Tutorial básico de criação de jogos para o Intellivision

Texto original: Carlos Madruga
Adaptação para PDF: Sergio Vares

Hello World passo a passo

Aprender qualquer linguagem sempre é mais fácil quando podemos observar exemplos.
Veja este programa, criado por Merrick J. Stemen, que participou do primeiro concurso de programação de SDK1600 no site Intellivision Library.
Ele é bastante simples: apenas gera uma tela inicial de apresentação, e responde com uma mensagem quando algum comando é acionado. Mas acho que ele bem didático.
Tentarei colocar comentários extras junto com o código. É claro que não sei o que significa tudo ainda (falta de tempo é sempre o meu problema), mas se você quiser complementar minhas explicações, fique a vontade !

```
;;-----------------------------------------------------------------------
;;MICKEY.ASM
;;-----------------------------------------------------------------------
;; by Merrick J. Stemen
;; 18 JANUARY 2000
;; This program does use EXEC calls.
;; Sorry it is not really a game, but it is a start!
;;
;; Compiled with the AS1600 compiler.
;;
;;-----------------------------------------------------------------------
;; Portions of the ROMHDR adapted from Joe Zbiciak's PONG
```

Observe que o compilador AS1600 entende como comentários tudo que você colocar depois de ; ou ;;
O que ele quis dizer com aquelas "EXEC calls" ?
O EXEC (executive) é um programa de 4K contido na ROM do Intellivision. O EXEC contém rotinas de movimentação de objetos na tela, manipulação de sons, etc.
Se quisermos usar rotinas do EXEC em nossos programas, devemos possuir, juntamente com um emulador, uma cópia deste EXEC. Este arquivo vem com o CD do Intellivision Lives, e se chama "Exec.bin".
Se você estiver usando o emulador Jzintv, Joe Zbiciak disponibilizou sua própria versão do EXEC ("miniexec.bin"), que é incompleta, mas pelo menos é freeware. (O EXEC é copyrighted.)
Não sei se este programa funciona com este mini-EXEC dele. Em seus programas, Joe tem feito suas próprias rotinas sempre, pra não ter que usar nenhuma do EXEC oficial.

```
        ROMW    10
        ORG     $5000
```

O label ROMW (rom width) indica que iremos trabalhar com um cartucho de 10 bits ! Pode parecer estranho, mas isso é normal no Intellivision. O Intellivision também pode usar RAMs/ROMs de 8, 12, 14 ou 16 bits.
Para efeito de nomenclatura, 10 bits são chamados "decles".
O ORG indica onde nosso programa começa na memória. Tipicamente, nossos programas podem ocupar 8Kb de RAM, localizados entre $5000 e $6FFF. (O "$" na frente de um número indica que ele está em hexadecimal.)
Se quiséssemos fazer um programa maior que 8Kb, poderíamos contar com um bloco de 4Kb entre $D000 e $DFFF e mais outro bloco de 4Kb entre $F000 e $FFFF. Para maiores detalhes sobre como criar programas maiores que 8Kb, veja o código do "Large Object File Demo" (large.asm) , que acompanha o SDK1600.

```
;;-----------------------------------------------------------------------
ROMHDR:
```

Todo programa deve incluir um cabeçalho, identificado pelo label ROMHDR. Este cabeçalho é sempre igual, e é usado pelo EXEC pra inicialização.
Obs.: este cabeçalho também é chamado de "Universal Data Block" (UDB)

```
      WORD    $0001           ; Movable Object Data Pointer
```

O STIC (Standard Television Interface Chip) é o chip que controla a parte gráfica do Intellivision. Uma das coisas que ele faz é gerenciar até 8 "moving objects" na tela. Este objeto é uma forma que é definida, e pode ser movimentada facilmente pela tela, por cima de uma imagem de background. Se você está familiarizado com o termo "sprite", é isso mesmo.
Esta linha indica o endereço dos "pictures databases" dos sprites, ou seja, onde começam as 8 (ou 16) decles que formam o desenho do sprite. O EXEC carrega este database na GRAM durante a inicialização.
De qualquer forma, não usamos sprites neste programa. Este database está vazio.

```
        WORD    $0000           ; RTAB
```

Este endereço aponta para uma rotina que é executada periodicamente.
A frequência a princípio é a cada 0.5 segundos. (Não sei se é possível alterar esta frequência.)

```
        WORD    START           ; START of program
```

Aponta para o início do programa de fato, também chamado de rotina RESET.
Esta rotina só será chamada após a execução do código que vem depois da descrição do título do cartucho (você entenderá melhor mais adiante).

```
        WORD    $0000           ; Background graphics pointer
```

Também chamado de "BKGDATA". Tal como a primeira linha do cabeçalho, este endereço aponta para o lugar onde estão definidos os formatos de imagens de fundo. Neste programa, não usamos isto.

```
        WORD    ROMHDR          ; CARDTAB
```

Card Table. A informação que eu tenho é que esta é uma lista que especifica quantas figuras do BKGDATA serão transferidas para a RAM, seguida de uma lista de números que indica quais figuras serão copiadas. O número de figuras não pode ser 0.
Então não captei por que ele colocou o ROMHDR aqui... será que ele viajou na maionese ?

```
        WORD    TITLE           ; Pointer to ASCIIZ Title String
```

Aponta para o título do cartucho (aquele nome que aparece depois de "Mattel Electronics presents...").

```
        BYTE    $80             ; Bit 7 SET -- Run code after title
```

Os bits 0 a 4 deste endereço controlam se haverá ou não som de "clicks" quando o disco/teclas/action keys do controle são acionados.
Se o bit 7 estiver setado, o código imediatamente após a definição do título do cartucho será executado.
O número 80 em hexadecimal equivale a 10000000 em binário, ou seja: "clicks off" e rodará o código depois do título.

```
         BYTE    $02             ; H/V offset
```

O que é preenchido nesta linha vai diretamente para o endereço $32 do STIC. Se o bit 0 deste endereço estiver setado (isto é, valendo 1), a primeira coluna de cards é bloqueada, ou seja, fica uma borda mais grossa à esquerda da tela. O bit 1 é a mesma coisa, só que relacionado à linha.
No caso deste programa, o valor colocado é 02, que equivale a 00000010. Ou seja, estamos fazendo um offset na tela de 1 linha.
Mas o que são "cards" ?
O STIC monta a tela usando 240 cards (20 largura x 12 altura), que são grupos de 8x8 pixels. Isto totaliza uma resolução de 160x96 pixels. Como na verdade o STIC não mostra a coluna à extrema direita (não sei ainda por que), a resolução de fato é 159x96. Cada pixel pode ter 16 cores (8 "primárias" e 8 "pastéis").

```
         BYTE    $00             ; Color Stack mode (0=CS; 1=F/B)
```

Este endereço é importante, pois indica qual será o esquema de cores a ser usado pelo STIC. (Cada card pode ter 2 cores).
O primeiro esquema é o "Foreground/Background" (F/B). Neste esquema, cada card pode ter suas cores determinadas individualmente, só que uma destas cores TEM QUE SER uma das 8 cores primárias. A outra pode ser uma primária ou pastel.
O segundo esquema é o "Color Stack Mode" (CS). Neste esquema, definimos uma pilha circular de 4 cores. Estas cores podem ser quaisquer dentre as primárias e pastéis. Aí a regra é seguinte: uma das cores de cada card deve ser a "cor da vez" dessa pilha, ou a próxima. Desse modo, poderíamos ter duas cores pastéis no mesmo card, o que é impossível pelo modo F/B.
Existe um terceiro esquema chamado "Colored Squares", muito pouco usado. Para informações, veja o "De re Intellivision, parte 2".
Este programa utiliza o modo CS de cores.

```
         BYTE    1, 1, 1, 1      ; Color Stack data (All blue)
```

Esta é a pilha de cores a que eu me referi. Aqui listamos as 4 cores que comporão a pilha. Se estivéssemos usando o modo F/B, esta linha poderia conter qualquer coisa, pois seria ignorada.
Os códigos das cores possíveis são:
Primárias: 0=Black, 1=Blue, 2=Red, 3=Tan, 4=Dark Green, 5=Green, 6=Yellow, 7=White
Pastéis: 8=Gray, 9=Cyan, 10=Orange, 11=Brown, 12=Pink, 13=Light Blue, 14=Yellow-Green, 15=Purple
Neste programa, o autor só colocou a cor Blue na pilha.

```
         BYTE    $02             ; Red Border
```

Cor da borda da tela. Neste programa, Red.
Neste ponto, o cabeçalho ROMHDR acaba.
(Na verdade, existe mais uma parte do cabeçalho, mas ela diz respeito a sprites, que não são usados aqui. Esta parte pode ser omitida então)

```
;;-------------------------------------------------------------------------
TITLE:        BYTE    100, "MICKEY'S GAME",0
```

O título do cartucho, indicado pelo lable TITLE.
O primeiro valor é o ano, que no caso deste programa foi chutado 100. Na verdade, o Intellivision sofre do bug do milênio, e o ano só vai até 1999. (O valor que você coloca aqui é acrescido de um "19" na frente na hora de mostrar na tela).
O truque então é escrever por cima deste ano, pra mascarar o problema.
Depois do ano vem o título ("MICKEY'S GAME"), e no final sempre vem o número 0.

```
;;-------------------------------------------------------------------------
;; Print "2000" where the copyright date goes
;; (I realize that I only need two zeros)
POST_TITLE:       MVII    #$2D0, R4       ; Where the copyright date goes
                  MVII    #$97, R1        ; Character "2" - white
                  MVO@    R1, R4          ; Put the "2" on the screen
```

Aqui começa o truque de escrever por cima da tela inicial pra contornar o problema do bug do milênio, e sobrescrever o nome "Mattel" (afinal, você não vai querer atribuir o copyright do seu programa à Mattel !).
Tudo isso faz parte do código que roda após o título, que está sendo executado conforme ordenado no cabeçalho.

A instrução MVII iguala o conteúdo de um registrador a um valor fixo.
Começamos igualando R4 a $2D0 (localização do caracter a ser escrito) e R1 a $97. Em seguida, usamos a instrução MVO@ ,que equivale a um POKE Rx, Ry em BASIC.
Mas porque igualamos R4 e R1 a estes valores ?
Vamos devagar. Primeiro sobre o valor de R4:
Os endereços na tela (chamados BACKTAB - background cards) começam em $200 e vão até $2F0, correspondendo a 240 cards em sequência. Lembrando, são 20 cards por linha. Então se fizermos as contas, o endereço BACKTAB $2D0 equivale a 10ª linha, 8ª coluna.
Agora sobre o valor de R1:
Como estamos usando o modo CS de cores, um card é definido da seguinte maneira:

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
valor	?	?	a	cF	t	n	n	n	n	n	n	n	n	f	f	f
$97 =	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1

Onde: c = se setado, e o bit 11 valer 0, aciona o modo Colored Squares para este card
a = se setado, avança para a próxima cor de fundo na Color Stack
Ffff = cor de foreground. Pode ser primária ou pastel.
t = se valer 0, indica que o card aponta pra GROM. Caso contrário, aponta pra GRAM.
nnnnnnnn = posição do card na GROM (0-212) ou GRAM (0-63)

A GROM conta com mais caracteres que a GRAM. Na GROM podemos usar caracteres minúsculos, pontuação e caracteres gráficos. No nosso programa, vemos que o valor $97 equivale ao card 18 na GROM, que é o número "2".
Com isso, colocamos o caracter "2" (cuja cor é White - ver bits 0,1, 2 e 12) no endereço BACKTAB $2D0. (Sempre lembrando que nosso objetivo aqui é escrever o ano "2000" por cima da tela de apresentação.)
Uma característica da instrução MVO@ é que quando usada com um "Auto-Incrementing Data Counter" (registradores R4 e R5), estes Data Counters são automaticamente incrementados. Isso significa que já estamos posicionados pra escrever os próximos carateres em sequência.
(Para maiores detalhes sobre instruções, veja o excelente documento "Intro to CP1600", que vem com o SDK1600)

```
MVII    #$87, R1        ; Character "0" - white
MVO@    R1, R4          ; "0"
MVO@    R1, R4          ; "0"
MVO@    R1, R4          ; "0"
```

Em seguida, carregamos R1 com o caracter "0". Não precisamos carregar R4 com a localização $2D1, pois a instrução MVO@ anterior já fez isso.
Só precisamos então executar 3 vezes a instrução MVO@ R1,R4 pra completar a impressão de "2000" na tela.

```
MVO@    R5, R6          ; Push R5
INCR    R4
JSR     R5, $187B
BYTE    "Mickey", 0     ; Put my name over last Mattel :-)
MVI@    R6, R7          ; Return from Title Screen
```

O próximo passo foi colocar o nome "Mickey" sobre o da Mattel. Para isso, o autor resolveu experimentar uma subrotina do EXEC para impressão de uma palavra inteira (ao invés de ficar imprimindo caracter por caracter).
É por isso que eu acho este programa didático :-)

Antes disso, salvamos o conteúdo de R5 no registrador R6 com a instrução MVO@ R5,R6 , pois esse valor será perdido quando chamarmos a tal subrotina de impressão de palavras. (A instrução JSR chama essa subrotina, mas gravará por cima o endereço de retorno dela em R5).
Mas por que guardar o conteúdo inicial de R5 em R6 é importante ?
Na verdade, quando executamos o código após o título, tudo é encarado como uma subrotina, cujo endereço de retorno fica armazenado em R5 (é como se o EXEC tivesse dado um JSR R5, POST_TITLE)
Guardando este endereço inicial de R5 em R6, poderemos depois voltar esse endereço de R6 para o Program Counter (R7) e permitir que o programa continue normalmente.
Também incrementamos o valor de R4 com a instrução INCR, pra dar um espaço entre "2000" e "Mickey".

A subrotina de impressão funciona do seguinte modo:
1- colocamos uma instrução JSR para chamá-la. Como já disse, essa instrução armazena o endereço de retorno em R5.
2- uma string deve estar localizada sempre após a instrução JSR, contendo a palavra a ser impressa, seguida de um 0.
3- o registrador R4 deve conter o endereço BACKTAB a ser escrito na tela
4- o registrador R3 deve conter o código da cor dos caracteres.Como neste programa o autor não atribuiu nenhum valor ainda a R3, suponho que a cor default deva ser White.
5- o registrador R7 (Program Counter) é auto-incrementado após a execução da subrotina.

Depois da chamada desta rotina, executamos a instrução MVI@ R6,R7. Se você se lembra bem, R6 continha o valor de R5 quando a rotina POST_TITLE começou a ser executada. Ou seja, R5 continha o endereço de retorno de POST_TITLE.
Se transferirmos este valor para o R7 (Program Counter), o programa fará um desvio para esse endereço de retorno.
E o que acontece depois ? Você saberia dizer ?
Sim, você acertou, a rotina RESET (isto é, o programa principal) é chamada pelo EXEC.
(Você não acertou ? Não tem problema. Agora você já sabe. :-)

```
;;-------------------------------------------------------------------------
START:          MVO@    R5, R6          ; Push R5
                MVII    #$0200, R4       ; Start of ScreenMem
                MVII    #240, R0         ; To clear entire screen
                JSR     R5, $1738        ; CLEARMEM
```

Aqui começa o programa de fato, que é o que vem depois da tela de título.
A primeira coisa feita foi salvar o valor de R5 em R6. O raciocínio é análogo ao da rotina POST_TITLE.
Em seguida, chamamos uma outra subrotina do EXEC que serve pra limpar a tela.
Essa rotina funciona do seguinte modo:
1- R4 deve conter o endereço inicial do trecho que será limpo.
2- R0 deve conter o número de cards a serem limpos. No caso, o valor 240 usado equivale a toda a tela. Note que 240 está sem "$" na frente, então estamos nos referindo a um valor decimal agora.

```
                MVII    #$027C, R4       ; Where to print the message
                MVII    #$0004, R3       ; Color to print the message
                JSR     R5, $187B        ; Print Data that follows
                BYTE    "HELLO,WORLD!", 0
                MVII    #$02DE, R4       ; Where to print the message
                MVII    #$0000, R3       ; Color to print the message
                JSR     R5, $187B        ; Print the data that follows
                BYTE    "PRESS F10 OR ESC", 0
                MVI@    R6, R7           ; IS THIS THE END??
;;-------------------------------------------------------------------------
```

Finalmente, após limpar a tela, iremos imprimir duas mensagens.
Para isso, usaremos aquela velha subrotina do EXEC situada em $187B.
A última instrução do programa é a transferência do valor contido no endereço de memória apontado por R6 para o Program Counter R7. Isto finaliza o programa.

O que você achou deste primeiro tutorial ?
Achou tudo fácil ? Ficou apavorado ? Não entendeu nada ?
Mande suas dúvidas !
Não garanto que responderei tudo, mas se você leu tudo o que escrevi até aqui, e pelo menos ficou com uma coceirinha de fazer um programa só seu, então meu esforço não terá sido em vão.

Algumas sugestões de exercícios:

1 - Mude a frase da tela de título "Mattel Electronics presents" para "<seu nome> presents"
2 - Mude a cor de fundo de azul para preto
3 - Elimine o offset das bordas

Usando sprites

Neste tutorial, você verá que criar e animar um sprite no Intellivision é uma moleza !
Na verdade, devemos agradecer ao STIC, que funciona de maneira quase "independente" da CPU. Ele gerencia gráficos e animações de maneira muito simples, e explica porque temos jogos tão bons para o Intellivision .
Eu criei este programa para fazer uma demonstração disso.

A seguir, vou tentar comentar o código o máximo possível para que você não tenha a sensação de que eu tirei coisas "da cartola".
Novamente, gostaria de deixar claro que a documentação sobre isso é escassa, e estou me baseando grandemente no que outros programadores já descobriram.
Se você tiver algum comentário a fazer, por favor mande-me um e-mail. Vamos aprender juntos !
Este tutorial pressupõe que você já viu e entendeu o anterior (Hello Word).

```
;;-------------------------------------------------------------------------
;; CEMSPRITE.ASM
;;-------------------------------------------------------------------------
;; by Carlos Madruga (aka Helixx)
;; 29 APRIL 2000
;; This program does use EXEC calls.
;;
;; Compiled with the AS1600 compiler.;;
;; This simple program shows how to define and move a single sprite.
;; Feel free to modify and experiment with it !
;;
;;-------------------------------------------------------------------------
;; Created based on:
;; - Pong by Joe Zbiciak
;; - SPRDEMO.SRC by Doug Parsons
;; - RUNNING.SCR by Carl Mueller
;; - EGGS.SCR by Scott Nudds

        ROMW    10
        ORG     $5000
```

Até aqui, nada novo. Apenas comentários, definição de largura da ROM e endereço de início do programa.

```
;
; constants
;

XSPEED      EQU 30
YSPEED      EQU 0
CENTRX      EQU $5800
CENTRY      EQU $3800
XPOS_INIT   EQU CENTRX
YPOS_INIT   EQU CENTRY
```

Aqui definimos algumas constantes a serem usadas no programa.
(A instrução EQU atribui o valor à variável.)
Lembrando, valores que começam com o "$" estão em hexadecimal.

As variáveis XSPEED e YSPEED servirão para especificarmos ao STIC a velocidade do sprite.
As variáveis CENTRX e CENTRY indicam as coordenadas do centro da tela.
As variáveis XPOS_INIT e YPOS_INIT são as coordenadas iniciais do sprite.

```
;;-----------------------------------------------------------------------
ROMHDR:
        WORD    SPRITES         ; Movable Object Data Pointer
        WORD    $0000           ; RTAB
        WORD    START           ; START of program
        WORD    $0000           ; Background graphics pointer
        WORD    ROMHDR          ; CARDTAB
        WORD    TITLE           ; Pointer to ASCIIZ Title String
        BYTE    $82             ; Bit 7 SET -- Run code after title
        BYTE    $00             ; H/V offset
        BYTE    $00             ; Color Stack mode (0=CS; 1=F/B)
        BYTE    0, 0, 0, 0      ; Color Stack data
        BYTE    $00             ; Black Border
   ; List of "address offsets" for sprite "picture spaces" in GRAM:
   WORD   $180, $190, $1A0, $1B0, $1C0, $1D0, $1E0, $1F0
```

Este é o velho cabeçalho inicial do programa (Universal Data Block). Nada de novo, exceto a primeira e a última linha.
A primeira linha aponta onde estão os desenhos dos sprites na memória. Eles estão mais pra frente no programa, indicados pelo label SPRITES.

Agora vamos explicar a última linha:
Cada sprite possui um "picture space" na GRAM. Este "picture space" é justamente a própria imagem dele.
Estas imagens ficam alocadas no final da GRAM ($3800 a $38FF).
A GRAM pode alocar dentro dela até 64 imagens 8 x 8 (picture spaces de 8 bytes). Estas imagens podem ser usadas não só para sprites, mas também para imagens de background (veremos estas últimas num próximo tutorial).
Existem sprites de 16 bytes (double resolution). A utilização deles afetaria esta alocação e diminuiria aquele limite de 64 imagens.
No nosso programa, usamos apenas 1 sprite de 8 bytes, então aquela última linha do cabeçalho poderia ser somente "WORD $180".
(Por que coloquei 8 offsets então ? Porque o STIC só suporta até 8 sprites simultaneamente, então com estes offsets você poderia criar seus próprios programas usando até 8 sprites de 8 bytes.)

```
;;-----------------------------------------------------------------------
TITLE:      BYTE    100, "ANIMATION TEST", 0

;;-----------------------------------------------------------------------
POST_TITLE:
        MVO@    R5, R6          ; Push R5
        MVII    #$2D0, R4       ; Where the copyright date goes
        JSR     R5, $187B  ; call print routine
        BYTE    "2000 Madruga", 0   ; Overwrite initial screen
        MVI@    R6, R7          ; Return from Title Screen
```

Neste trecho nada novo.
Usei a rotina de impressão do EXEC ($187B) para corrigir a tela de apresentação.

```
;;-----------------------------------------------------------------------
START:
        MVO@    R5, R6          ; Push R5
   MVII    #$0200, R4       ; Start of ScreenMem
        MVII    #240, R0        ; To clear entire screen
        JSR     R5, $1738       ; CLEARMEM
```

Limpo a tela usando a rotina CLEARMEM do EXEC.

```
        MVII    #$31d,R5   ; STIC sprite data port
        SDBD
        MVII    #SPRTAB,R4      ; sprite ram database
```

A seguir copiaremos alguns dados sobre a definição de sprites para o endereço $31D da RAM.

Estes dados estão todos em SPRTAB , mas pra você não ficar boiando por enquanto, eles são os seguintes:
- sprite status word
- ponteiro para a collision table
- posição x inicial
- posição y inicial
- sprite motion word
- sprite sequence word

Cada um dos dados mencionados será descrito com mais detalhes lá na frente, mas por enquanto é preciso que você entenda que eles simplesmente devem ser transferidos para o endereço $31d para o sprite ser definido.
Mas porque devemos copiar para este endereço específico ?
A resposta é que o trecho $31D a $35A da RAM é exclusivamente destinado aos Moving Object (sprite) Records.

A posição destes records é a seguinte:
- sprite status word: $31D + (n*8)
- ponteiro para a collision table: $31E + (n*8)
- posição x inicial: $31F + (n*8)
- posição y inicial: $320 + (n*8)
- sprite motion word: $321 + (n*8)
- sprite sequence word: $322 + (n*8)
... onde n é o número do sprite (0-7). O STIC só suporta 8 sprites !

Existe mais um dado chamado "sprite time-out counter word", que fica em $323 + (n*8). Não copiaremos este dado, pois ele não será usado. Ele será apenas zerado (maiores detalhes mais adiante).

Finalmente, você deve ter observado a instrução SDBD. Pra que ela serve ?
No Intellivison a maioria dos programas é armazenada em ROMs de 10 bits de largura. Isto causa um problema, pois o processador CP-1600 manipula 16 bits e 8 bits, e não 10.
(As ROMs de 10 bits foram usadas pois eram mais baratas !)
A solução então é usar a instrução SDBD ("Set Double Byte Data"), que permite a manipulação de um valor de 16 bits em dois de 8 bits. O SDBD indica que a próxima instrução acessa "Double Byte Data". Para maiores detalhes, leia "Introduction to CP1600 Programming", que acompanha o SDK1600.
Teoricamente, o assembler AS1600 deveria inserir estas instruções automaticamente pra facilitar nossas vidas, mas as vezes ele falha. A documentação dele menciona que ele por default não coloca SDBD quando fazemos referências a código que vem lá na frente (como no caso da referência SPRTAB).
Poderíamos fazer com que o assembler fizesse isso declarando no começo ROMW 10,1.

O segundo parâmetro (modo 1), diz ao assembler para inserir sempre um SDBD em referências futuras, independente de ser necessário ou não. Eu testei isso e não funcionou, então acho que existem bugs.
Na dúvida, coloquei SDBD manualmente em tudo.
Ah, a instrução MVII é uma instrução de modo "imediato", o que é atestado pelo último "I" do nome.
Instruções de modo imediato manipulam constantes diretamente. Portanto MVII armazena uma constante em um registrador.

```
        SDBD
        mvi@ R4,R1
        mvo@ R1,R5    ; copy sprite status word
        SDBD
        mvi@ R4,R1
        mvo@ R1,R5    ; copy pointer to collision table
        SDBD
        mvi@ R4,R1
        mvo@ R1,R5    ;  copy initial sprite x pos
        SDBD
        mvi@ R4,R1
        mvo@ R1,R5    ; copy initial sprite y pos
        SDBD
        mvi@ R4,R1
        mvo@ R1,R5    ; copy sprite motion word
        SDBD
        mvi@ R4,R1
        mvo@ R1,R5    ; copy sprite sequence word
        clrr R0
        mvo@ R0,R5    ; set sprite timer
        MVI@ R6, R7   ; THE END
```

Aqui nós copiamos os dados para os endereços corretos na RAM.
A instrução MVI@ R4,R1 armazena o conteúdo do endereço de memória apontado por R4 (isto é, dados dos sprites) em R1. R4 é incrementado automaticamente, já permitindo a leitura do próximo valor.
A instrução MVO@ R1,R5 armazena o valor contido em R1 em R5. R5 é incrementado automaticamente, permitindo a escrita do próximo valor.
Instruções que terminam com "@ " são instruções de modo "indireto", ou seja, só acessam valores na memória que são apontados por registradores.
A instrução CLRR R0 zera o registrador R0. Com isso, a instrução MVO@ R0,R5 zera a "sprite time-out counter word" já mencionada anteriormente.
Esta "sprite time-out counter word" é um timer, e pode ser usado pra disparar uma rotina quando chegar a 0. (Mais detalhes adiante, na explicação da COLLTAB.)

```
;;-------------------------------------------------------------------------
; Sprite (moving object) bitmaps
```

```
;
SPRITES:
; running man frame #0
 BYTE $000C ;             OO
 BYTE $0008 ;             O
 BYTE $001C ;            OOO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0078 ;          OOOO
 BYTE $0050 ;          O O
 BYTE $0010 ;            O
 BYTE $0018 ;            OO
 ; running man frame #1
 BYTE $000C ;             OO
 BYTE $0008 ;             O
 BYTE $003C ;           OOOO
 BYTE $0078 ;          OOOO
 BYTE $0078 ;          OOOO
 BYTE $007C ;          OOOOO
 BYTE $0030 ;           OO
 BYTE $0038 ;           OOO
 BYTE $007C ;          OOOOO
 BYTE $007C ;          OOOOO
 BYTE $0044 ;          O   O
 BYTE $005C ;          O OOO
 BYTE $0050 ;          O O
 BYTE $0040 ;          O
 BYTE $0060 ;          OO
 BYTE $0020 ;           O
 ; running man frame #2
 BYTE $000C ;             OO
 BYTE $0008 ;             O
 BYTE $003C ;           OOOO
 BYTE $00F8 ;         OOOOO
 BYTE $00BC ;         O OOOO
 BYTE $0080 ;         O
 BYTE $0030 ;           OO
 BYTE $0038 ;           OOO
 BYTE $007C ;          OOOOO
 BYTE $007C ;          OOOOO
 BYTE $0044 ;          O   O
 BYTE $004C ;          O  OO
 BYTE $004C ;          O  OO
 BYTE $00C8 ;         OO  O
 BYTE $0080 ;         O
 BYTE $0080 ;         O
 ; running man frame #3
 BYTE $0000 ;
 BYTE $000C ;             OO
 BYTE $0008 ;             O
 BYTE $003C ;           OOOO
 BYTE $0078 ;          OOOO
 BYTE $0058 ;          O OO
 BYTE $005E ;          O OOOO
 BYTE $0000 ;
 BYTE $0018 ;            OO
 BYTE $003F ;           OOOOOO
 BYTE $0021 ;           O    O
 BYTE $0061 ;          OO    O
 BYTE $0043 ;          O    OO
 BYTE $00C2 ;         OO    O
 BYTE $0080 ;         O
 BYTE $0000 ;
 ; running man frame #4
 BYTE $0000 ;
 BYTE $0006 ;              OO
 BYTE $0004 ;              O
 BYTE $007E ;          OOOOOO
 BYTE $005C ;          O OOO
 BYTE $005F ;          O OOOOO
 BYTE $0018 ;            OO
 BYTE $0000 ;
 BYTE $0018 ;            OO
```

```
 BYTE $003F ;           OOOOOO
 BYTE $003F ;           OOOOOO
 BYTE $0021 ;           O    O
 BYTE $0061 ;          OO    O
 BYTE $0061 ;          OO    O
 BYTE $00C0 ;         OO
 BYTE $0080 ;         O
 ; running man frame #5
 BYTE $0006 ;              OO
 BYTE $0004 ;              O
 BYTE $003E ;           OOOOO
 BYTE $007C ;          OOOOO
 BYTE $005C ;          O OOO
 BYTE $005F ;          O OOOOO
 BYTE $0000 ;
 BYTE $0018 ;            OO
 BYTE $0038 ;           OOO
 BYTE $003C ;           OOOO
 BYTE $002C ;           O OO
 BYTE $006C ;          OO OO
 BYTE $00C6 ;         OO   OO
 BYTE $0086 ;         O    OO
 BYTE $0002 ;               O
 BYTE $0003 ;               OO
 ; running man frame #6
 BYTE $0000 ;
 BYTE $0003 ;               OO
 BYTE $0002 ;               O
 BYTE $001F ;            OOOOO
 BYTE $003E ;           OOOOO
 BYTE $007E ;          OOOOOO
 BYTE $005F ;          O OOOOO
 BYTE $0040 ;          O
 BYTE $0018 ;            OO
 BYTE $0038 ;           OOO
 BYTE $003C ;           OOOO
 BYTE $003C ;           OOOO
 BYTE $0024 ;           O  O
 BYTE $00E4 ;         OOO  O
 BYTE $0084 ;         O    O
 BYTE $0006 ;              OO
 ; running man frame #7
 BYTE $0006 ;              OO
 BYTE $0004 ;              O
 BYTE $001E ;            OOOO
 BYTE $003C ;           OOOO
 BYTE $003C ;           OOOO
 BYTE $003E ;           OOOOO
 BYTE $0038 ;           OOO
 BYTE $0038 ;           OOO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0018 ;            OO
 BYTE $0078 ;          OOOO
 BYTE $0048 ;          O  O
 BYTE $000C ;             OO
```

Aqui colocamos os desenhos do sprite na memória. Note que apesar de termos vários desenhos, eu disse "sprite" e não "spriteS".
Isso porque colocamos várias figuras que formarão a animação de um único sprite.
Lembrando, a indicação para estes desenhos é feita na primeira linha de ROMHDR.

```
;;-----------------------------------------------------------------------
; sprite RAM databases
;
SPRTAB:

   word 0011100101001001b ; sprite status word
   word COLLTAB   ; pointer to collision table
   word XPOS_INIT   ; initial sprite x pos
   word YPOS_INIT   ; initial sprite y pos
   byte YSPEED,XSPEED  ; sprite motion word
   word 0000001001110000b ; sprite sequence word
```

Aqui começa a parte realmente importante do programa.

Vamos as definições:
Sprite Status Word (também chamado de Sprite Attribute Word):

bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
valor	u	u	dx	v	c	my	mx	q	d	r	b	f	g	f	f	f
usado	0	0	1	1	0	0	0	1	0	1	0	0	1	0	0	1

onde:

u = não usado
dx = tamanho duplo na direção X (quando vale 1)
v = invisível (0) / visível (1)
c = collision detection ON (1) / OFF (0)
my = espelhamento no eixo y ON (1) / OFF (0)
mx = espelhamento no eixo x ON (1) / OFF (0)
q = altura quadruplicada ON (1) / OFF (0)
d = altura duplicada ON (1) / OFF (0)
r = resolução 16 bits (1) / 8 bits (0)
b = background priority: na frente do background (0) / atrás (1)
g = imagem da GROM (0) / GRAM (1)
f = cor do sprite

A segunda linha indica a COLLTAB. Ela será descrita mais pra frente.
A próxima linha é a coordenada X inicial do sprite.

bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
valor	i	i	i	i	i	i	i	i	f	f	f	f	f	f	f	f
usado	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0

onde:

i = parte inteira da coordenada X
f = parte fracionária da coordenada X

A coordenada Y inicial também é descrita deste modo.,

bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
valor	i	i	i	i	i	i	i	i	f	f	f	f	f	f	f	f
usado	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0

A velocidade dos sprites são descritas por 2 bytes. Atenção para o fato de que a velocidade Y vem primeiro !
Finalmente, temos a sprite sequence word:

bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
valor	d	d	d	d	d	d	s	s	s	s	s	s	n	n	n	n
usado	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0

onde:

n = número do frame atual da animação (0-15)
s = velocidade da sequência (0 - 63) << só preencha isto !!!
d = contador (usado internamente pelo EXEC)

```
;;-----------------------------------------------------------------------
; Collision Table
;
COLLTAB:

   BYTE 0,0    ; off playing-field dispatch
```

Finalmente, temos a Collision Table. Tudo o que vemos aqui é apenas o cabeçalho desta tabela.
Na primeira linha, o primeiro "0" é a parte menos significativa, e o segundo "0" é a parte mais significativa de um dado cujos bits 8 e 9 correspondem ao tipo de colisão a ser detectado:

0 = intersecção com o "screen frame"
1 = ??? Não achei documentação.
2 = fora dos limites físicos da tela (qualquer lugar fora da região de 160 x 92 da tela)
3 = fora dos limites virtuais da tela (definidos nos endereços $116 - $119). (Não tenho mais detalhes...)

```
BYTE 0,0    ; time-out dispatch
```

Você se lembra que a definição de um sprite envolvia um timer, que no nosso caso foi zerado ?
Pois é, na segunda linha, temos o endereço da rotina a ser chamada quando este timer do sprite chega a 0. O byte menos significativo do endereço vem primeiro.

```
BYTE 8,0    ; sequence definition  (# of pictures in sequence,base picture)
```

A terceira linha tem dois bytes. O primeiro indica o número de figuras na animação do sprite, e o segundo indica qual é a primeira figura da sequência.

```
BYTE 0      ; interaction dispatches (1st decle is the number of dispatches defined)
```

A quarta linha possui apenas um byte. Este byte está relacionado a entradas na Collision Table. Não usamos tais entradas neste programa.
O programa acaba aqui...

Se você estiver curioso sobre entradas na Collision Table, continue lendo. Vou tentar explicar um pouco sobre elas.
Voltando ao byte da quarta linha, os bits 1,2 e 3 formam o número de entradas nesta Collision Table. Podemos ter até 8 entradas nesta tabela, correspondendo ao número máximo de sprites definidos (lembre-se sempre: o STIC só suporta um máximo de 8 sprites).
Se o bit 0 estiver setado, a primeira entrada da Collision Table vai apontar para uma rotina a ser acionada em caso de colisão com o background. (Hmmm... "colisão com o background" ?!? Preciso testar isso.)

Mas afinal, como são definidas as entradas da Collision Table ? Vou explicar a teoria, pois eu ainda não testei.
Cada entrada é composta de 2 words de 10 bits cada (vulgas "decles").

Primeira word:

```
09 08 07 06 05 04 03 02 01 00
 1  2  l  l  l  l  l  l  l  l
```

Segunda word:

```
09 08 07 06 05 04 03 02 01 00
 x  0  h  h  h  h  h  h  h  h
```

onde:

hhhhhhhhllllllll	= endereço a ser chamado se a colisão ocorrer
210	= número do sprite que colidiu com este. (Provavelmente ignorado quando estivermos lidando com colisões com o background.)
x	= não usado

Em resumo, com a Collision Table podemos definir detalhadamente o que acontece com cada sprite ao colidir com outros. Cada possibilidade de colisão pode chamar rotinas diferentes, conforme as entradas mostradas anteriormente..
Também podemos definir o que acontece quando cada sprite colide com as bordas da tela (físicas ou virtuais).

... Simples e poderoso, não ? :-)

Algumas sugestões de exercícios:

1 - Faça o homem andar de costas !
2 - Mude a cor do homem.
3 - Faça o homem "patinar no gelo" ! (Passos lentos mas com grande deslocamento X)

Usando imagens de fundo

Neste tutorial, você verá como definir imagens de fundo para seus programas. Se você entendeu o tutorial anterior, este será fácil !
Eu criei este programa para fazer uma demonstração disso.

Imagens de fundo ("background images") são figuras 8x8 bits que são carregadas na GRAM (Graphics RAM).
A GRAM possui espaço para 64 imagens 8x8, que podem ser ocupadas tanto por Moving Objects ("sprites"), como por background images. Ou seja, se seu programa definir 4 Moving Objects por exemplo, você poderá contar com até 60 background images.
Costumamos colocar os Moving Objects no final da GRAM, daí os "address offsets" que você verá depois do cabeçalho inicial do programa.
A seguir vou comentar o código de meu programa. Ele realmente é muito simples, mas servirá pra você pegar os conceitos.

```
;;-----------------------------------------------------------------------
;;BACKG.ASM
;;-----------------------------------------------------------------------
;; by Carlos Madruga (aka Helixx)
;; 06 OCTOBER 2000
;; This program does use EXEC calls.
```

```
;;
;; Compiled with the AS1600 compiler.
;;
;; This simple program shows how to define and move a single sprite,
;; with a simple background.
;; Feel free to modify and experiment with it !
;;
;;-----------------------------------------------------------------------
;; Created based on:
;; - Pong by Joe Zbiciak
;; - SPRDEMO.SRC by Doug Parsons
;; - RUNNING.SCR by Carl Mueller
;; - EGGS.SCR by Scott Nudds
        ROMW    10
        ORG     $5000
;
; constants
;

XSPEED      EQU 30
YSPEED      EQU 0
CENTRX      EQU $5800
CENTRY      EQU $3800
XPOS_INIT   EQU CENTRX
YPOS_INIT   EQU CENTRY
```

Estas constantes funcionam de modo idêntico ao do tutorial anterior (sobre sprites). A diferença desta vez é que usamos um sprite de Dragão ao invés de um Running Man...

```
;;-----------------------------------------------------------------------
ROMHDR:

        WORD    SPRITES         ; Movable Object Data Pointer
        WORD    $0000           ; RTAB
        WORD    START           ; START of program
        WORD    bkgdata         ; Background graphics pointer
        WORD    cardtab         ; CARDTAB
        WORD    TITLE           ; Pointer to ASCIIZ Title String
        BYTE    $82             ; Bit 7 SET -- Run code after title
        BYTE    $00             ; H/V offset
        BYTE    $00             ; Color Stack mode (0=CS; 1=F/B)
        BYTE    0, 4, 0, 0      ; Color Stack data
        BYTE    $00             ; Black Border
   ; List of "address offsets" for sprite "picture spaces" in GRAM:

   WORD   $180, $190, $1A0, $1B0, $1C0, $1D0, $1E0, $1F0
```

Este é o bom e velho cabeçalho (UDB).
Note na quarta linha, que contém um apontador para o endereço BKGDATA. É lá que está armazenada nossa imagem de fundo.
Na quinta linha temos um apontador para o endereço CARDTAB. Este endereço terá informações sobre o número de imagens de fundo a serem usadas.

```
cardtab:
        word       1, $001, $100                ; load 1 card
```

O CARDTAB indica quantas imagens de fundo serão carregadas na GRAM. O primeiro número indica 1 imagem.
Os outros dois eu não estou bem certo do significado ainda. Já vi programas usando 3 imagens de fundo, mas com os mesmos números $001 e $100 depois.

```
;;-----------------------------------------------------------------------
TITLE:        BYTE    100, "BACKGROUND TEST", 0

;;-----------------------------------------------------------------------
POST_TITLE:
        MVO@    R5, R6              ; Push R5
        MVII    #$2D0, R4           ; Where the copyright date goes
        JSR     R5, $187B           ; call print routine
        BYTE    "2000 Madruga", 0   ; Overwrite initial screen
        MVI@    R6, R7              ; Return from Title Screen
```

Aqui montamos a tela de título, usando uma rotina de impressão para sobrescrever a parte de ano e copyright.

```
;;-----------------------------------------------------------------------
START:
        MVO@    R5, R6          ; Push R5
        MVII    #$0200,R4       ; Fills memory with value in R1
        MVII    #$0804,R1       ; R4=starting address
        MVII    #240,R0         ; R1=value
        JSR     R5, $1741       ; R0=count
```

Aqui usamos uma rotina do EXEC destinada a encher a tela com o valor $0804, que significa o card 0 da GRAM, com a cor dark green. Veja o primeiro

tutorial (Hello World) para lembrar como é a definição de um card no modo Color Stack.
O card 0 da GRAM está ocupado pela nossa imagem de fundo, então o resultado é que encheremos a tela com ela. Lembrando, a tela tem 240 cards (valor de R0), começando pelo endereço $0200 (valor de R4).

```
        MVII    #$22A, R4       ; Where the message goes
        MVII    #6,R3   ; color
        JSR     R5, $187B  ; call print routine
        BYTE    "Keep Inty Alive !", 0
```

Imprimimos uma mensagem na posição $22A da tela...

```
        MVII    #$31d,R5        ; STIC sprite data port
        SDBD
        MVII    #SPRTAB,R4      ; sprite ram database

        SDBD
        mvi@ R4,R1
        mvo@ R1,R5    ; copy sprite status word
        SDBD

        mvi@ R4,R1
        mvo@ R1,R5    ; copy pointer to collision table
        SDBD

        mvi@ R4,R1
        mvo@ R1,R5    ;  copy initial sprite x pos
        SDBD

        mvi@ R4,R1
        mvo@ R1,R5    ; copy initial sprite y pos
        SDBD

        mvi@ R4,R1
        mvo@ R1,R5    ; copy sprite motion word
        SDBD

        mvi@ R4,R1
        mvo@ R1,R5    ; copy sprite sequence word

        clrr R0
        mvo@ R0,R5    ; set sprite timer

        MVI@    R6, R7           ; THE END
```

Esta parte de definição do sprite é idêntica ao tutorial anterior. Copiamos os dados do sprite para os endereços a partir de $35d.

```
;;-------------------------------------------------------------------------
; Sprite (moving object) bitmaps
;

SPRITES:

; dragon frame 1

 byte 00001110b
 byte 00011011b
 byte 00011100b
 byte 00110010b
 byte 00111000b
 byte 00111000b
 byte 00111100b
 byte 01110011b
 byte 01111001b
 byte 01111010b
 byte 00111000b
 byte 00111100b
 byte 01101100b
 byte 11001100b
 byte 01100110b
 byte 00000000b

; dragon frame 2

 byte 00001110b
 byte 00011011b
 byte 00011111b
 byte 00110000b
 byte 00111000b
 byte 00111000b
```

```
 byte 00111100b
 byte 01110011b
 byte 01111001b
 byte 01111010b
 byte 00111000b
 byte 00111000b
 byte 00111000b
 byte 00110000b
 byte 00111000b
 byte 00000000b
```

Estes são os frames do Dragão !

```
;;-----------------------------------------------------------------------
; sprite RAM databases
;

SPRTAB:

   word 0011000011011111b ; sprite status word
   word COLLTAB           ; pointer to collision table
   word XPOS_INIT         ; initial sprite x pos
   word YPOS_INIT         ; initial sprite y pos
   byte YSPEED,XSPEED     ; sprite motion word
   word 0000001001110000b ; sprite sequence word


;;-----------------------------------------------------------------------
; Collision Table
;

COLLTAB:

   BYTE 0,0    ; off playing-field dispatch
   BYTE 0,0    ; time-out dispatch
   BYTE 2,0    ; sequence definition  (# of pictures in sequence,base picture)
   BYTE 0      ; interaction dispatches (1st decle is the number of dispatches defined)
```

Aqui definimos o sprite do mesmo modo que no tutorial anterior, com a diferença de que agora temos apenas 2 frames na sequência do sprite, e que nosso sprite agora tem uma cor diferente.

```
; Background Cards bitmaps

bkgdata:

 byte 01000000b
      byte 11101000b
      byte 00011100b
      byte 00111110b
      byte 01011111b
      byte 11100000b
      byte 01110010b
      byte 00000111b
```

Esta é finalmente a imagem de fundo !

Fácil, não ? Agora é só treinar !

Algumas sugestões de exercícios:

1 - Faça um programa que use uma composição de imagens de fundo. Por exemplo, uma imagem de fundo de 16x8 (ou seja, duas imagens 8x8 diferentes lado a lado em sequência). Você precisará bolar um loop !

Acessando os controles do Intellivision

Este tutorial mostra como receber os sinais de entrada dos controles do Intellivision, e como direcioná-los para disparar as ações que você quiser.

Mas antes de comentarmos sobre o programa propriamente dito, vamos a uma descrição das especificações dos controles do Intellivision. Desse jeito ficará mais fácil entender o resto.

Os controles do Intellivision são formados pelo amado (ou seria mais odiado ?) disco , que suporta 16 posições distintas. Também temos um keypad de 12 teclas (0 a 9, enter, clear), e quatro botões laterais (também chamados de "action buttons").

Talvez você não tenha percebido, mas os dois botões laterais superiores são ligados ! Então na verdade tanto faz apertar o direito ou esquerdo, e temos na verdade 3 "action buttons" no total.
O EXEC automaticamente lê os valores fornecidos pelos controles e os armazena nos registradores R0 e R1.
Você também pode informar ao EXEC para direcionar cada sinal recebido para uma subrotina distinta. Por exemplo, em um jogo você pode especificar que o disco ativará a subrotina que move uma nave espacial para os lados, e os botões laterais ativarão subrotinas que dispararão os tiros dela.
Para tanto, você deverá usar a HANDTAB (handler table ?), situada a partir do endereço $035D. Veja a tabela abaixo, retirada do documento "De Re Intellivision - part 2":

```
0035DH Pointer to HANDTAB
HANDTAB (each a pointer to a handler routine, stored lo, hi)
    + 000H DISC          R0 = disc value (0 - 15) or negative if released
    + 002H KEYPAD        R0 = keypad value (0 - 9, 10 = clear, 11 = enter)
    + 004H UPPER BUTTONS R0 = 1 if pressed, -1 if depressed (?)
    + 006H LOWER RIGHT   same
    + 008H LOWER LEFT    same    R1 = 0 for left controller, 1 for right (?)
```

Vamos ao programa então. Você entenderá tudo olhando o código dele.

```
;;-----------------------------------------------------------------------
;; CONTROL2.ASM
;;-----------------------------------------------------------------------
;; by Carlos Madruga (aka Helixx)
;; 7 OCT 2000
;; This program does use EXEC calls.
;;
;; Compiled with the AS1600 compiler.
;;
;; This simple program shows how to handle controller inputs.
;; Feel free to modify and experiment with it !
;;
;;-----------------------------------------------------------------------
        ROMW    10
        ORG     $5000

;;-------------------------------------------------------------------------------
SCRATCHPAD:

LR          EQU   $130
POS         EQU   $140
SCINI       EQU   $200
```

Opa ! Novidade ! O que é Scratchpad ???
Scratchpad é uma área de memória mapeada entre $0100 e $01EF. Ela está contida em um chip de RAM de 8 bits, e tem dois usos:
1- entre $0100 e $015C, é usada para armazenamento de dados temporários do EXEC. Cuidado quando mexer aqui !
2- entre $015D e $01EF, pode ser usada à vontade para guardar dados do seu jogo, como variáveis por exemplo.

Neste programa, usei os endereços $130 e $140 para guardar os valores das variáveis LR e POS.
A variável LR dirá se o controle que foi acionado é Left ou Right.
A variável POS dirá por exemplo em qual direção o disco foi apertado, ou qual tecla do keypad foi pressionada.
A variável SCINI na verdade não tem nada a ver com Scratchpad. Ela somente guarda uma constante equivalente ao primeiro card da tela (que vai de $200 a $240). Mas deixei esta variável aqui para deixar todas agrupadas.
(DICA: Para ver como a memória do Intellivision é organizada, veja o documento MEM_MAP.TXT, que vem dentro do SDK1600.)

```
;-------------------------------------------------------------------------------
ROMHDR:
        BYTE    $01,$00         ; Movable object data (ignored)
        WORD    $0000           ; RTAB (ignored)
        WORD    START           ; Program start address (ignored)
        WORD    $0000           ; Background graphics
        WORD    ROMHDR          ; Card table
        WORD    TITLE           ; Title string

        BYTE    $80             ; run code after title, clicks off
        BYTE    $00             ; -> to STIC $32
        BYTE    $00             ; 0 = color stack, 1 = f/b mode
        BYTE    0, 0, 0, 0      ; color stack elements 1 - 4
        BYTE    $00             ; border color
;-------------------------------------------------------------------------------


TITLE:        BYTE    100, "Controller Test", 0

;;-----------------------------------------------------------------------
POST_TITLE:
        MVO@    R5, R6           ; Push R5
        MVII    #$2D0, R4       ; Where the copyright date goes
        JSR     R5, $187B       ; call print routine
```

```
        BYTE    "2000 Madruga", 0   ; Overwrite initial screen
        MVI@    R6, R7              ; Return from Title Screen
```

Nada de novo... note que não usamos Moving Objects neste programa, e que deixamos a borda com a cor Black.

```
;;-------------------------------------------------------------------------
START:
        MVO@    R5, R6          ; Push R5
        MVII    #$0200, R4       ; Start of ScreenMem
        MVII    #240, R0         ; To clear entire screen
        JSR     R5, $1738        ; CLEARMEM

        MVII    #$0007, R3       ; Color to print the message
        MVII    #SCINI+$3C,R4
        JSR     R5, $187B
        BYTE    "   Press anything !",0
```

Aqui limpamos a tela e imprimimos a mensagem, usando a variável SCINI para posicionar a mensagem no card correto.

```
   SDBD
   MVII    #HANDTAB, R0
   MVO     R0, $035D        ; points $035D to HANDTAB
   MVI@    R6, R7           ; THE END

HANDTAB:
        WORD     DISC
        WORD     KEYPAD
        WORD     UPPER
        WORD     LOWERRIGHT
        WORD     LOWERLEFT
```

Aqui veio a parte interessante: a HANDTAB.
Colocamos o registrador R0 apontando para o endereço inicial da HANDTAB através da instrução MVII. Como este endereço estava lá na frente, por segurança colocamos a instrução SDBD antes (o assembler deveria fazer isso sozinho, mas bugs são bugs). Já comentei sobre o uso de SDBD e este problema em um tutorial anterior.
Em seguida, com a instrução MVO, depositamos o valor de R0 (HANDTAB) no endereço $035D, que é o endereço que interessa para o EXEC. Assim, o EXEC sabe onde está a HANDTAB !
Depois, terminamos a rotina principal com o bom e velho MVI@ R6, R7.

A HANDTAB está logo a seguir, e é composta de 5 words. Estas words são os nomes das rotinas de tratamento dos controles, E NÃO DEVEM TER SUA ORDEM ALTERADA.

```
; Disc handler

DISC:       MVO@    R5, R6
            MVO     R0,POS
            MVO     R1,LR

            MVII    #$0002, R3       ; Color to print the message
            MVII    #SCINI,R4
            JSR     R5, $187B
            BYTE    "Disc     position", 0

            MVII    #$0007, R3       ; Color to print the message
            MVII    #SCINI+$3C,R4
            JSR     R5, $187B
            BYTE    "Controller:         #0=left #1=right
                     Directions:         0(right) to 15 ccw                ", 0

            JSR     R5, PRINT
            MVI@    R6,R7
```

Aqui temos a primeira subrotina de tratamento do disco pressionado.

Quando você pressiona o disco, o EXEC armazena em R0 a informação de qual controle foi acionado (Left / Right), e em R1 qual direção foi pressionada (0 a 15 em sentido anti-horário).
Primeiramente, guardamos estes valores nas variáveis LR e POS, para não termos o risco de perdê-las se por algum motivo o conteúdo de R0 e R1 for alterado.
Em seguida, imprimimos a mensagem "Disc position " em vermelho na tela, e depois uma mensagem explicando o que são os valores capturados dos controles.
Finalmente, chamamos a rotina PRINT para encaixar os valores de LR e POS naquela mensagem em vermelho.
Por que coloquei este último passo numa rotina separada ? Porque todas as outras subrotinas de tratamento de controles farão isto também !

Este programa poderia ser mais simplificado ainda, mas deixo a cargo de você :-)

```
KEYPAD:     MVO@    R5, R6
            MVO     R0,POS
```

```
            MVO     R1,LR

            MVII    #$0003, R3       ; Color to print the message
            MVII    #SCINI,R4
            JSR     R5, $187B
            BYTE    "Keypad   position ", 0

            MVII    #$0007, R3       ; Color to print the message
            MVII    #SCINI+$3C,R4
            JSR     R5, $187B
            BYTE    "Controller:         #0=left #1=right
                     Keys:               0 to 9           10=clear   11=enter", 0

            JSR     R5, PRINT
            MVI@    R6,R7


UPPER:      MVO@    R5, R6
            MVO     R0,POS
            MVO     R1,LR

            MVII    #$0004, R3       ; Color to print the message
            MVII    #SCINI, R4
            JSR     R5, $187B
            BYTE    "Upper    position", 0

            MVII    #$0007, R3       ; Color to print the message
            MVII    #SCINI+$3C,R4
            JSR     R5, $187B
            BYTE    "Controller:         #0=left #1=right
                     Status:             0=pressed                          ", 0

            JSR     R5, PRINT
            MVI@    R6,R7


LOWERRIGHT: MVO@    R5, R6
            MVO     R0,POS
            MVO     R1,LR

            MVII    #$0007, R3       ; Color to print the message
            MVII    #SCINI, R4
            JSR     R5, $187B
            BYTE    "LowerR   position", 0

            MVII    #$0007, R3       ; Color to print the message
            MVII    #SCINI+$3C,R4
            JSR     R5, $187B
            BYTE    "Controller:         #0=left #1=right
                     Status:             0=pressed                          ", 0

            JSR     R5, PRINT
            MVI@    R6,R7


LOWERLEFT:  MVO@    R5, R6
            MVO     R0,POS
            MVO     R1,LR

            MVII    #$0006, R3       ; Color to print the message
            MVII    #SCINI, R4
            JSR     R5, $187B
            BYTE    "LowerL   position", 0

            MVII    #$0007, R3       ; Color to print the message
            MVII    #SCINI+$3C,R4
            JSR     R5, $187B
            BYTE    "Controller:         #0=left #1=right
                     Status:             0=pressed                          ", 0

            JSR     R5, PRINT
            MVI@    R6,R7
```

Aqui tivemos todas as rotinas de tratamento, que são muito parecidas. Acho que dispensam comentários.

```
PRINT:      MVO@    R5,R6
            MVI     LR,R0
            MVII    #1,R1            ; number of digits
            MVII    #1,R3            ; color
```

```
MVII    #SCINI+7,R4      ; beggining position
JSR     R5,$18C5         ; call print R0 for L/R info

MVI     POS,R0
MVII    #3,R1            ; number of digits
MVII    #1,R3            ; color
MVII    #SCINI+$11,R4    ; beggining position
JSR     R5,$18C5         ; call print R0 for status info
MVI@    R6,R7
```

A rotina PRINT funciona do seguinte modo:
Recupera-se os valores das variáveis LR e POS colocando-os de volta no registrador R0 (uma de cada vez).
Em seguida, chamamos uma rotina do EXEC destinada a imprimir o valor de R0 na tela.

Esta rotina precisa dos seguintes parâmetros:
R1 = número de dígitos máximo do valor a ser mostrado.
R3 = cor da mensagem a ser mostrada.
R4 = a posição a ser impressa na tela (colocada em função do endereço inicial SCINI)
R0 = obviamente, o valor a ser impresso.

Esta rotina fica em $18C5, e é chamada 2 vezes...

Pronto ! Acabou !
Mais mole que isso, só ... bom, esqueça.

Algumas sugestões de exercícios:
1 - Faça com que a tela mude de cor conforme cada botão for pressionado.
2 - Tecle F3 no Intellivision Lives pra inverter os controles, e observe como o programa se comporta.