. 8C 7F18C62E ;
168 : .B. 7C 6317C62F ;
169 : .C. F0 4210843E ;
170 : .D. 7C 6318C62F ;
171 : .E. F8 4217843F ;
172 : .F. 08 4217843F ;
173 : .G. 74 6316843E ;
174 : .H. 8C 631FC631 ;
175 : .I. 71 0842108E ;
176 : .J. 32 5084211C ;
177 : .K. 8C 63149D31 ;
178 : .L. F8 42108421 ;
179 : .M. 8C 631AD771 ;
180 : .N. 8C 639ACE31 ;
181 : .O. 74 6318C62E ;
182 : .P. 08 42F8C62F ;
183 : .Q. 20 CA94A526 ;
184 : .R. 8A 4AF8C62F ;
185 : .S. 74 6107062E ;
186 : .T. 21 0842109F ;
187 : .U. 74 6318C631 ;
188 : .V. 21 14A54631 ;
189 : .W. 55 6B5AC631 ;
190 : .X. 8A 94422951 ;
191 : .Y. 21 08452A31 ;
192 : .Z. F8 4222221F ;
193 : .0. 74 6318C62E ;
194 : .1. F9 084210E4 ;
195 : .2. F8 4444462E ;
196 : .3. 74 6106422E ;
197 : .4. 42 3E952988 ;
198 : .5. 7C 2107843F ;
199 : .6. 74 6317862E ;
200 : .7. 21 0844421F ;
201 : .8. 74 6317462E ;
202 : .9. 74 610F462E ;
203 : .DP. 0 40008000 ;
204 : .SPACE. 0 0 ;

```

Analizziamo il codice.

Inizialmente vengono inseriti nei registri i valori memorizzati nelle variabili in Forth.

```

1  POS = 0x1E4E0
2  CHAR1 = 0x1E6B8
3  CHAR2 = 0x1E710
4  DIM = 0x1E654
5  COL = 0x1E598

```

La corrispondenza tra indirizzi Assembly -> Forth è:

POS	→	PIXEL
CHAR1	→	CHAR1
CHAR2	→	CHAR2
DIM	→	CHAR_SIZE
COL	→	COLOR

Il procedimento seguito è lo stesso di print_pixel.s.

```

12      ldr a1, =POS
13      ldr a1, [a1]
14      ldr a2, =CHAR1
15      ldr a2, [a2]
16      ldr a3, =CHAR2
17      ldr a3, [a3]
18      ldr a4, =DIM
19      ldr a4, [a4]
20      ldr v1, =COL
21      ldr v1, [v1]
22      orr a4, a4, v1, lsl #0x8

```

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.

```
192 \ Campiona i bit del range selezionato
193 : SAMPLES \ ( -- )
194   0 SCOUNT
195   0 SSAMPLE
196   BEGIN INPUT 1 <> UNTIL
197     GCLOCK STIME 2710 GRANGE *
198     BEGIN DUP GCLOCK GTIME - < UNTIL
199     DROP
200
201   BEGIN
202     INPUT
203     DUP GSAMPLE <> IF
204       DUP SSAMPLE INC_COUNT GCLOCK GTIME -
205     ELSE
206       DROP
207     THEN
208     GCLOCK GTIME - 2710 GRANGE 1 + * > UNTIL
209
210   GCOUNT 2 * SCOUNT
211   BEGIN
212     DECIMAL . HEX
213     GCOUNT 2 MOD 1 = IF CR THEN
214     GCOUNT 0 <> IF GCOUNT 1 - SCOUNT THEN
215     GCOUNT 0 = UNTIL
216 ;
```

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

```
186 \ Contiene il range selezionato
187 \ per il campionamento
188 VARIABLE RANGE
189 : GRANGE RANGE @ ; \ ( -- b1 )
190 : 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 istanti 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):



```
COM3 - PuTTY
3 SRANGE
SAMPLES
39155 1
38566 0
36891 1
36302 0
34666 1
34071 0
32417 1
31800 0
30267 1
█
```

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.

<

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.