PROGRAMMING COLECOVISION GAMES

IN C LANGUAGE
WITH HI-TECH C COMPILER
UNDER WINDOWS/DOS ENVIRONMENT

BY DANIEL BIENVENU

Version 1-k

Last update: October 26, 2005

Table of content

INTRODUCTION

Dear ColecoVision programmers,

This document shows you the basic of the ColecoVision capabilities with technical information, programming tools and simple codes in C language. This document is "a good starting point" to learn how to program ColecoVision games in C language like I did. However, don't hesitate to seek for more information and to ask questions to other ColecoVision programmers.

The first part of this document contains the basic concepts of the ANSI C language based on the Coleco library and the Hi-Tech C compiler. If you don't know how to program in ANSI C language, look for C programming (not C++ or C#) in the Internet.

The second part of this document talks about tools (for Windows) Marcel de Kogel and I programmed to speed up the development process.

The third part of this document is about specific libraries already made for the ColecoVision game development, and programming samples.

The annexe gives you technical information for programmers about the ColecoVision.

Before trying to do your first ColecoVision project, improve your programming skills first by testing all the concepts mentioned in this documentation before thinking of releasing new ColecoVision games.

This document can be used as a tutorial and a reference guide.

Have fun making new games for the ColecoVision game system! Good Luck! ☺

Best regards,

Daniel Bienvenu

SETUP YOUR DEVELOPMENT ENVIRONMENT

If you are using a Windows environment or a Linux with a great DOS emulator, you may have no problem to set up this programming environment.

First choice:

First, you need to download the Hi-Tech C compiler for CP/M (freeware).
Second, you need to download a CP/M emulator for your system. 22NICE for DOS is my personal choice but you need to read carefully the text files because you have to modify the Hi-Tech C compiler executables.
Third, you need to download the Coleco library by Marcel de Kogel. You have to extract files into the Hi-Tech C compiler directory.

For Windows users, you can avoid all these steps by downloading an archive named “z80.zip” from the web site.

Use your preferred text editor to code in C your Coleco projects. Make sure your C code is in a pure text file format before trying to compile it. My personal choice is NOTEPAD or Win32PAD.

You have two choices now to access to your Coleco environment:

- Create a shortcut on your desktop to run the CP/M emulator and go directly into the Hi-Tech C compiler directory.
- Use a GUI (front-end) to use this environment. I personnaly made one for windows named CCI (Coleco Compiler Interface). You just need to put the EXE file into your project directory to avoid using the command-line based interface under DOS.

Note: There is an online help section in the web site to show you how to use the compiler with a sample code.

Second choice:

Download and install the Hi-Tech C compiler for DOS (shareware) and try using the ColecoVision and Getput1 libraries with this environment. Because it's the same company, this solution may works just fine.

Third choice:

Download and install a cross C compiler for your system to be able to compile a binary file for the Zilog 80 processor. Download the libraries source code to adapt them for your cross-compiler.

MEMORY

Before programming a ColecoVision project, you must learn about the memory "limits" for this game system.

ROM (Read Only Memory)

The binary code in the cartridge is named ROM because it's a read-only memory. The memory space for the game starts at 8000. Today, we have no problem using big memory capacity so we try to use all the 32K available: 8000 - FFFF. At 8000, there is a header. The header starts with 55 AA or AA 55. After this 2 bytes, there are hex values in the header to say where start the game, what to do if there is an interrupt (example: rst 08h, NMI), etc.

The ColecoVision BIOS start at the address 0000 and it's the first thing running in the ColecoVision game system. First, the BIOS check if a cartridge is inserted by looking for the first two bytes of the ROM (cartridge). If these two bytes are “55 AA” or “AA 55”, the Coleco BIOS check the start address in the ROM header then start the game, otherwise, a default screen appear about how to insert a cartridge: “TURN GAME OFF BEFORE INSERTING CARTRIDGE OR EXPANSION MODULE”. Second, the BIOS had many routines like sounds and sprites routines. Using the BIOS routines gives more free ROM space for the game itself but you have to be careful to not use address in RAM used by the BIOS.

By using the Coleco library by Marcel de Kogel, don't worry about the header of your Coleco project because it’s already compiled into the “crtcv.obj” file.

RAM (Read-Write Memory)

The ColecoVision RAM is a bit weird. There is only 1K RAM (with a copy of itself at another memory location) and this RAM is not fully available if you use routines in the Coleco BIOS. The Coleco BIOS routines use some parts of the RAM at specific memory locations above 73B8. This is important to know to avoid memory corruption by using too big tables in RAM. Don’t worry, normally, your first ColecoVision project will never use more than the free memory space available… except if you implement complex AI and/or big dynamic environment.

MEMORY MAP

The ColecoVision BIOS starts at the address 0000 and ends at the address 1FFF.
The cartridge (ROM) starts at the address 8000 and ends at the address FFFF.
The read-write memory space is only 1K. The real addresses for the RAM are 7000-73FF. The RAM space addresses available in memory for your ColecoVision projects are 7000-73B8. The stack pointer is initialised at 73B9 (firsts instructions in BIOS) but the stack really started at 73B8, 73B7, 73B6, etc. If you need more RAM space, you can use the video memory.

RAM USED BY THE COLECOVISION BIOS

Thanks to Steve Bégin for all the informations (for expert only).

7020-702A: Used by BIOS sound routines
73B9-73C2: Used but we don't know by which routines yet
73C3: Used by video Read/Write routines. Copy of the video control register number 0.
73C4: Used by video Read/Write routines. Copy of the video control register number 1.
73C5: Seems to be unused by any BIOS routines (free)
73C6: Used by BIOS sprite routines.
73C7: Used by BIOS sprite routines.
73C8-73C9: Used by BIOS to generate Random numbers (call 1FFD).
73CA-73D2: Used by BIOS sprite routines.
73D3-73D6: Used by BIOS timer routines.
73D7-73EA: Used by BIOS joystick routines. (call 1FEB)
73EB: Used by BIOS joystick routines. Value of spinner on port#1
73EC: Used by BIOS joystick routines. Value of spinner on port#2
73ED: Seems to be unused by any BIOS routines (free)
73EE-73F1: Used by BIOS joystick routines. Raw joystick data from ports (Call 1F76).
73F2-73F3: Used Address of sprites attribute in VRAM.
73F4-73F5: Used by video Read/Write routines. Address of sprites pattern in VRAM.
73F6-73F7: Used by video Read/Write routines. Address of screen image (NAME).
73F8-73F9: Used by video Read/Write routines. Address of character pattern (PATTERN).
73FA-73FB: Used by video Read/Write routines. Address of character color pattern (COLOR).
73FC-73FD: Seems to be unused by any BIOS routines (free)
73FE-73FF: Temporary used when a call at routine in 1fbe is made.

Ok, it's more information than you really need to know especially if you program your ColecoVision projects in C like me. The most important to know is you can't use the address 73B9-73FF in RAM for your own purpose. The only exception is the RAM used by the BIOS for the sound routines. Marcel de Kogel writes his own sound routines in the ColecoVision library so you can use the addresses 7020-702A for your game without any problem. The only problem is the data sound format is not the same between BIOS sound routines and ColecoVision library sounds routines. In this document, you will not find information about BIOS sound routines.

If you don't really know what is a stack, you may have a problem to understand why you can't really use all the free memory space 7000-73B8. If you program in C language, the Hi-Tech C compiler add some codes "pop" and "push" to stock information in the stack like the registers status. If you program in ASM, you have to add by yourself each "pop" and "push" instructions so you have more control but you have also more responsibility if your program doesn't run well. The stack is filled with data by decreasing first the stack pointer then by adding information. So, the real RAM space you can use depends on how big your stack can be. I think you must not use RAM address over 7300, otherwise you may have a problem of memory corruption.

INITIALIZING TABLES AND ARRAYS

Tables in another C file are declared with the instruction "extern".

init.c

```
#include <coleco.h>

extern byte pattern[];
```

tables.c

```
#include <coleco.h>

byte pattern[] = {0xff,0xfe,0xfc,0xf8,0xf0,0xe0,0xc0,0x80};
```

Tables restricted in a C file are declared with the instruction "static".

init.c

```
#include <coleco.h>

static byte pattern[] = {0xff,0xfe,0xfc,0xf8,0xf0,0xe0,0xc0,0x80};
```

Tables in ROM are initialised directly outside functions.

init.c

```
#include <coleco.h>

byte pattern_rom[] = {0xff,0xfe,0xfc,0xf8,0xf0,0xe0,0xc0,0x80};
```

Tables in RAM are initialised inside functions by using other tables, or filled with a value.

init.c

```
#include <coleco.h>

byte pattern_ram[8];
byte pattern_rom[] = {0xff,0xfe,0xfc,0xf8,0xf0,0xe0,0xc0,0x80};

void init_pattern_ram(void) {
  memcpy(pattern_ram, pattern_rom, 8); /* Initialize pattern_ram with pattern_rom */
  memset(pattern_ram,0x00,8); /* Set the 8 bytes of pattern_ram table to 0x00 */
}
```

DATA TYPES

char

Definition: character or short integer
1 byte and signed
Values : [-128, 127]

Example:

```
char c = 'A';
```

byte

Definition: byte or short unsigned integer
1 byte and unsigned
Values : [0, 255]
This type is defined in the coleco.h file.

Example:

```
byte i = 200;
```

int

Definition: integer
2 bytes and signed
Values : [-32768, 32767]

Example:

```
int i = -1000;
```

unsigned

Definition: unsigned integer
2 bytes and unsigned
Values : [0, 65535]

Example:

```
unsigned score = 1000;      print_at (10,0,str(score));
```

float

Definition: floating point
2 bytes and signed
Values : [??, ??]

Example:

```
float ratio = 10.0/4.0;
```

Note: Try to never use floating point variables.

char [n]

Definition: character array or string
Max 256 bytes long

Note: A string ends with the character '/0'.

Example:

```
char numbers[]={'0','1','2','3','4','5','6','7','8','9'};
```

* (pointer)

Definition: pointer, address in memory where is the data
2 bytes
Values : [0000 , FFFF] (0,65535)

Example:

```
int score1, score2;
int *score;          /* pointer to integer value named score */
score = &score1;  /* set score pointer to score1 */
char *msg;          /* char pointer, also used as a variable size string variable */
```

Note: A character pointer is not a character array.

void

Definition: no defined type, void
2 bytes by default

Example:

```
void *msg;
```

Note: This data type must be reserved for routines only.

SPECIAL DATA TYPES

The following data types are defined into specialized libraries, not included with the C compiler.

From Marcel's Coleco library

sprite_t

Definition: Sprites table data format, consist of 4 bytes named: y, x, pattern and colour.

Example:

```
extern sprite_t sprites[];
put_vram(0x1b00, sprites, 13); /* 1b00 is default vram address for sprite_attribute_table */
```

From modified version of Marcel's Coleco library, to use Coleco bios sound routines.

sound_t

Definition: Songs table data format, consist of a pointer to sound data and an address to a sound area.
Predefined sound areas addresses are : SOUNDAREA1, SOUNDAREA2... SOUNDAREA7.

Exemple:

```
static byte sound1[] = {0x43,0x00,0x72,5,0x11,0x90,0xF3,0x11,0x50};
sound_t snd_table[] = {sound1,SOUNDAREA1};
play_sound(1); /* Play the first sound data specified in the sound table */
```

From Getput v1.1 library.

score_t

Definition: Score format is composed into two unsigned values. Needs special functions.
Values: [0,655359999]

Exemple:

```
score_t score;
score_reset(&score);
score_add(&score,1000);
print_at(10,12,score_str(&score,9));
```

STRUCTURES (CUSTOM DATA TYPE)

typedef struct

definition: conglomerate data structure

Example:

```
/* coordinates (x,y) */
typedef struct {
        byte x;
        byte y;
} coorxy;

coorxy position[10];

position[0].x = 2;
```

Note: A structure like this one is used in the Coleco library for the sprites table in ram.

Important: When using a pointer to a structure element, you access data by using the arrow "->" syntax, not the dot.

Exemple :

```
/* coordinates (x,y) */
typedef struct {
        byte x;
        byte y;
} coorxy;

coorxy position[10];

coorxy *ptr_position = &position[index];

ptr_position->x = 2;
```

OPERATORS

USUAL SET OF BINARY ARITHMETIC OPERATORS:

- multiplication (*)
- division (/)
- modulus (%)
- addition (+)
- subtraction (-)

Note: Unary minus performs an arithmetic negation. Unary plus is supported.

INCREMENT (++) AND DECREMENT (--)

The most well know unary operators are increment (++) and decrement (--). These allow you to use a single operator that "adds 1 to" or "subtracts 1 from" any value. The increment and decrement can be done in the middle of an expression, and you can even decide whether you want it done before or after the expression is evaluated.

Example:

```
int sum = a + b++; /* sum = a + b then increment 'b' */
int sum = --a + b; /* decrement 'a' then sum = a + b */
```

BITWISE OPERATORS

- shift left (<<)
- shift right (>>)
- AND (&)
- OR (|)
- XOR (^)
- NOT (~)

COMBINED OPERATORS

The expression form:
<variable> = <variable> <operator> <expression>;

Can be replaced with:
<variable> <operator>= <expression>;

Example:

```
a +=b; /* is the same thing as the expression "a = a + b;"
```

RELATIONAL OPERATORS

Relational operators allow you to compare two values, yielding a result based on whether the comparison is true or false. If the comparison is false, then the resulting value is 0.

- greater than (>)
- greater than or equal (>=)
- less than (<)
- less than or equal (<=)
- equal to (==)
- not equal to (!=)

LOGICAL OPERATORS

There are three logical operators:

- AND (&&)
- OR (||)
- NOT (!)

Example:

```
If (!(a==b && b==c)) /* If not (a equal to b and b equal to c) then… */
If (a<b || a<c) /* If a less than b or a less than c then… */
```

Note: Do not be confused with the bitwise operators (&,!,~) previously mentioned. If you use "MASK" in a complex if condition you must isolate the bitwise operation with (and) like this: if ((a&b)==c) /* If a with an "AND MASK" b is equal to c then… */

IF STATEMENT

SIMPLE IF

if (condition) statement1;

Example:

```
if  (x==0) {x++;}
```

IF … ELSE

if (condition) statement1; else statement2;

Example:

```
if  (x==0) {x++;} else {x--;}
```

LOOPS

FOR LOOP

Example:

```
for (x=0;x<10;x++)
```

WHILE LOOP

Example:

```
x=0; while (x<10) {x++;}
```

DO WHILE LOOP

Example:

```
x=0; do {x++;} while (x<10);
```

SCREEN MODES

Before seeing any "HELLO WORLD" on screen, you have to setup the screen mode and update the video ram memory with an appropriate characters set.

To find the information about the Video Display Processor (VDP), look at this txt file.
URL:
http://www.msxnet.org/tech/tms9918a.txt
http://home.swipnet.se/~w-16418/tech_vdp.htm

For a text adventure, the screen mode 1 with 40 columns should be a good choice. But for the other Coleco projects, the screen modes 0 and 2 are the most appropriate choices. The screen mode 3 (very big pixels on screen) is not a good choice. Do not use the undoccumented screen mode. For all my ColecoVision projects, I use the screen mode 2. This screen mode allow me to do bitmap title screen and colorfull characters set. Compute the VDP control registers values based on your own requierments.

In the Coleco library, “vdp_out” is a routine to update the VDP control registers' values:

vdp_out(register,value);

The most important control registers are 0 and 1.

Reg. 0	-	-	-	-	-	-	M2	ExtVID
Reg. 1	4K/16K	BLANK	IE	M1	M3	-	SIZE	MAG

M1, M2 and M3 are the bits to set the screen mode.
EXVID is External VDP input (always disable this one with bit 0)... see TXT file

To set the screen mode 2 (like in my all my Coleco projects), I compute M1=0 (disable), M2=1 (enable), M3=0 (disable), ExtVID=0 (disable), 4/16K=1 (16K), BLANK=1 (enable display), IE=1 (enable NMI interruptions), SIZE=1 (16x16 sprites), MAG=0 (normal sprites, not doubled in size).

Reg. 0	0	0	0	0	0	0	1	0	02
Reg. 1	1	1	1	0	0	0	1	0	E2

SCREEN MODE 2 (normal with 16x16 sized sprites) :
vdp_out(0,2);
vdp_out(1,0xe2);

We need to setup the other VDP registers too to complete the screen mode setup.

For the special screen mode 2, we have to imagine the screen divided in three parts: TOP, MIDDLE, BOTTOM. Each part (of 8 lines) use the same character set or different character sets. Normally, we use three different character sets to do a bitmap title screen. Otherwise, only one character set is needed. You must read carefully the text files about VDP before trying to compute yourself the control registers values.

CHARACTERS

The characters are used for most of the graphics on screen. They can be copied many times on screen without any problem. All the letters, numbers and symbols are characters. A character is 8x8 pixels sized except for screen mode 1 where a character is only 6x8. Normally, there is 32x24 spaces on screen where characters can be placed except for screen mode 1 (40 columns).

VIDEO MEMORY FOR CHARACTERS

The names of the three tables in Video RAM for characters are:

NAME: The screen (24x32)
PATTERN: The characters pattern (256 characters: HEX values 00-FF)
COLOR: The characters color(s)

A character has the same pattern and color anywhere on screen (one exception in screen mode 2). So you can't use the same character to print a blue 'A' and a red 'A' side-by-side. The solution is using two characters with the same pattern but with different colors.

In the ASCII code, the character '1' is the character 49 (31 in HEX value). You must understand the difference between the ASCII code and the symbol. The ASCII code for the character 'A' is 65 (41 in hex value).

Now, if the color of the character 'A' is blue and if you change the pattern of the character '1' to looks like an 'A' but with a red color, then you just have to print the characters 'A' and '1' side-by-side on screen to show two 'A' side-by-side... one blue and one red.

In the NAME table, there is HEX values 41 for the 'A' and 31 for the '1'.
In the PATTERN table, there are identical HEX values for the character 'A' and '1'.
In the COLOR table, there are different HEX values to have blue color(s) for the character 65 ('A') and red color(s) for the character 49 ('1').

DACMAN is a good example of a video game based on characters graphics.

In screen mode 0, there are only two colors (one color for bits 1 and one color for bits 0) for each bloc of 8 characters in the character set.
In screen mode 1, there are only two colors (one color for bits 1 and one color for bits 0) for all the characters.
In screen mode 2, there are two colors (one color for bits 1 and one color for bits 0) by line of 8 pixels for all the characters in the character set.

For the screen mode 0 and 2, if you want to see the background color set in the VDP register, you have to use the INVISIBLE color “0”.

CHARACTER PATTERN

A character pattern is an 8x8 graphic. Some guys name this kind of graphic: tile.

EXAMPLE - SPACESHIP

Spaceship pattern

0	0	0	1	1	0	0	0	18
0	0	0	1	1	0	0	0	99
1	0	0	1	1	0	0	1	99
1	0	1	1	1	1	0	1	BD
1	1	1	0	0	1	1	1	E7
1	1	1	0	0	1	1	1	E7
1	0	1	1	1	1	0	1	BD
0	0	1	1	1	1	0	0	3C

In screen mode 0, a character can use only two colors (one for bits 1 and one for bits 0).

Spaceship colors

Bits 1	Bits 0	Code
		E1

Spaceship pattern with colors

0	0	0	1	1	0	0	0
1	0	0	1	1	0	0	1
1	0	0	1	1	0	0	1
1	0	1	1	1	1	0	1
1	1	1	0	0	1	1	1
1	1	1	0	0	1	1	1
1	0	1	1	1	1	0	1
0	0	1	1	1	1	0	0

In screen mode 2, a character can use more than two colors but limited to two colors per line.

Spaceship colors

Bits 1	Bits 0	Code
		81
		A1
		E1
		E1
		EF
		E7
		E1
		81

Spaceship pattern with colors

0	0	0	1	1	0	0	0
1	0	0	1	1	0	0	1
1	0	0	1	1	0	0	1
1	0	1	1	1	1	0	1
1	1	1	0	0	1	1	1
1	1	1	0	0	1	1	1
1	0	1	1	1	1	0	1
0	0	1	1	1	1	0	0

UPLOAD CHARACTER SET

To upload a character set in video memory, you can use the “upload_ascii” routine from Coleco library.

upload_ascii (number of the first character to upload, number of characters to upload, offset, flags for the character format);

Example: upload_ascii (29,128-29, chrpos+29*8, BOLD); /* chrpos is the address location in video memory for the charcarter patterns */

But, you can use your own character set. You can use one of my tools named Win-ICVGM (I.C.V.G.M for Windows) to create your own character set. This tool is based on the screen mode 0 but can be used for the screen mode 2 too (except for the color patterns).

URL:
http://www.geocities.com/newcoleco/tools.html

When your character set is done, save as a C file and upload your character set into video memory by using one of these routines from Coleco library.

put_vram (video memory location for character patterns, pointer to character set table without RLE compression, size of the character set table);

rle2vram (pointer to character set table with RLE compression, video memory location for character patterns);

SPRITES

The sprites are easy to use because you can place them anywhere on screen. Each sprite can be identified like a layer on screen. Normally, the size of a sprite is 16x16 but there is also the 8x8 format. The limits for using sprites are : never more than 4 sprites in a row, on the same scan line and never more than 32 sprites on screen at the same time. All sprites can be magnified by 2 (by changing size of the pixels in sprites).

To display a sprite on screen, you need a vector of 4 bytes (Position Y, Position X, Pattern and Colour) in the right video memory location.

SPRITES COLOR

The bits 0 are already replaced by the invisible color so there is only one color per sprite.
To use more than one color, you have two solutions:

- Use more than one sprite (one for each color)
- Use a combination of sprites and characters like the solution used for the ghosts in Atarisoft PacMan game: the eyes are one or two characters in grey color and the body is a sprite.

SPRITES LOCATIONS ON SCREEN

The Y location of a sprite can be any values between 0 and 255 except 208. The special value 208 tells the video chip to stop checking for sprites to display on screen. If you want to not show sprite#1 but you want to show sprite#2, use a value like 207 for the Y location of sprite#1.

SPRITES PATTERN

8x8 SPRITE

A 8x8 sprite looks like a character in screen mode 0 but all bits 0 are colored with the invisible color 0. If we re-use the spaceship example, a 8x8 spaceship sprtie could be something like this.

Spaceship color

Color	Code
	4

Spaceship pattern with color

0	0	0	1	1	0	0	0
1	0	0	1	1	0	0	1
1	0	0	1	1	0	0	1
1	0	1	1	1	1	0	1
1	1	1	0	0	1	1	1
1	1	1	0	0	1	1	1
1	0	1	1	1	1	0	1
0	0	1	1	1	1	0	0

16x16 SPRITE

A 16x16 sprite is a combination of four (4) 8x8 patterns. These patterns are displayed like this:

1	3
2	4

Win-ICVGM can be used to create sprites pattern 16x16.
Sprite pattern with color code A (10)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00	00
0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	03	E0
0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0F	F8
0	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0	19	CC
0	0	1	1	0	1	1	0	1	0	1	1	0	1	1	0	36	B6
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	3F	FE
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7F	FF
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7F	FF
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7F	FF
0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	70	07
0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	30	06
0	0	1	1	1	0	0	0	0	0	0	0	1	1	1	0	38	0E
0	0	0	1	1	1	1	0	0	0	1	1	1	1	0	0	1E	3C
0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0F	F8
0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	03	E0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00	00

The sprite pattern is coded like this in video memory:
00, 03, 0F, 19, 36, 3F, 7F, 7F, 7F, 70, 30, 38, 1E, 0F, 03, 00, 00, E0, F8, CC, B6, FE, FF, FF, FF, 07, 06, 0E, 3C, F8, E0, 00

SPRITES ROUTINES

Note: These routines are available in the Coleco library.

```
/* Update sprites attributes in VRAM */
update_sprites (number of sprites, sprites table);

/* Make sprites disappear by changing their Y location (in video memory only) to 207 */
clear_sprites (first sprite to clear, number of sprites to clear);
/* Make sure no sprite is showed on screen, use the following code */
clear_sprites (0, 64);

/* Check collision is complex to understand. Read Coleco library section */
check_collision (…);
```

JOYSTICK (Hand Controller)

The Coleco joystick is divided in two parts: keypad and joypad.

JOYPAD

Use joypad_1 (joystick in port#1) and joypad_2 (joystick in port#2) to know the direction(s) and the "pressed" fire button(s).

How to use the joypad_1 and joypad_2 values?
The value is code as a byte where each bit represents a fire button or a direction. You have to use an AND mask to extract what you need from the joypad value.

Fire 1	Fire 2	Fire 3*	Fire 4*	Left	Down	Right	Up

*: Only available for the Super Action controller.

Example: I want to do nothing until a fire button is pressed on port#1.

```
/* Bits for fire buttons are the four (4) highest bits in the joypad value. So, you need to do an AND mask with four (4) bits 1 and four (4) bits 0 like this: 11110000 = F0 */

while ( (joypad_1 & 0xf0) == 0 ); /* loop if no fire button is pressed (port#1) */
```

With Getput1 library, you can also use the predefined constants : FIRE1, FIRE2, FIRE3, FIRE4, LEFT, DOWN, RIGHT and UP. It makes coding more easier to read and write.

```
If (joypad_1 & FIRE1) /* if Fire 1 is pressed */
```

KEYPAD

Use keypad_1 and keypad_2 to know which number was pressed. The keypad_1 is for the keypad on port#1. The keypad_2 is for the keypad on port#2. The first time you call keypad_1 and keypad_2, the result 0 means nothing. After this first time, the keypad value will match with the right key number pressed.

0-9 = 0-9
10 = * (standard to pause the game and replay game
with the same game option)
11 = # (standard to return to menu after the game)
12-14 = unused
15 = no key pressed

OTHER CONTROLLERS

Super Action Controller

This special hand controller has four fire buttons and a speed roller. The speed roller is not quite effective, so you need to make it moving really fast. This controller is used in several games: Front Line, Rocky Super Action Boxing, Super Action Baseball, Super Action Football and Super Action Soccer.

The speed roller is a bit slow to be used as a paddle. It's why the speed roller is most a speed indicator like in the Super Action Baseball game to make running slow or fast baseball players. The signed value of the spinner shows you the direction of the movement and also the speed movement. Note that the spinner in port#2 works the same way as the spinner in port#1.

Expansion Module 2: Turbo Drive

This Turbo Drive controller allows you to actually drive vehicles in games: Bump'n Jump, Destructor, Dukes of Hazzard, and Turbo. The steering wheel is plugged into port#1. It needs four (4) C batteries and a joystick plugged into port#2. The accelerator pedal is only a trigger.

The spinner in port#1 is working backward for the Turbo Drive module. You have to subtract the port#1 spinner value to the 'x' position to obtain the new position.

Roller Controller

This free-rolling control ball (track ball) gives you a 360 degrees field of lightning fast movement. This controller is plugged into joystick ports to allow movement in all directions and power supply port to avoid using batteries like the Turbo Drive module. There are four fire buttons on the Roller Controller, these buttons are (left) fire 1 and 2 for port#1 and (right) fire 2 and 1 for port#2. You need at least one joystick plugged into one of the joystick ports on the roller controller to select game options before playing. This controller is used in two games: Slither and Victory.

The spinner in port#1 is working backword for the roller controller, but the spinner in port#2 is working forward. You have to subtract the port#1 spinner value to the 'x' position and add the port#2 spinner value to the 'y' position to obtain the new position.

SPINNERS VARIABLES

In the Coleco library by Marcel de Kogel, the spinners values are updated in global variables named spinner_1 (port#1) and spinner_2 (port#2) each time a NMI interrupt occurs. It's a good idea to update the player position into the nmi routine, otherwise, the player movements may be not fluid.

Note: Declared as "extern byte" in the "coleco.h" file, the spinners variables type is much a char (signed value [-128,127]) than a byte (unsigned value [0,255]).

SOUND

I use the sound routines included in the Coleco library by Marcel de Kogel so I can talk a little about making some sounds in Coleco projects.

The way I learn how to generate sounds in my Coleco project is by looking the source code of Cosmo Challenge by Marcel de Kogel and the technical information about tone generator.

To avoid computing myself a sound effect, I created a tool named WAV2CV to convert a sound from a normal WAV file into a C file to be used with the Coleco library sound routines. I have done two programs about sound effects with menus. The source code is available: visit my Coleco web site.

URL:
http://www.geocities.com/newcoleco

Ok! My Coleco web site is not up-to-date but this is the web page where you can found some of my codes. Only in French, sorry!

URL:
http://www.geocities.com/newcoleco/dev/devfr.html

THE SOUND ROUTINES

At the end of the NMI routine, I add the following instruction:

update_sound ();

This instruction placed in the nmi will update the sound at each vertical retrace. Yes, in the NMI routine, it's the best place to put this instruction.

start_sound (sound_data,sound_priority);

This instruction adds information in the RAM like the pointer to the sound in ROM and the sound priority. The information in RAM will be used by the update_sound routine to play the sound. When 3 sounds are played and another sound is started, the sound_priority is used to see which sound must be ignored because there is only 3 sound channel. A sound priority number 10 win over a sound priority number 1. (I hope you understand)

sound_pointer = start_sound(sound_data, sound_priority);

You can get the information about the pointer in RAM where the sound information is placed. This way, you can use the following instruction to stop this particular sound without stops all sounds:

stop_sound(sound_pointer);

You cannot use stop_sound without the sound_pointer otherwise all sounds may stop forever.

sound_on();

This instruction enables sound output.

sound_off();

This instruction disables sound output.

play_dsound(sound_pointer, step);

This routine, originally from DSound library but now in Getput library, plays digital sounds. You can set the sound pitch speed with 'step' parameter. You must disable NMI before using this routine.

THE WAY I ADD SOUNDS IN MY COLECO PROJECTS

I don't fully understand how works the sound routine "update_sound" but I can talk about my tool WAV2CV.

It's very easy. Simply open wav2cv and select the WAV file you want to convert. You can also change the parameters before selecting the WAV file to increase or decrease the sound quality. If it's a simple sound like BEEP, use only one channel. If it's a complex sound like a digitalized sound, use two or three channels. Note: WAV2CV can freeze if you ask for more channels than you really need (it's a bug i can't fix). Normally, WAV2CV will be minimized for short laps of time and take 90% of your CPU time. More longer is the sound, more longer you will freeze your Windows.
WAV2CV applies FAST FOURRIER TRANSFORM to find frequencies of the sound... for each laps of time based on the vertical retrace frequency (NTSC 60Hz or PAL 50Hz).

The generated C file can be renamed and added into your Coleco project. You can also copy-paste the C code into your source code where you want to avoid having too much C files to compile. I suggest using one big C file to regroup all the sound data for your Coleco project.

WAV2CV cannot convert noisy sounds. To add a noise sound effect, you must refer to the sound data you can found in “Cosmo Challenge” source code.

```
byte shoot_sound[]= {
        1,
        0xf8,0xe4, 1,0xf2, 1,0xe4, 1, 0x01,0xe5,
        1,0xe4,
        1,0xe5, 1,0xe4, 1,0xe5, 1,0xe4,
        1,0xe5, 1,0xe4, 1,0xe5, 1,0xe4,
        5,
        0,0,0
};
```

Note: Normally, a sound data table ends with three zero like this: "0,0,0".

For more details, look at the "MARCEL's SOUND DATA FORMAT" (page 90).

TOOLS

The following pages show you some tools for Windows made by Marcel de Kogel and Daniel Bienvenu to help you in your ColecoVision projects. You may have to create your own tools if you are not satisfied.

Let's start with the only one who can convert sounds into ColecoVision sound format.

WAV2CV

by Daniel Bienvenu

WAV2CV is programmed to convert uncompressed mono WAV files into a ColecoVision format. But, the current version of WAV2CV is based on the ColecoVision library, not on the ColecoVision BIOS. It uses the Fast Fourier Transform to extract frequencies from the WAVE for each time pitch. The first conversion result is never perfect but you can use different parameters to see if the result can be better.

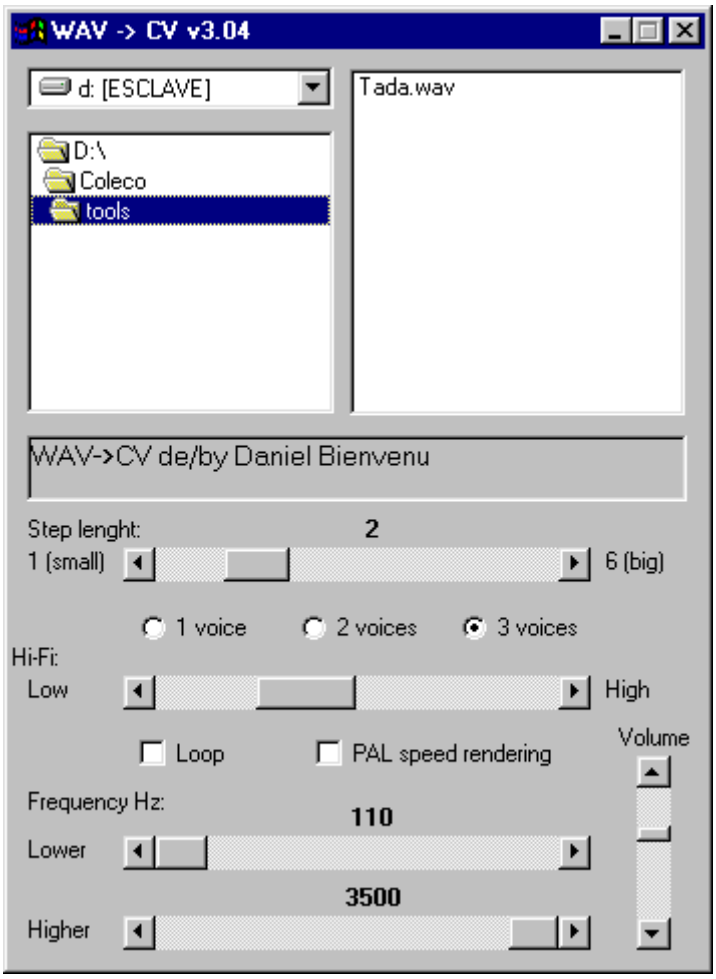


Figure 1 WAV2CV user interface

WAV2CVDS

by Daniel Bienvenu

WAV2CVDS is the only tool who can convert mono WAV files into a digital sound format for the ColecoVision. The digital sound format used is 4 bits per sample with special code $00 to indicate an RLE compression or the "end of sound" with another $00.

How to use it

1. Select a WAV file in the filelist box. The software read the WAV file primary information like the sample rate.
2. Write the sound size limit (in ROM) you want. The software compute the minimum step length you can use.
3. Set the step length of the digital sound. A big step gives you a sound with a poor quality but with less memory space than a small step.
4. Set the amplification.
5. Click on the "Generate Wav File" button to listen the digital sound.
6. Select your destination file format and click on the "Convert to Coleco" button.

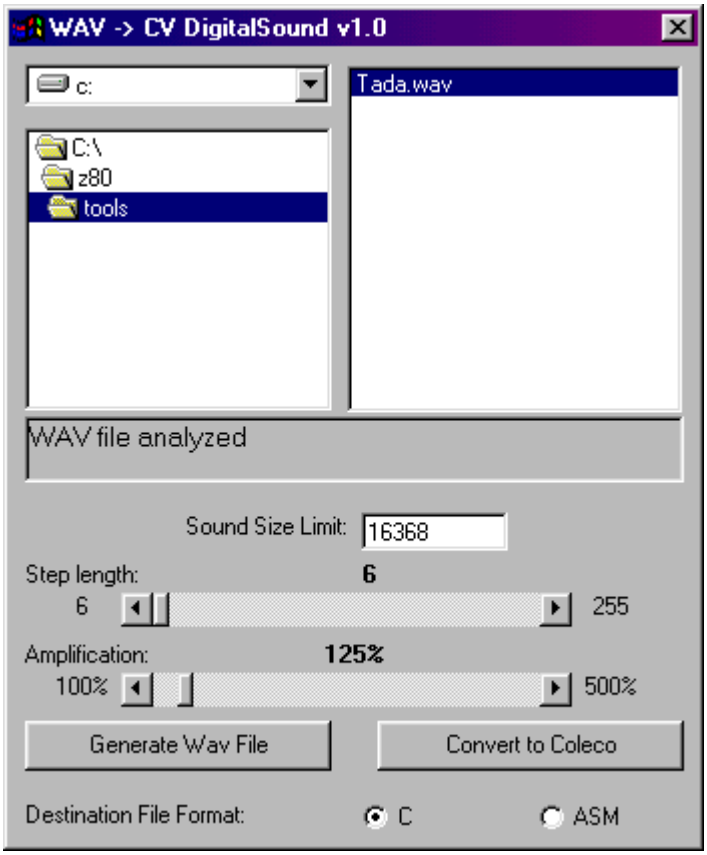


Figure 2 WAV2CVDS user interface

Use play_dsound routine from DSound library to play the digital sounds generated with this tool.

I.C.V.G.M. v2

by Daniel Bienvenu

ICVGM is my best graphics editor (for Windows) to quickly create characters and sprites. This software was based on CVEDITOR by John Dondzila and improved during the development of Miss Space Fury project. ICVGM is almost completed and stable.

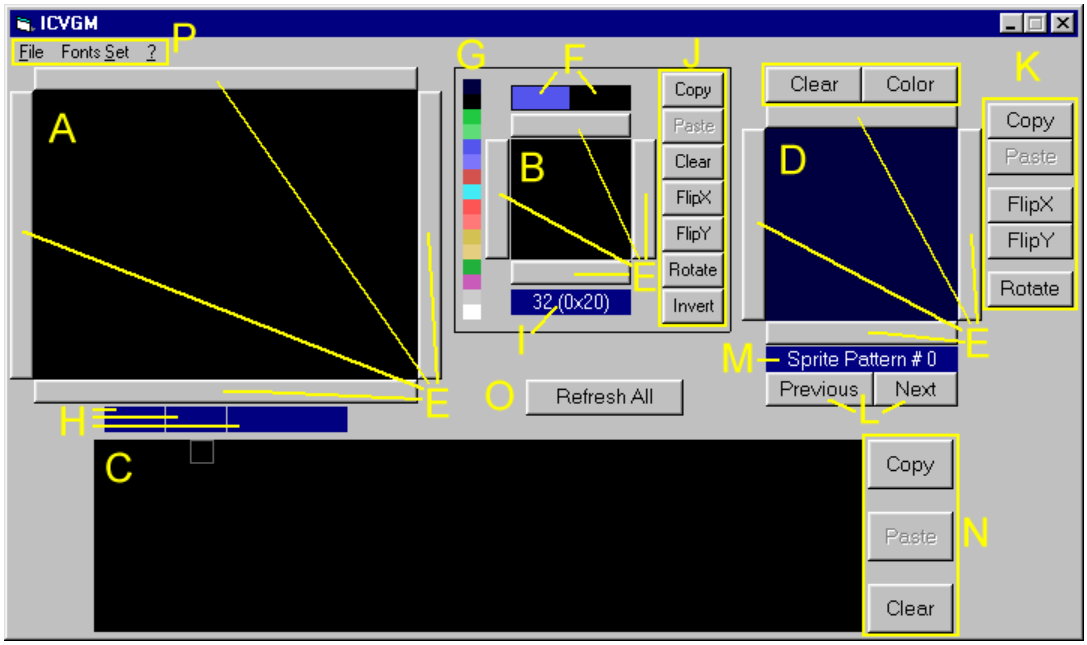


Figure 3 I.C.V.G.M. version 2 user interface

A. This is the screen area. Put characters in the screen by choosing first a character in the C zone then by clicking in the screen area.
B. This is the character editor area. Modify a character pattern by clicking in the character editor area with the left and right mouse buttons.
C. This is the character set (pattern) area.
D. This is the sprite editor area. Modify a sprite pattern by choosing first which pattern to edit with the Previous and Next buttons then by clicking in the sprite editor area with the left and right mouse buttons.
E. These buttons help you to move one step up, down, left and right.
F. This is the colors of the character. Change colors by clicking on it first then by clicking the color in the color palette.
G. This is the color palette.
H. Move your mouse over the screen area to find out which character number is at X,Y.
I. This is a simple indicator to know which character pattern you currently edit.
J. These buttons are useful tools to help you with your characters drawing.
K. These buttons are useful tools to help you with your sprites drawing.
L. These buttons help you to select a sprite pattern to edit.
M. This is a simple indicator to know which sprite pattern you currently edit.
N. These buttons help you to copy, paste and clear all the character set at once. This could be useful to copy a character set from another work file.
O. This button will refresh all the areas: screen, character editor, characters set and sprites.
P. This is the menu bar. Use "File" menu to load, save and export. Use "Fonts set" menu to select, load or save a characters set. Use "Print" in "File" menu to print your character pattern.

I.C.V.G.M. v3

by Daniel Bienvenu

This special version of my best graphic editor software is only for screen mode 2 (aka Graphic Mode II), and limited to one character set. It is not an update version of ICVGM v2. ICVGM v3 came after a suggestion received by email to simply create and edit multicolor characters.

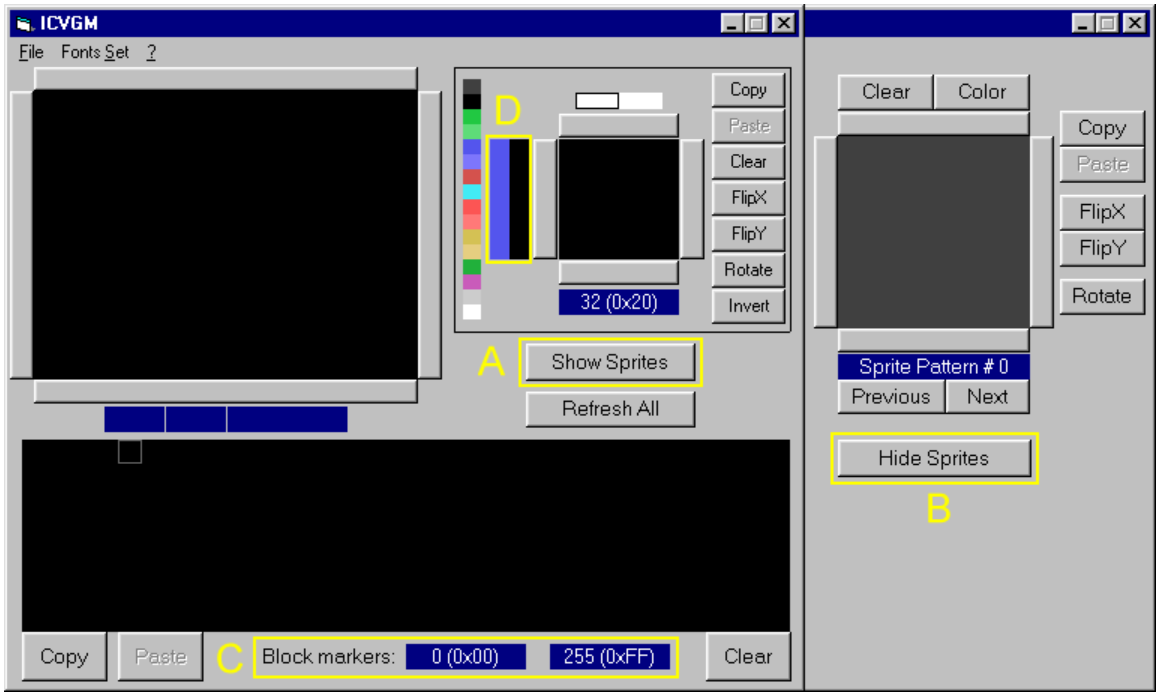


Figure 4 I.C.V.G.M. version 3 user interface

A. This button allow the user to show the sprites editor.
B. This button allow the user to hide the sprites editor.
C. This text show the block markers information. By clicking on the character set with the right mouse button, you set a block of markers to be able to copy, paste or clear character patterns.
D. Because this software version is based on screen mode 2, you can set colors for each line of a character pattern.

BMP2PP

by Marcel de Kogel

BMP2PP convert BMP pictures and also pictures from the clipboard into a valid ColecoVision format named PowerPaint. You can also convert a PowerPaint picture into a BMP file. Adjust the parameters to convert as well as possible your bitmap picture. Go in the File menu to save the result into a PowerPaint file ".pp". If you need to edit some pixels on the converted picture, I suggest you to use CVPAINT. Otherwise you can only use PP2C or PP2ASM to convert and compress PowerPaint files into data for your ColecoVision projects.

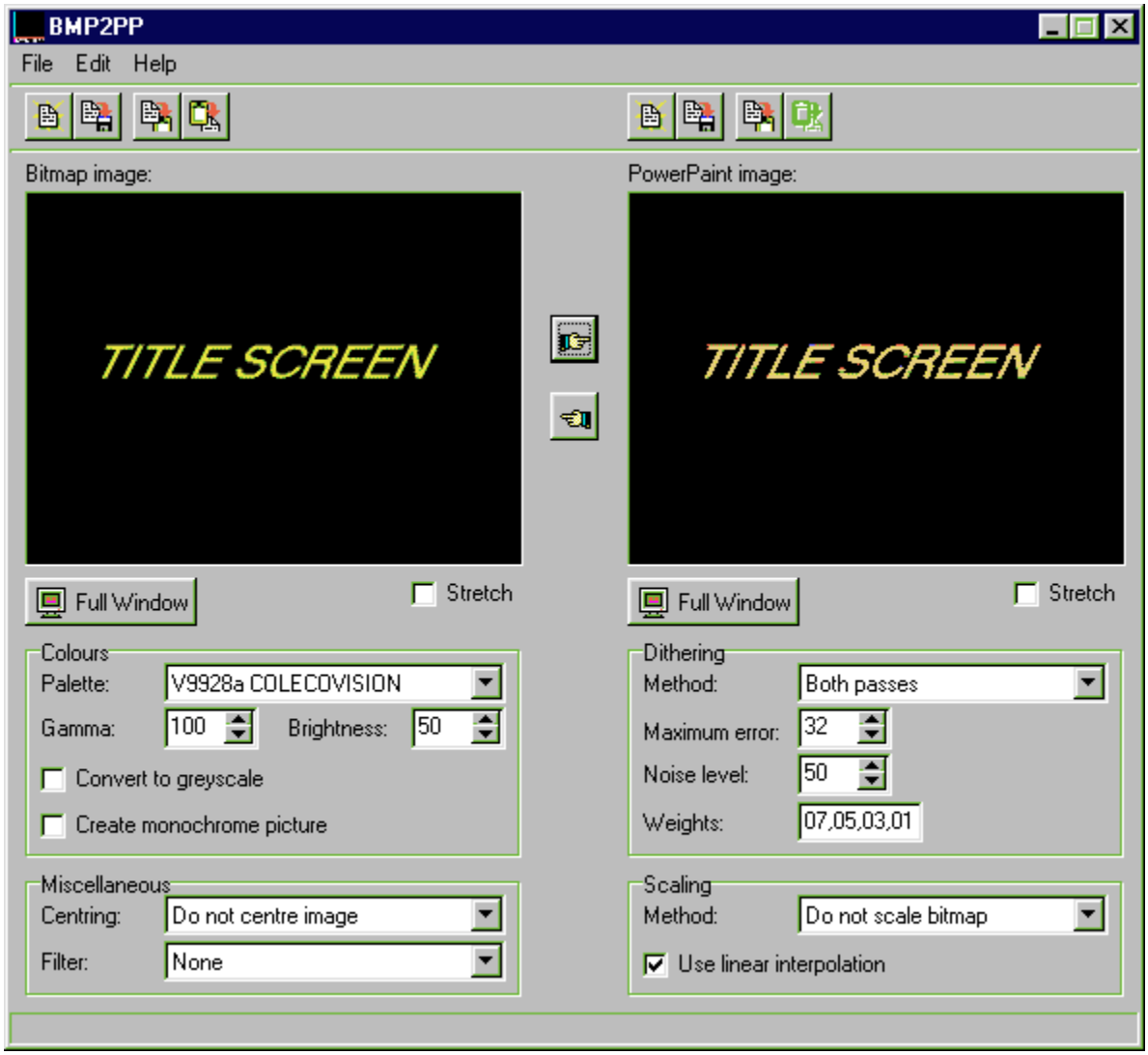


Figure 5 BMP2PP user interface

PP2C and PP2ASM

by Daniel Bienvenu

PP2C and PP2ASM are useful to quickly convert and compress (RLE encoding based on Marcel's ColecoVision library) PowerPaint pictures. Use PP2C to convert your picture into C format, and PP2ASM to convert into [T]ASM format. PP2C and PP2ASM are also integrated in CVPAINT tool. Using RLE compressed data allow to add more than one picture in your ColecoVision projects without taking too much memory space (ROM space).

First of all, (figure 6) you need to double-click on the PowerPaint file ".pp" you want. After seeing the picture in the preview zone, click OK to continue. After, (figure 7) write or select the file name to export data. And finally, (figure 8) write the name of this generated data table.

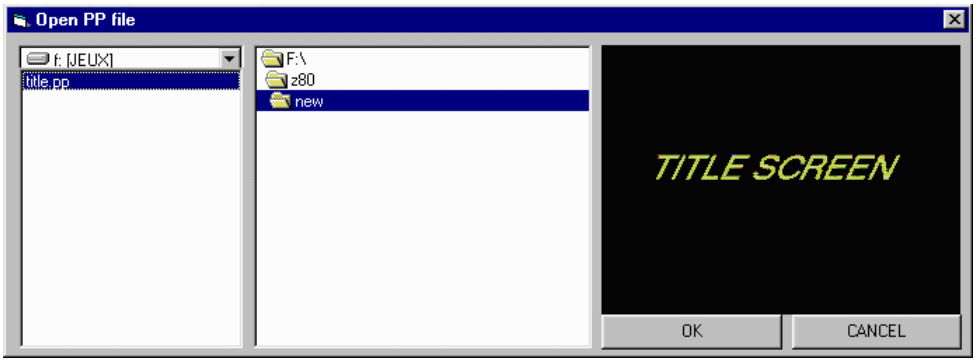


Figure 6 PP2C - Open PP file window

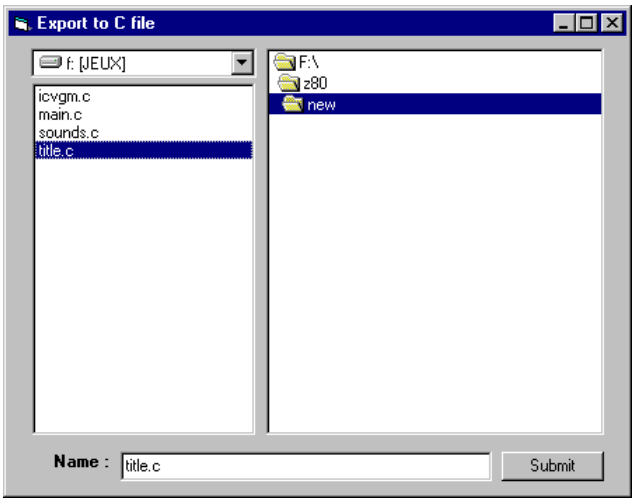


Figure 7 PP2C - Export to C file window

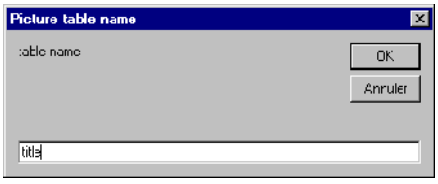


Figure 8 PP2C - Picture table name window

CVPAINT

by Daniel Bienvenu

CVPAINT still a beta version. We will not talk too much about this software. The reason I programmed this tool is to avoid using many times BMP2PP to convert pictures just because I add some pixels on it. CVPAINT is (for now) the only graphic editor tool for Windows who respects the limit of the ColecoVision graphic chip. You can draw pixels per pixels, zoom and use a grid but you can't do a COPY-PASTE (not well programmed).

You can also import pictures from other formats like the one used for the ZX Spectrum (SCR and MLT). I added also the "Export to C" and "Export to ASM" routines from PP2C and PP2ASM to avoid using too many tools.

The "Get" button is a primitive "copy" function but it doesn't work well if you don't know how to use it. This is how it works. When you click on the "Get" button, the name change for "Select" and you have to select an area on screen to copy. When it's done, press the "Select" button. It creates a new layer at the top left of the screen. Use your mouse to drag this layer on screen where you want to copy it. Use the right click button on your mouse to merge the layer on screen. Don't use "Zoom" or "grid" button otherwise the copy can't be done. There is no "Back" button so be careful.

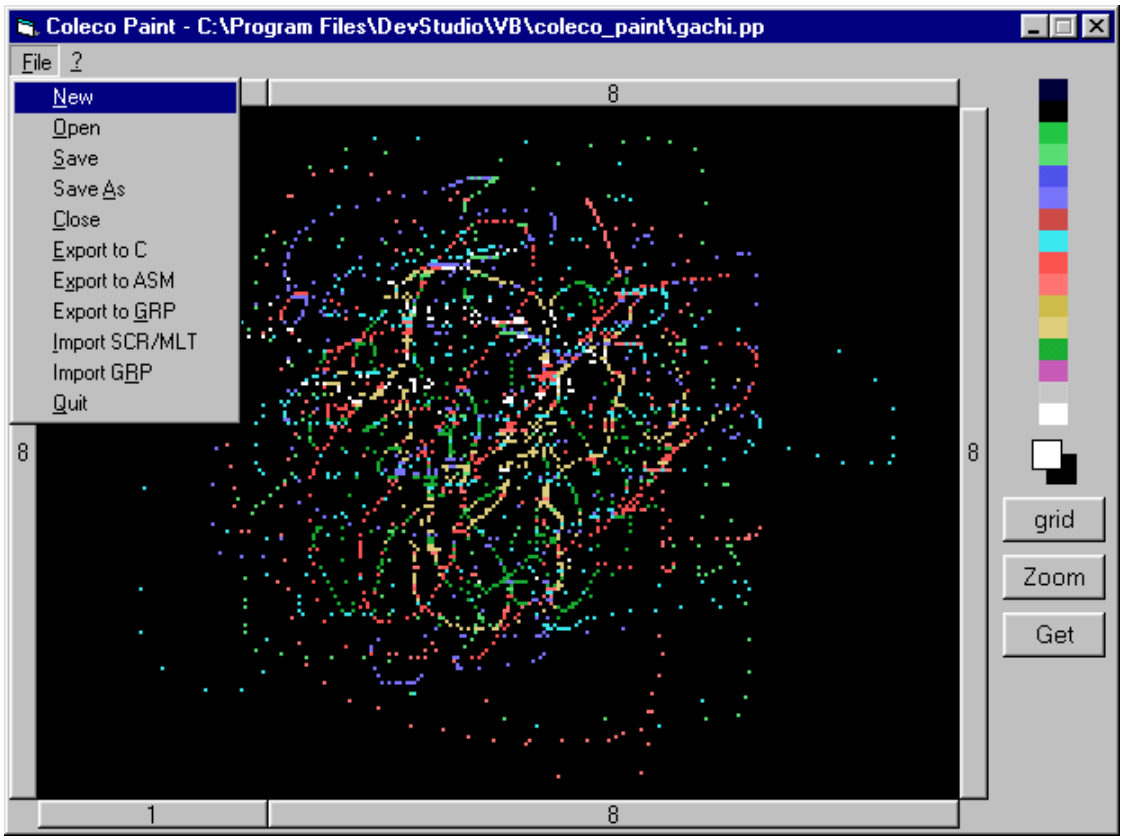


Figure 9 CVPAINT user interface with "File" menu opened

CCI - Coleco Compiler Interface

This tool uses the Hi-Tech C compiler (for CP/M) and 22NICE CP/M emulator (for DOS). This tool helps you to compile and link your project without editing batch files and writing command lines.

How to use CCI?

Before using this software you must copy it in your Coleco project directory.

1. To use the "Compile" button, you must select a file in the file list box first.
2. Use the "Compile All" button to compile your entire project in one step.
3. Select the libraries you need by checked the appropriated checkboxes.
4. If you use a french version of Windows NT, 2000 or XP, then you may need to check the "French Windows NT" checkbox.
5. After selecting the right libraries, use the "Link" button to link them with your project.
6. If every things ok, you will see a valid "result.rom" file in your project directory with a "map.txt" file.
7. The "Run" button will start the "result.rom" file with VirtualColeco emulator. If you want to use another emulator, add a batch file named "run.bat" in your project directory. This batch file must have the right instructions to play "result.rom" file with another emulator.

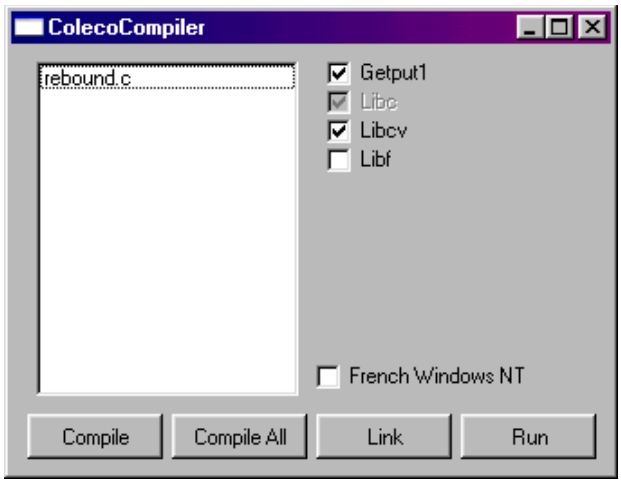


Figure 10 CCI user interface

See a compiling example in the programs section in this document.

LIBRARY: COLECO

The ColecoVision library made by Marcel de Kogel is programmed in ASM to be used with Hi-Tech C compiler. This library is divided in two parts: "crtcv.obj" and "cvlib.lib". "crtcv.obj" is the header of the ROM started at address 8000. "cvlib.lib" is the ColecoVision library itself with many useful routines.

ROUTINES IN COLECO LIBRARY

The following routines are the most important ones available in the ColecoVision library.

rle2ram

```
/* RLE decode specified data to specified RAM area. Returns pointer to first unused free */
void *rle2ram(void *rledata, void *dest);
```

rle2vram

```
NMI*
/* RLE decode specified data to specified VRAM area. Returns pointer to first unused free */
void *rle2vram(void *rledata, unsigned dest);
```

put_vram

```
NMI*
/* Upload RAM to VRAM. count should be a multiple of 256 */
void put_vram (unsigned offset,void *ptr,unsigned count);
```

get_vram

```
NMI*
/* Get array of VRAM bytes. count should be a multiple of 256 */
void get_vram (unsigned offset,void *ptr,unsigned count);
```

fill_vram

```
NMI*
/* Fill VRAM area with specified value */
void fill_vram (unsigned offset,byte value,unsigned count);
```

put_vram_ex

```
NMI*
/* Upload RAM to VRAM, applying specified AND and XOR masks */
void put_vram_pattern (unsigned offset,void *ptr, unsigned count, byte and_mask, byte xor_mask);
```

put_vram_pattern

```
NMI*
/* Upload pattern to VRAM */
void put_vram_pattern (unsigned offset,void *pattern, byte psize,unsigned count);
```

set_default_name_table

NMI*
/* Upload default name table */
void set_default_name_table (unsigned offset);

Note: set_default_name_table is used in bitmap mode to fill the screen with 3 set of characters from 00 to FF. This is necessary to show a bitmap picture on screen by using all the characters patterns.

vdp_out

NMI*
/* Write specified VDP register */
void vdp_out (byte reg,byte val);

screen_on

NMI*
/* Turn display on */
void screen_on (void);

screen_off

NMI*
/* Turn display off */
void screen_off (void);

Note: "screen_on" and "screen_off" are used to display or not the screen. You can update the screen without showing the modification unless it's done.

disable_nmi

/* Disable NMI */
void disable_nmi (void);

enable_nmi

/* Enable NMI */
void enable_nmi (void);

Note: Use *disable_nmi* to stop NMI interruptions when you do some important update then use *enable_nmi* to restart NMI interruption. In Getput-1 library, all the "print" routines already use *disable_nmi* and *enable_nmi* except *put_char*, *get_char*, *put_frame* and *get_bkgrnd*.

update_sound

NMI*
/* Check for new sound events */
void update_sound (void);

start_sound

```
/* Setup a sound channel. Returns pointer to sound channel allocated */
void *start_sound (void *data,byte priority);
```

stop_sound

```
/* Stop specified sound channel */
void stop_sound (void *channel);
```

sound_on

```
/* Enable sound output */
void sound_on (void);
```

sound_off

```
/* Disable sound output */
void sound_off (void);
```

delay

```
/* wait specified VBLANKs */
void delay (unsigned count);
```

Note: "delay" is necessary to slowdown the execution. Otherwise, the gameplay is too fast, unplayable.

get_random

```
/* Fast random routines. Return a byte value between 0 and 255. */
byte get_random (void);
```

upload_ascii

```
NMI*
/* Upload ASCII characters */
void upload_ascii (byte first,byte count,unsigned offset,byte flags);

#define NORMAL          0
#define ITALIC  1
#define BOLD    2
#define BOLD_ITALIC     (ITALIC | BOLD)
```

Note: If you use the Getput-1 library, you must use "upload_default_ascii" routine.

utoa

```
/* Convert unsigned integer to ASCII. Leading zeros are put in buffer */
void utoa (unsigned value, void *buffer, byte null_character);
```

Note: If you use Getput-1 library and you don't know well how to use *utoa*, you must use *str*.

sprites struct and table

```
/* sprite_t: position Y,X, pattern number and colour code */
typedef struct
{
 byte y;
 byte x;
 byte pattern;
 byte colour;
} sprite_t;
extern sprite_t sprites[64];
```

update_sprites

NMI*

```
/* Upload sprites to VRAM. Arguments are maximum number of sprites to upload
   (normally 32) and the sprite attribute table offset */
void update_sprites (byte numsprites,unsigned sprtab);
```

Note: If you use Getput-1 library and you don't know well how to use *update_sprites*, you must use *updatesprites*.

check_collision

```
/* Check collision between two sprites (between areas). Sizes decode as follows:
 lobyte - first pixel set
 hibyte - number of pixels set */
byte check_collision (sprite_t *sprite1,sprite_t *sprite2,
                      unsigned sprite1_size_hor,unsigned sprite1_size_vert,
                      unsigned sprite2_size_hor,unsigned sprite2_size_vert);
```

NMI* : Be aware to not use these functions when NMI interrupt may occurs. Use them directly in the nmi routine or use disable_nmi.

LIBRARY: COLECO OPTIMIZED FOR 4K

This modified version of Marcel's ColecoVision library is optimized to produce smaller games by using ColecoVision BIOS functions and no support for : spinner, 3rd fire button, 4th fire button and less sprites.

NEW ROUTINES IN THIS COLECO LIBRARY

The following routines are the new routines implanted in the ColecoVision library for this special version.

play_sound

/* Play the Nth sound data specified in the sound table "snd_table". play_sound(1) plays the 1st sound */
void play_sound (byte number);

stop_sound

/* Stop the Nth sound data specified in the sound table "snd_table" if playing */
void stop_sound (byte number);

reflect_vertical

NMI*
/* Reflect N times 8 bytes of pattern around the vertical axis */
/* table_code : 0 = sprite_name, 1 = sprite_generator, 2 = name (screen), 3 = pattern, 4 = color */
void reflect_vertical (byte table_code, unsigned source, unsigned destination, unsigned count);

reflect_horizontal

NMI*
/* Reflect N times 8 bytes of pattern around the horizontal axis */
/* table_code : 0 = sprite_name, 1 = sprite_generator, 2 = name (screen), 3 = pattern, 4 = color */
void reflect_horizontal (byte table_code, unsigned source, unsigned destination, unsigned count);

rotate_90

NMI*
/* Rotate clockwise N times 8 bytes of pattern */
/* table_code : 0 = sprite_name, 1 = sprite_generator, 2 = name (screen), 3 = pattern, 4 = color */
void rotate_90 (byte table_code, unsigned source, unsigned destination, unsigned count);

NMI* : Be aware to not use these functions when NMI interrupt may occurs. Use them directly in the nmi routine or use disable_nmi.

LIBRARY: GETPUT

This is my own toolbox programmed in C to be used in any ColecoVision game projects. This library is based on screen mode 0 or 2 and Cosmo Challenge source code.

GETPUT

The first version of "getput" made in year 2000 has only 6 routines.

cls

NMI*
```
/* cls is a CLEAR SCREEN routine */
void cls(void);
```

Example:
```
cls();
```

get_char

NMI*
```
/* get_char: This routine return the character value (in video memory) at location X,Y */
char get_char (byte x,byte y);
```

Example:
```
char c;
c = get_char (15,11);
```

put_char

NMI*
```
/* put_char: This routine put a character (in video memory) at location X,Y */
void put_char (byte x,byte y,char s);
```

Example:
```
char c;
c = 'A';
put_char(15,11,c);
```

center_string

NMI*
```
/* center_string: This routine PRINT a string on screen in the middle of the line L */
void center_string (byte l,char *s);
```

Example:
```
center_string (11,"GAME  OVER");
```

print_at

NMI*

```
/* print_at: This routine PRINT a string on screen at location X,Y */
void print_at (byte x, byte y,char *s);
```

Example:

```
print_at (0,0,"HELLO WORLD");
```

pause

NMI**

```
/* pause: This routine waits for any pressed fire button on port#1 or port#2. */
void pause (void);
```

Example:

```
pause ();
```

NMI* : Be aware to not use these functions when NMI interrupt may occurs. Use them directly in the nmi function or use disable_nmi.

NMI** : These functions doesn't work if NMI is disabled. Use enable_nmi.

GETPUT 1

Extended version of "getput" made in years 2002-2004 has over 20 more routines. This new library is compatible with the original version "getput" but compiled in a real library format.

pause_delay

```
/* pause_delay: waits for a laps of time but can be interrupted by pressing a fire button */
void pause_delay(unsigned i);
```

Example:

```
pause_delay (200);
```

rnd

```
/* rnd: Return a random number between min and max [0, 65535] */
unsigned rnd(unsigned min, unsigned max);
```

Example:

```
unsigned number = rnd(1,1000);
```

rnd_byte

```
/* rnd_byte: Return a random number between min and max [0, 255] */
byte rnd_byte(byte min, byte max);
```

Example:

```
byte number = rnd_byte(1,6);          /* dice 6 faces*/
```

str

```
/* str: Convert an unsigned value into a string  */
char *str(unsigned value);
```

Example:

```
print_at (10,10,str (100));
```

show_picture

NMI*

```
/* show_picture: Show a picture (encoded RLE) on screen */
void show_picture(void *picture);
```

Example:

```
show_picture (title);
```

screen_mode_2_bitmap

NMI*
/* screen_mode_2_bitmap: Set the screen mode 2 to show a bitmap picture */
void screen_mode_2_bitmap(void);

Example:

```
screen_mode_2_bitmap ();
```

screen_mode_2_text

NMI*
/* screen_mode_2_text: Set the screen mode 2 to print text on screen */
void screen_mode_2_text(void);

Example:

```
screen_mode_2_text ();
```

upload_default_ascii

NMI*
/* upload_default_ascii: Set the default character set */
void upload_default_ascii(byte flags);

Example:

```
upload_default_ascii (BOLD);
```

paper

NMI*
/* paper: Set the background color by using the VDP register 7 */
void paper(byte color);

Example:

```
Paper (4); /* A dark blue background color */
```

load_color

NMI*
/* load_color: Set the color of the character set (ICVGM) */
void load_color(byte *color);

Example:

```
load_color (color);
```

load_namerle

NMI*
/* load_namerle: Show a screen (rle encoded) on screen (ICVGM) */
void load_namerle(byte *namerle);

Example:

```
load_namerle (namerle);
```

load_patternrle

NMI*

```
/* load_patternrle: Set the characters pattern (ICVGM) */
void load_patternrle(byte *patternrle);
```

Example:

```
load_patternrle (patternrle);
```

load_spatternrle

NMI*

```
/* load_spatternrle: Set the sprites pattern (ICVGM) */
void load_spatternrle(byte *spatternrle);
```

Example:

```
load_spatternrle (spatternrle);
```

change_pattern

NMI*

```
/* change_pattern: update N characters starting with the character c */
void change_pattern(byte c, byte *pattern, byte N);
```

Example:

```
byte pattern[] = {1,2,4,8,16,32,64,128};
change_pattern ('A',pattern,1);
```

change_spattern

NMI*

```
/* change_spattern: update N sprites patterns, starting with sprite pattern number s */
/* N = number of 8x8 sprites patterns to update OR 4x number of 16x16 sprites pattern */
/* If sprites are sized 16x16, s must be 4x the number of the first sprite pattern to update */
void change_spattern(byte s, byte *pattern, byte N);
```

Example:

```
byte spattern[] = {1,2,4,8,16,32,64,128, 128,64,32,16,8,4,2,1, 128,64,32,16,8,4,2,1, 1,2,4,8,16,32,64,128};
/* Can be one big 16x16 sprite pattern OR 4 little 8x8 sprite patterns */
change_spattern (0,spattern,4);
/* Note: s = (sprite number 0) x 4 = 0, N=(1 "big 16x16 pattern") x 4 = 4 */
```

change_color

NMI*

```
/* change_color: update N characters colour (starting with the character c) with color data  */
void change_color(byte c, byte *color, byte N);
```

Example:

```
byte pattern_font[] = {0xA0, 0x70, 0x80};
/* change the color of characters A (in yellow),B (in cyan) and C (in red)*/
change_color ('A',pattern_font,3);
```

fill_color

NMI*

```
/* fill_color: fill the N characters (starting with the character c) in a color */
void fill_color(byte c, byte color, byte n);
```

Example:

```
        /* The characters D,E,F and G will be in green color */
        fill_color ('D', 0x20 ,4);
```

change_multicolor

NMI*

```
/* change_multicolor: update the colors of the character c */
void change_multicolor(byte c, byte *color);
```

Example:

```
        byte red_yellow_font [] = {0x60,0x80,0x90,0xA0,0x90,0x80,0x60,0x60};
        /* update the colors of the character H with the red-yellow color font */
        change_multicolor ('H',red_yellow_font);
```

change_multicolor_pattern

NMI*

```
/* change_multicolor_pattern: update the colors of the N characters (c and the following) */
void change_multicolor_pattern(byte c, byte *color, byte n);
```

Example:

```
        byte green_yellow_font [] = {0xC0,0x20,0x30,0xA0,0x30,0x20,0xC0,0xC0};
        /* update the colors of the characters I and J with the green-yellow color font */
        change_multicolor ('I',green_yellow_font,2);
```

choice_keypad_1 and choice_keypad_2

```
/* Wait until a key is pressed on keypad#1 or keypad#2 between min and max */
byte choice_keypad_1(byte min, byte max);
byte choice_keypad_2(byte min, byte max);
```

Example:

```
        byte choice;
        /* use keypad#1 to select a number between 1,2,3 and 4 */
        choice = choice_keypad_1 (1,4);
        /* use keypad#2 to select a number between 5,6,7 and 8 */
        choice = choice_keypad_2 (5,8);
```

updatesprites

NMI*

```
/* updatesprites: Update N(count) sprites data in video memory with the sprites table */
void updatesprites(byte first, byte count)
```

Example:

```
        /* Update 32 sprites [0,31] */
        updatesprites(0,32);
```

sprites_simple

NMI*
/* sprites_simple: To set sprites pixels at the normal size */
void sprites_simple(void);

sprites_double

NMI*
/* sprites_double: To set sprites pixels two times bigger than normal */
void sprites_double(void);

sprites_8x8

NMI*
/* sprites_8x8: To set the sprites to be 8x8 pixels */
void sprites_8x8(void);

sprites_16x16

NMI*
/* sprites_16x16: To set the sprites to be 16x16 pixels */
void sprites_16x16(void);

Set of "AND" masks for the joystick: UP, DOWN, LEFT, RIGHT and FIREs

/* A set of CONSTANTS to help with the joystick return value (AND MASK) */
/* These constants are named : UP, RIGHT, DOWN, LEFT, FIRE1, FIRE2, FIRE3, FIRE4 */

Example:

```
If (joypad_1&LEFT) /* joystick#1 go left? */
If (joypad_2&RIGHT) /* joystick#2 go right? */
```

wipe_off_down

/* This is a special effect to clean smoothly the screen in bitmap mode from top to bottom */
void wipe_off_down (void);

wipe_off_up

/* This is a special effect to clean smoothly the screen in bitmap mode from bottom to top */
void wipe_off_up (void);

NMI* : Be aware to not use these functions when NMI interrupt may occurs. Use them directly in the nmi function or use disable_nmi.

Note: Before GETPUT version 1.1, you needed to enable the NMI interrupts to properly use pause_delay, wipe_off_down and wipe_off_up functions. Now, pause_delay enables NMI by itself, and the wipe_off_down and wipe_off_up functions are working whatever if the NMI is enabled or not.

New routines in Getput1 library in years 2003-2004

play_dsound

NMI*

```
/* This routine plays a digital sound ('sound_data') at a specified speed ('step') */
void play_dsound (byte *sound_data, byte step);
```

Note: play_dsound was originally done in another library named "dsound". Because of a weird compiler bug who doesn't allow to use dsound and getput library togheter, I had to put play_dsound in the Getput-1 library.

put_frame

NMI*

```
/* Useful routine from the original ColecoVision bios to put a tile of many characters on screen in one single routine call. */
void put_frame(void *table, unsigned char x, unsigned char y, unsigned char width, unsigned char height);
```

get_bkgrnd

NMI*

```
/* This routine allow you to get the values of a tile of characters on screen. */
void get_bkgrnd(void *table, unsigned char x, unsigned char y, unsigned char width, unsigned char height);
```

Note: Using put_frame and get_bkgrnd in the same program may be fun to put and many move a "window" on screen.

load_colorrle

NMI*

```
/* loadcolorle: Set the multicolor patterns (ICVGM v3+) */
void load_colorrle(void *colorrle);
```

strlen

```
/* This function came from the C library but added in Getput library to avoid the boring warning message : "declared implicite int" */
byte strlen(void *table);
```

put_frame0

NMI*

```
/* This routine is more faster and smaller than the original put_frame function because it doesn't allow printing whole or a part of a frame (tile) outside the screen. */
void put_frame0(void *table, unsigned char x, unsigned char y, unsigned char width, unsigned char height);
```

screen

NMI*

```
/* This routine set two screen tables address (one to be showed, one to be updated). See swap_screen function. */
void screen (unsigned screen_table1_offset, unsigned screen_table2_offset);
```

Note: This function is to be used with pre-defined offsets named "name_table1" and " name_table2".

```
#define  name_table1       0x1800
#define  name_table2       0x1c00
```

swap_screen

NMI*

```
/* This routine shows the hidden screen and hide the showed screen set by screen function. */
void swap_screen (void);
```

put_at

NMI*

```
/* This routine print a part of an array of bytes or characters on screen at location X,Y */
void print_at (byte x, byte y,char *s, byte size);
```

Example:

```
        put_at (0,0,"HELLO WORLD",5);
```

Result:

```
        HELLO
```

fill_at

NMI*

```
/* This routine print n times a character on screen at location X,Y */
void print_at (byte x, byte y,char s, unsigned n);
```

NMI* : Be aware to not use these functions when NMI interrupt may occurs. Use them directly in the nmi function or use disable_nmi.

Extra routines in Getput library version 1.1

New version of "getput" started in year 2005. This new library is done entirely in assembler code and optimized to produce even more smaller rom file. Also new functions are added from the previous “getput” library version.

score_reset

```
/* Initialize a score type variable to zero */
void score_reset (score_t *score);
```

score_add

```
/* Add value to a score type variable */
void score_add (score_t *score, unsigned value);
```

score_str

```
/* Convert to string a score type value. Valid number of digits to print on screen is from 1 to 9 */
char *score_str(score_t *score, byte number_of_digits);
```

score_cmp_lt

```
/* Return true if the first score value is less than the second one */
int score_cmp_lt(score_t *score1,score_t *score2);
```

score_cmp_gt

```
/* Return true if the first score type value is greater than the second one */
int score_cmp_gt(score_t *score1,score_t *score2);
```

score_cmp_equ

```
/* Return true if the first score value is equal to the second one */
int score_cmp_equ(score_t *score1,score_t *score2);
```

intdiv256

```
/* Return a signed integer value divided by 256 */
int intdiv256(int value);
```

utoa0

```
/* Convert an unsigned value to characters ('0' to '9') */
/* Same as the utoa function from the Coleco library but already set with '0' as char for the zeros */
void utoa0(unsigned value,void *buffer);
```

load_ascii

NMI*

```
/* Call ColecoVision bios LOAD_ASCII function */
void load_ascii();
```

rlej2vram

NMI*

```
/* Same as rle2vram but with a little twist */
/* A joker code 00 in RLE data results in a “write no data here” */
void *rlej2vram (void *rledata,unsigned offset);
```

NMI* : Be aware to not use these functions when NMI interrupt may occurs. Use them directly in the nmi function or use disable_nmi.

GETPUT-1 VIDEO MEMORY MAP

Screen Mode 2 Text - Video Memory Map

Start Address	*End Address*	*Table Name*	*Information*
0000	07FF	CHRGEN	Characters Pattern (charset)
0800	17FF	-	Free
1800	1AFF	CHRTAB	Characters on Screen (NAME table)
1B00	1BFF	SPRTAB	Sprites Table (y,x,pattern,colour)
1C00	1FFF	-	Free (reserved for the swap screen functions)
2000	27FF	COLTAB	Characters Color Pattern
2800	37FF	-	Free
3800	3FFF	SPRGEN	Sprites Pattern

Screen Mode 2 Bitmap - Video Memory Map

Start Address	*End Address*	*Table Name*	*Information*
0000	17FF	CHRGEN	Screen Graphic Pattern
1800	1AFF	CHRTAB	Initialised with set_default_name_table
1B00	1BFF	SPRTAB	Sprites Table (y,x,pattern,colour)
1C00	1FFF	-	Free (reserved for the swap screen functions)
2000	37FF	COLTAB	Screen Graphic Colors
3800	3FFF	SPRGEN	Sprites Pattern

LIBRARY: C

Useful routines from the C library.

memcpy

```
/* Copy data from one memory location to another memory location (in RAM of course). */
int memcpy(void *destination, void *source, unsigned number_of_byte_to_copy);
```

Example:

```
/* This command can be used to initialise the table "sprites" with data in ROM. */
memcpy(sprites, sprites_init, sizeof(sprites_init));
```

memset

```
/* Fill-up ram table. Useful to initialise a ram table. */
int memset(void *table, unsigned number, byte value);
```

Example:

```
/* This command set all bytes to 0 in the ram table. */
memset(table,sizeof(table),0);
```

sizeof

```
/* Return the table size (in byte); in getput, it used to print (center) a string on screen. */
int sizeof(*table);
```

switch case

```
/* Return the table size (in byte); in getput, it used to print (center) a string on screen. */
switch (choice) {
        case 1: choice_one(); break;
        case 2: choice_two(); break;
        default: not_valid(); break;
        }
```

SHOW BITMAP PICTURE WITHOUT GETPUT 1

The following C code is used to show a bitmap title screen.
Remarks in C are between '/*' and '*/'. The '//' is not an ANSI C standard remark syntax but used frequently in C++ language.

```
/* this is the header file (.h) for the Coleco library by Marcel de Kogel */
#include <coleco.h>

/* Important Video Memory Locations based on VDP control registers values */
#define chrgen 0x0000
#define coltab 0x2000
#define chrtab 0x1800

/* title is the name of the bitmap title screen table generated with PP2C in another C file */
extern byte title[];

/* The SHOOT sound from Cosmo Challenge */
static byte shoot_sound[]=
{       1,
        0xf8,0xe4, 1,0xf2, 1,0xe4, 1,0x63,0x02,0x01,0xe5,
        1,0xe4, 1,0xe5, 1,0xe4, 1,0xe5, 1,0xe4, 1,0xe5, 1,0xe4, 5,
        0,0,0};

/* The NMI routine: update sound at every vertical retrace. 60Hz (NTSC) or 50Hz (PAL) */
void nmi(void) { update_sound (); }

/* To setup the graphic screen mode 2 */
void screen_mode_2_bitmap(void)
{
        /* screen mode 2 */
        vdp_out (0,2);
        /* set video memory address for chrgen and colortable */
        vdp_out (3,0xff);  vdp_out (4,0x03);
        /* fill screen with characters 00 to FF three times */
        set_default_name_table (chrtab);
        /* clear chrgen and color table */
        fill_vram(chrgen,0x00,0x1800);
        fill_vram(coltab,0x00,0x1800);
        /* setup sprites and video memory size and enable NMI calls*/
        vdp_out(1,0xe2);
}
```

```
/* By using tools like BMP2PP by Marcel de Kogel and PP2C by Daniel Bienvenu (me), you can do a bitmap title screen without any problem. The title screen will be RLE encoded so you can use the following lines to show the title screen. */

/* To show a picture on screen */
void show_picture(void *picture)
{
        /* turn display off */
        screen_off ();
        /* Upload picture */
        rle2vram (rle2vram(picture,coltab),chrgen);
        /* turn display on */
        screen_on ();
}

/* The "main" routine to show the title screen can be something like this: */
void main(void)
{
        /* init graphic mode 2 */
        screen_mode_2_bitmap();
        /* show title screen */
        show_picture(title);
        /* enable NMI calls*/
        enable_nmi ();
        /* play sound SHOOT with priority 1 */
        start_sound(shoot_sound,1);
        /* infinite loop: constant relational expression (warning) */
        while(1);
}
```

There is no need to use disable_nmi before screen_mode_2_bitmap because the Coleco library disabled NMI interrupts in the Coleco starting code (crtcv.obj) before calling the main function.

To avoid using headers, there is a simple rule to respect. If a routine needs another one to run properly, the needed routine must be programmed or identified (#include ".h") before in the code. Otherwise, the compiler will not see the dependencies and the compiler will refuse to compile properly. Another solution is to add (functions) headers at the top of the C file just before all the functions.

SHOW BITMAP PICTURE WITH GETPUT 1

The following C code is used to show a bitmap title screen by using getput library routines.

```
/* This is the header files (.h) to include in the project for the Coleco library by Marcel de Kogel  and the Getput 1 library by Daniel Bienvenu */
#include <coleco.h>
#include <getput1.h>

/* title is the name of the bitmap title screen table generated with PP2C in another C file */
extern byte title[];

/* The SHOOT sound from Cosmo Challenge */
static byte shoot_sound[]=
{       1,
        0xf8,0xe4, 1,0xf2, 1,0xe4, 1,0x63,0x02,0x01,0xe5,
        1,0xe4, 1,0xe5, 1,0xe4, 1,0xe5, 1,0xe4, 1,0xe5, 1,0xe4, 5,
        0,0,0};

/* The NMI routine: update sound */
void nmi(void) { update_sound (); }

/* Do a title screen by using tools like BMP2PP by Marcel de Kogel and PP2C by Daniel Bienvenu (me). The title screen will be RLE encoded so you have to use show_picture */

/* The "main" routine to show the title screen can be something like this: */
void main(void)
{
        /* init graphic mode 2 */
        screen_mode_2_bitmap();
        /* show title screen */
        show_picture(title);
        /* enable NMI calls*/
        enable_nmi ();
        /* play noise sound: SHOOT with priority 1 */
        start_sound(shoot_sound,1);
        /* infinite loop: constant relational expression (warning) */
        while(1);
}
```

This new version of the "picture show" program with getput library is smaller than the first version. Getput 1 rules! It's the solution to program very quickly a ColecoVision project in C.

There is no disable_nmi before screen_mode_2_text because the Coleco library disabled NMI interrupts in the Coleco starting code (crtcv.obj) before calling the main function.

FACES - SPRITES DEMO

```
/* To test this demo, use ADAMEM with the following command line: */
/* cvem -vi 1 -if 60 -sprite 1 result.rom */

#include <coleco.h>
#include <getput1.h>

/* This flag is used to avoid VRAM corruption */
byte flag;

/* sprite pattern - laughing face */
byte sprite_pattern[]=
{       0,3,15,25,54,63,127,127, 127,112,48,56,30,15,3,0,
        0,224,248,204,182,254,255,255, 255,7,6,14,60,248,224,0};

void initialize(void)
{
        byte i;
        /* load 1 big 16x16 sprite pattern in video memory */
        change_spattern(0,sprite_pattern,4);
        /* init the sprites table */
        clear_sprites(0,64);
        for (i=1;i<6;i++)
        {
                sprites[i].y = i<<5;
                sprites[i].y--;
                sprites[i].x = i<<5;
                sprites[i].pattern = 0;
                sprites[i].colour = i<<1;
                sprites[i].colour += 3;
        }
        sprites_double();
}

void faces(void)
{
        byte k;

        initialize();

        /* enable NMI calls*/
        enable_nmi ();

        while(keypad_1 == 6) delay(1);

        /* Allow updating sprites on screen now */
        flag=1;

        while(keypad_1 != 6)
        {
                /* UPDATE SPRITES POSITION */
                k = keypad_1;
                if (k>0 && k<6)
```

```
                {
                        if (joypad_1&LEFT) sprites[k].x--;
                        if (joypad_1&RIGHT) sprites[k].x++;
                        if (joypad_1&UP) sprites[k].y--;
                        if (joypad_1&DOWN) sprites[k].y++;
                        if (sprites[k].y ==193 ) sprites[k].y = 241;
                        if (sprites[k].y ==240 ) sprites[k].y = 192;
                }
                /* TO SLOWDOWN THE ANIMATION */
                delay(1);
        }
}

void main(void)
{
        /* Don't update sprites on screen now */
        flag=0;
        /* Initialize the VDP to the screen mode 2 */
        screen_mode_2_text();
        /* Set the default ascii character set */
        upload_default_ascii (BOLD);
        /* Print an important message on screen */
        center_string (10,"HOLD A NUMBER BETWEEN 1 AND 5");
        center_string (11,"TO SELECT A SPRITE");
        center_string (13,"THEN USE THE JOYSTICK");
        center_string (14,"TO MOVE IT");
        center_string (16,"PRESS 6 TO RESET POSITIONS");
        /* Start the sprite demo program */
        faces();
}

/* NMI routine: update sprites at every vertical retrace */
void nmi(void)
{
        /* THIS FLAG IS NECESSARY TO AVOID VRAM CORRUPTION */
        if (flag) updatesprites(0,7);
}
```

To avoid a possible VRAM corruption when updating VRAM inside and outside the nmi function, it's a good idea to use a flag variable like this. However, this parcular program doesn't need a flag because all the VRAM calls outside the nmi function have been done before enabling the NMI interrupts. There is no disable_nmi before screen_mode_2_text because the Coleco library disabled NMI interrupts in the Coleco starting code (crtcv.obj) before calling the main function.

REBOUND

THE IDEA

The following text explains how to program a simple bouncing ball in characters.

First of all, we need to figure out how to move a character on screen. A character is not a sprite so you can't move a character on screen. You can create the illusion of a moving character. To create the illusion of a moving character, you have to erase the character you want to move by printing a "background" character over it and then print the same character at another location near the last location to create the illusion of a moving character. So, you need to know the exact location of the character to move at any moment. You will use the type "char" to use the negative and positive values: [-128,127].

```
/* To keep the information about the location of the ball */
char ball_x;
char ball_y;

(…)

ball_y = 0; /* top */
ball_x = 0; /* left */
```

Now, to create the movement of the bouncing ball, you have to change the location of the ball by adding the direction in X and Y to change the X and Y values of the ball.

```
/* To keep the information about the direction of the ball */
char ball_dx;
char ball_dy;

(…)

ball_dx = 1; /* moving to the right */
ball_dy = 1; /* moving to the bottom */

(…)

ball_x += ball_dx;
ball_y += ball_dy;
```

, to create the bouncing effect, you have to change the direction of the ball under some conditions. For this simple bouncing effect, you will use the border of the screen to rebound the ball on it. To know if the ball reach the border of the screen, you simply have to check the location of the ball you keep in memory. You can add a sound to indicate the bouncing condition is reached.

```
if ( ball_x == 0 || ball_x == 31) { ball_dx = -ball_dx; pop(); }
if ( ball_y == 0 || ball_y == 23) { ball_dy = -ball_dy; pop(); }
```

Finally, to see the ball on screen, you must add, at the strategic places, the routines to "erase" and "print" the ball at the X and Y location. You need to add a delay to slow down the animation.

```
/* Erase the ball at the actual location */
put_char ( ball_x, ball_y, 32 ); /* 32 is the value of the SPACE character */

/* Update the value of the location of the ball */
update_ball_location ();

/* Print the ball at the new location */
put_char ( ball_x, ball_y, 'O' );

/* Slowdown the animation */
delay(5);
```

To keep running, you have to do a LOOP with a condition to stop the animation like pressing the fire button #1 on joystick #1.

```
while (!(joypad_1&FIRE1))
{
        /* Move the ball on screen */
        (…)
}
```

THE PROGRAM

To keep it simple, you can use the capital letter 'O' to be the ball and the "put_char" routine in "getput" library to print and erase the ball on screen.

```
#include <coleco.h>
#include <getput1.h>

/* To keep the information about the location of the ball */
char ball_x;
char ball_y;

/* To keep the information about the direction of the ball */
char ball_dx;
char ball_dy;

/* A sound effect named "pop" is played when the ball reaches the border of the screen */
static byte pop_sound[] =
{
        0, 0x63,0xf,1,
        0x81,0xa0,0x90,1, 0x81,0x1c,0x97,1, 0x81,0x2c,0x9d,2, 0x81,0x32,0x9e,1,
        0,0,0
};

/* To start the "pop" sound */
static void pop (void)
{
         start_sound (pop_sound,2);
}

/* To initialize the location and direction of the ball */
static void initialize (void)
{
        ball_y = 0; /* top */
        ball_x = 0; /* left */
        ball_dx = 1; /* moving to the right */
        ball_dy = 1; /* moving to the bottom */
}

/* Change the direction of the ball when bouncing horizontally on the border of the screen */
static void bounce_on_walls_in_X (void)
{
        if ( ball_x == 0 ) { ball_dx = -ball_dx; pop(); }
        if ( ball_x == 31 ) { ball_dx = -ball_dx; pop(); }
}
```

```
/* Change the direction of the ball when bouncing vertically on the border of the screen */
static void bounce_on_walls_in_Y (void)
{
	if ( ball_y == 0 ) { ball_dy = -ball_dy; pop(); }
	if ( ball_y == 23 ) { ball_dy = -ball_dy; pop(); }
}

/* To update the location and direction of the ball. */
static void update_ball_location (void)
{
	ball_x += ball_dx;
	ball_y += ball_dy;
	bounce_on_walls_in_X();
	bounce_on_walls_in_Y();
}

/* This part of the program is the game engine and it's used to rebound a ball on screen */
static void bounce(void)
{
	/* Initialize the ball location and direction */
	initialize ();
	/* enable NMI calls*/
	enable_nmi ();
	/* The animation will stop when pressing on fire1 on joystick#1 */
	while (!(joypad_1&FIRE1))
	{
		/* disable NMI calls*/
		disable_nmi ();
		/* Erase the ball at the actual location */
		put_char ( ball_x, ball_y, 32 ); /* 32 is the value of the SPACE character */
		/* Update the value of the location of the ball */
		update_ball_location ();
		/* Print the ball at the new location */
		put_char ( ball_x, ball_y, 'O' );
		/* enable NMI calls*/
		enable_nmi ();
		/* Slowdown the animation */
		delay(5);
	}
	/* Exit the bouncing routine when the fire button will be released */
	while (joypad_1&FIRE1);
}

/* NMI routine: update sound at every vertical retrace */
void nmi(void) { update_sound(); }

void main(void)
{
	/* Initialize the VDP to the screen mode 2 */
	screen_mode_2_text();
	/* Set the default ascii character set */
	upload_default_ascii (BOLD);
	/* Start the bouncing ball program */
	bounce();
}
```

COMPILING

The rebound project must be in a sub-directory of the Hi-Tech C compiler. Name this sub-directory: "rebound". Create a new C file and write the rebound program. After writing the rebound program into a file named "rebound.c", you add the "cci.exe" program in the same sub-directory. If you don't see the extension of the files (".c" and ".exe"), there is a possibility that the rebound file you created didn't have the right extension. Note: CCI see only the files with the extension ".c" (for C files) and ".as" (for ASM files) in the current directory.

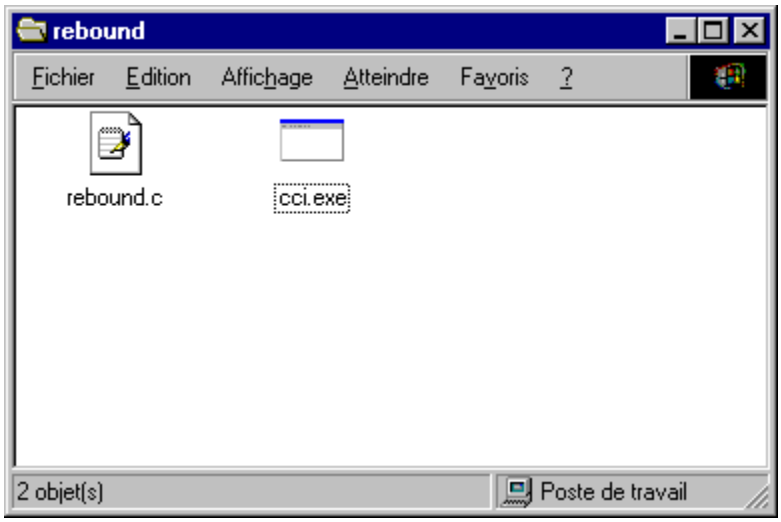


Figure 11 Rebound directory before compiling and linking the project

If the rebound C program is correctly done, the following steps will be so easy that you will not believe you are compiling a program. First, make sure that the checkbox Getput-1 is checked and the "rebound.c" file is listed in the file list box. If you are using a french version of Windows NT, XP or Windows 2000, you may have to check the "French Windows NT" checkbox.

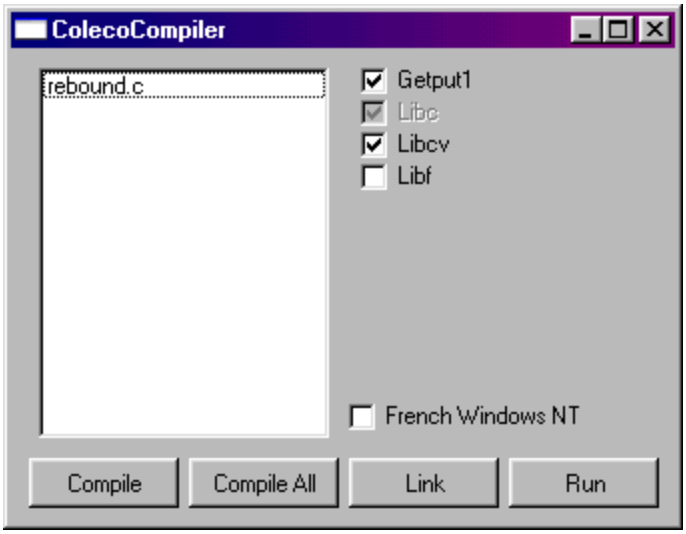


Figure 12 CCI running in Rebound sub-directory

Select the "rebound.c" file and click on the "Compile" button at the bottom. You can also simply click on "Compile All" button to compile all the files listed in the file list box. A popup DOS window will appear to run the 22NICE emulator. After you pressed the space bar to continue the execution of the 22NICE emulator, you will see the Hi-Tech C compiler compiling the Rebound program. Note: if you use the "Compile" button, after the compiling process, the execution will stop until you press a key. When the "rebound.c" file is compiled into a valid "rebound.obj" file, close the DOS Window if it's still opened. If the "rebound.obj" file is not a valid one (empty file, errors during the compiling process), it's because you make an error somewhere in your code. The error in your code is named a bug. A bug can be anything like not including the correct header files "#include <coleco.h>" and "#include <getput1.h>" or not using the correct routine name. There is no debug facility so you must be careful. To avoid doing so much errors when writing your code for the first time, try to write only one routine at a time and compile your code after each new version. To debug a code, you must use "/*" and "*/" (remarks) to "deactivate" the part of your code you think the bug is. Recompile your code and see if the bug is gone, if so, the bug is in the part of the code you placed in remarks.

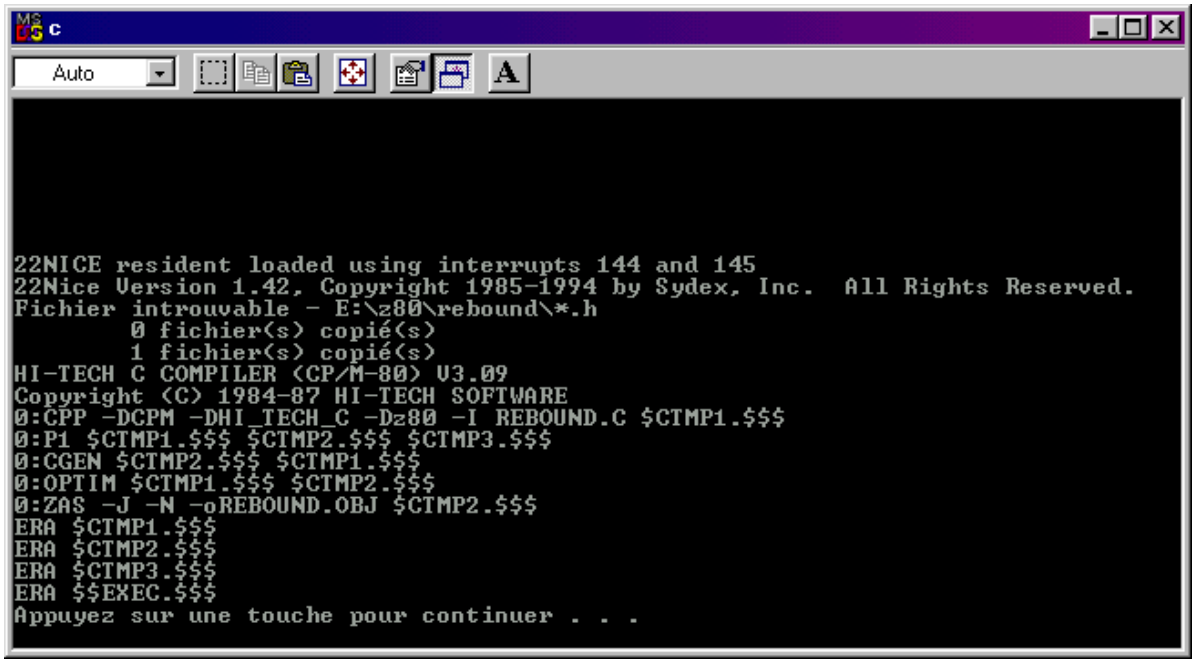


Figure 13 Compiling Rebound program without any error

After compiling the Rebound program, you have to link the "rebound.obj" file created by the compiler with the "coleco" and "getput" libraries to create a ROM file. If the Getput-1 checkbox is checked, click on the "Link" button at the bottom to start the 22NICE emulator in a popup DOS window. After pressing the space bar to continue the execution of the 22NICE emulator, you will see the word "LINK>". Simply do PASTE* in the popup DOS window to add the options for the linker. If the linker do right is job, you must find a valid "result.rom" file in the Rebound directory. If an error appear during the linking process, it's probably because you omit something like the nmi routine in your project or checked the Getput-1 checkbox.

*: If you don't know how to PASTE in a popup DOS window, you can use the following trick: right-click on the title bar of the popup DOS window to see a popup menu and then choose the "Edit" option in the popup menu to see a PASTE option.

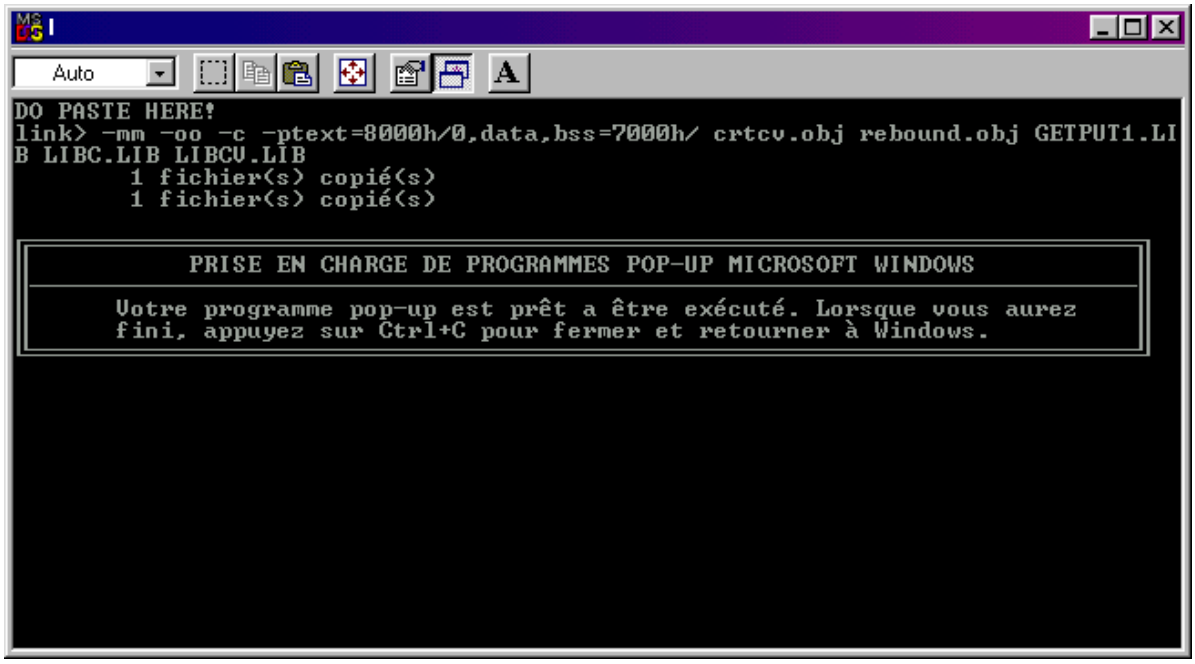


Figure 14 Linking Rebound program without any error

If a valid (not empty) "result.rom" file is created, simply click on the "Run" button to test Rebound with a ColecoVision emulator. If you didn't like the result, you may have to modify (calibrate) your code.

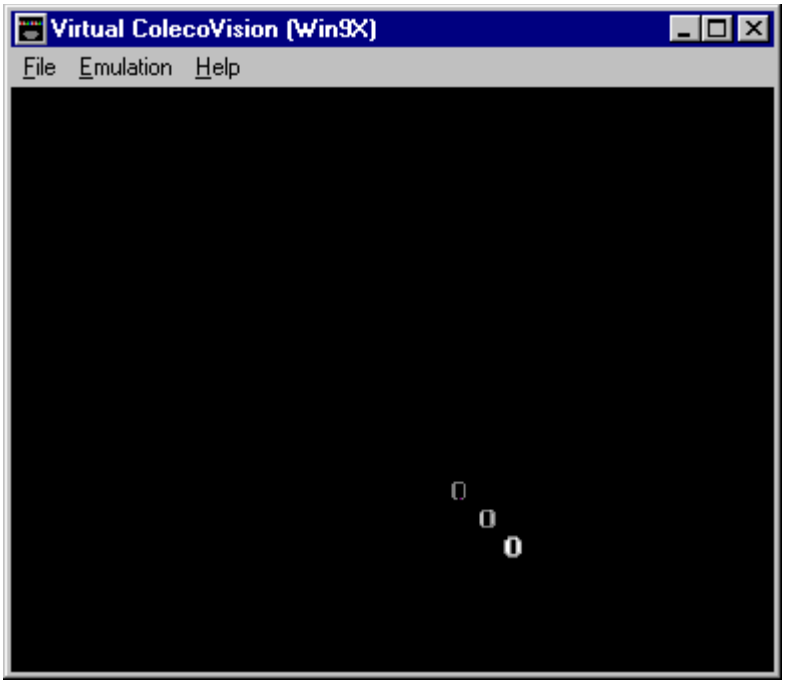


Figure 15 Running Rebound in the Virtual ColecoVision emulator for Windows

Now, if you check the Rebound directory you will find at least 6 files: "rebound.c", "cci.exe", "c.bat", "rebound.obj", "l.bat" and "result.rom". New versions of CCI add a file named "map.txt". The two batch files ("c.bat" and "l.bat") are generated by CCI to compile and link your project. You can erase these two batch files if you want.

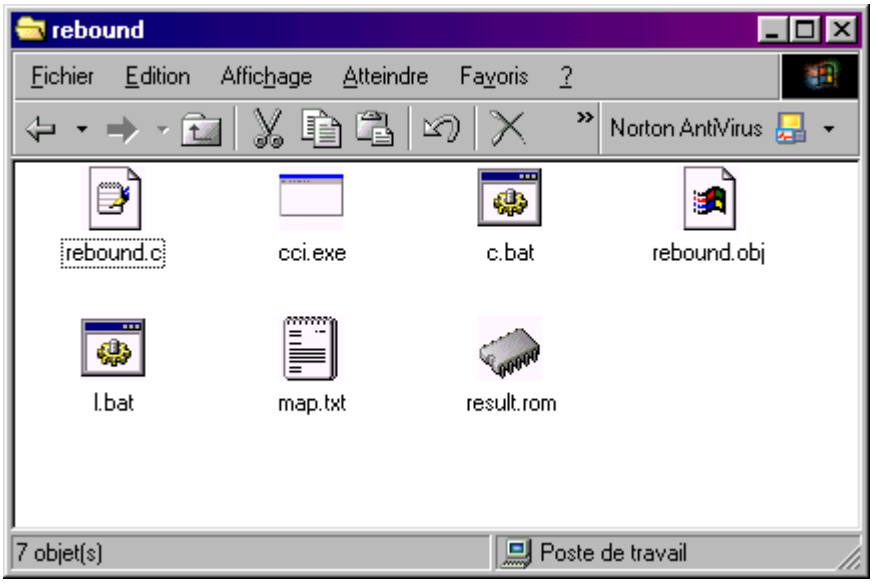


Figure 16 Rebound directory after compiling and linking the project

Congratulation! You compiled a ColecoVision project in C.

After June 2003, new CCI versions add a "map.txt" file in your project directory after linking. If you open it, you will see the memory map of your project: where are your code, your data and how much ram you use, etc.

The following table is only a small part of the memory map file.

TOTAL	Name	Link	Load	Length
	(abs)	0	0	0
	text	8000	0	75D
	data	875D	75D	2F
	bss	7000	78C	4E

Legend: "**text**" is for the ROM (header and code), "**data**" is for static data in ROM like text and graphics, and "**bss**" is for the RAM used by the ROM (doesn't include RAM used by the BIOS).

Note: Make sure that **bss length** is never bigger than 300 to avoid memory corruption. (see pages about memory) Nobody knows your code better than you and it's why you must try to figure out by yourself how to optimize your code.

STILL BUGY?

The compiler and the linker are not intelligent, so they can't find errors of logic you made in your code. These errors can be described like wrong instructions who make your project not do what supposed to do. This kind of bug is more difficult to find because it can be the result of a combination of instructions in your code or a problem with pointers (a programmer nightmare).

The programming kit came with the Virtual ColecoVision emulator for Windows but this emulator is not a good one to emulate perfectly the ColecoVision. My suggestion is to use ADAMEM and MESS to test your project. If your project works well with Virtual ColecoVision emulator but not with ADAMEM and MESS, maybe some instructions in your code are trying to write data in ROM (read only memory).

Look how easy a bug can be done.

```
# include <coleco.h>

/* "a" is a global variable (because not initialised here) */
byte a;
/* "b" is not a global variable but a constant in ROM */
byte b=2;

/* Oops! Where is the nmi routine? BUG! */

void main(void)
{
        /* c is a local variable */
        byte c;
        /* d is a valid local variable too because a local variable can be initialised */
        byte d=4;

        a=1; /* We finally initialise the "a" global variable! Yeah! */
        b=d-a; /* Oops, we try to write in ROM. BUG! */
        c=(b+d)/2; /* No problem here */
        d=a+b+c; /* No problem here */
}
```

Note: Except for the nmi routine bug, this program will run perfectly with Virtual Coleco. Please, use a better Coleco emulator to test your projects.

CAN IT BE BETTER?

Yes, the Rebound project can be better by adding cool graphics, some colours and sound effects, a title screen, a background music, a menu, etc. All you can add in the project to make it better are welcome. But, all you are trying to add can makes some unwanted bugs. If so, you may have to forget about it unless you find another way to program what you want.

You must change the ball by another graphic. Using the letter O is not really cool. You have to use an unused character to change his pattern and colours to be the new ball.

Character graphic: Ball								Color	Pattern
0	0	1	1	1	1	0	0	40	3C
0	1	1	1	1	1	1	0	40	7E
1	0	0	1	1	1	1	1	47	9F
1	0	0	1	1	1	1	1	47	9F
1	1	1	1	1	1	1	1	40	FF
1	1	1	1	1	1	1	1	40	FF
0	1	1	1	1	1	1	0	40	7E
0	0	1	1	1	1	0	0	40	3C

1. Add the color and pattern data in the project.

```
static byte ball_colors[] = {0x40, 0x40, 0x47, 0x47, 0x40, 0x40, 0x40, 0x40};
static byte ball_pattern[] = {0x3C, 0x7E, 0x9F, 0x9F, 0xFF, 0xFF, 0x7E, 0x3C};
```

2. Add routines to load the pattern and color data into the video memory.

```
void initialize (void)
{
        …
        /* To change the pattern and color of the character number 128 */
        /* We not use the capital letter O anymore to be the ball */
        change_pattern ( 128, ball_pattern,1 );
        change_multicolor ( 128, ball_colors );
}
```

3. Modify the game engine to use the new graphic.

```
        /* We replaced the 'O' by the character number 128 */
        put_char ( ball_x, ball_y, 128 );
```

SMASH - A VIDEO GAME VERSION OF REBOUND

The Rebound project can be modified to be a cool video game like Pong or Breakout. Try to figure out all you need to make this simple project a real video game. Think about all the variables and routines you need before starting to modify the source code.

Let's try to do a "smashing" game with the project Rebound by adding a paddle at the bottom of the screen. And if the ball reach the bottom of the screen, the game is over.

This paddle will be 4 characters long and will move only left and right. The ball will bounce on the walls and the paddle without erasing it.

You need to keep the information about the position of the paddle. Because the paddle move only horizontally (left and right) you can use a constant for the Y position of the paddle.

```
char paddle_x;
char paddle_y = 22;
char paddle_len = 4;
```

Don't forget to initialise the X position of the paddle.

```
static void initialize (void)
{
        …
        paddle_x = 20;
}
```

You can use the position in X and Y of the ball and the paddle to detect if the ball is on the paddle. To make the ball bounce on the paddle, you need to change the Y direction of the ball.

```
static void bounce_on_walls_in_Y (void)
{
        char temp_x;
        if ( ball_y == 0 ) { ball_dy = 1; pop(); }
        /* To make the ball bounce on the paddle */
        if ( ball_y == paddle_y-1 )
        {
        if ( ball_x >= paddle_x && ball_x < paddle_x+paddle_len )
                { ball_dy = -1; pop(); }
        else
        {       temp_x =  ball_x + ball_dx;
                if ( temp_x >= paddle_x && temp_x < paddle_x+paddle_len )
                {
                                ball_dx = -ball_dx;  ball_dy = -1; pop();
                                bounce_on_walls_in_X();
                        }
        }
        }
}
```

If the Y position of the ball is the same as the Y position of the paddle then the game is over. So you have to change the condition of the "while" loop. In the "while" loop, you must add the instructions to show and move the paddle on screen.

```
static void bounce(void)
{
        …
        while ( ball_y < paddle_y )
        {
                if ( joypad_1 & ( LEFT | RIGHT ) )
                {
                        /* Erase the paddle */
                        print_at ( paddle_x, paddle_y, "    " ); /* four(4) spaces */
                /* Change the X position of the paddle */
                        if ( joypad_1 & LEFT ) paddle_x--;
                        if ( joypad_1 & RIGHT ) paddle_x++;
                /* Keep the paddle in the limit of the screen */
                        if (paddle_x < 0 ) paddle_x = 0;
                        if (paddle_x > 32-paddle_len ) paddle_x = 32-paddle_len;
                }
                …
        print_at ( paddle_x, paddle_y, "XXXX" ); /* paddle: four(4) blocks *
                delay (5);
        }
        /* Exit this bouncing routine by pressing the fire button */
        while (!(joypad_1&FIRE1));
}
```

After compiling and testing these modifications, you may notice that the game is too slow. The actual game speed is not challenging. Try this:

```
static void bounce(void)
{
        …
                delay (3);
        …
}
```

The actual paddle speed is the same as the ball speed and this can be frustrating. You have to calibrate the game to let the paddle move faster than the ball sometimes. You may have to code a more natural movement for the paddle or use the fire button for fast movements. Find your own solution for the speed of the paddle.

Replace the characters of the paddle with cool graphics.

More suggestions: add bricks, modify the size of the paddle, add a "two players" mode, etc.

PADDLE GRAPHIC

Like for the ball character, we must use cool graphics for the paddle.

Paddle graphic: block								Color	Pattern
1	1	1	1	1	1	1	1	E0	FF
0	0	0	0	0	0	0	1	EF	01
1	1	1	1	1	0	1	1	E7	FB
1	0	0	0	0	0	1	0	E7	82
1	1	0	1	1	0	0	0	E7	D8
0	0	0	0	1	0	1	0	E7	0A
1	1	0	1	1	1	1	1	E7	DF
1	1	1	1	1	1	1	1	E0	FF

Paddle graphic: left								Color	Pattern
0	1	1	1	1	1	1	1	80	7F
1	1	0	0	0	0	1	1	89	C3
1	0	0	1	1	1	1	1	89	9F
1	0	1	1	1	1	1	1	89	BF
1	1	1	1	1	1	0	1	86	FD
1	1	1	1	0	0	0	1	86	F1
1	1	0	0	0	0	0	1	86	C1
0	1	1	1	1	1	1	1	80	7F

Paddle graphic: right								Color	Pattern
1	1	1	1	1	1	1	0	80	FE
1	0	0	0	0	0	1	1	89	83
1	0	0	1	1	1	1	1	89	9F
1	0	1	1	1	1	1	1	89	BF
1	1	1	1	1	1	0	1	86	FD
1	1	1	1	0	0	0	1	86	F1
1	1	0	0	0	0	1	1	86	C3
1	1	1	1	1	1	1	0	80	FE

Note: The colors pattern of the extreme left and right part of the paddle are the same.

THE PROGRAM

```
#include <coleco.h>
#include <getput1.h>

/* Information about the location of the ball */
char ball_x;
char ball_y;

/* Information about the direction of the ball */
char ball_dx;
char ball_dy;

/* Information about the paddle */
char paddle_x;
char paddle_y = 22;
char paddle_len = 4;

/* Graphics of the ball and the paddle */
static byte ball_colors[] = {0x40, 0x40, 0x47, 0x47, 0x40, 0x40, 0x40, 0x40};
static byte ball_pattern[] = {0x3C, 0x7E, 0x9F, 0x9F, 0xFF, 0xFF, 0x7E, 0x3C};

static byte paddle_bouncer_colors[] = {0x80, 0x89, 0x89, 0x89, 0x86, 0x86, 0x86, 0x80};
static byte paddle_block_colors[] = {0xE0, 0xEF, 0xE7, 0xE7, 0xE7, 0xE7, 0xE7, 0xE0};
static byte paddle_pattern[] =
{0x7F, 0xC3, 0x9F, 0xBF, 0xFD, 0xF1, 0xC1, 0x7F,
 0xFF, 0x01, 0xFB, 0x82, 0xD8, 0x0A, 0xDF, 0xFF,
 0xFE, 0x83, 0x9F, 0xBF, 0xFD, 0xF1, 0xC3, 0xFE};

/* A sound effect named "pop" is played when the ball reaches the border of the screen */
static byte pop_sound[] = {         0, 0x63,0xf,1,
        0x81,0xa0,0x90,1, 0x81,0x1c,0x97,1, 0x81,0x2c,0x9d,2, 0x81,0x32,0x9e,1,
        0,0,0};

/* The SHOOT sound from Cosmo Challenge */
static byte shoot_sound[]= {        1,
        0xf8,0xe4, 1,0xf2, 1,0xe4, 1,0x63,0x02,0x01,0xe5,
        1,0xe4, 1,0xe5, 1,0xe4, 1,0xe5, 1,0xe4, 1,0xe5, 1,0xe4, 5,
        0,0,0};

/* To start the "pop" sound */
static void pop (void) { start_sound (pop_sound,2); }

/* To start the "shoot" sound */
static void shoot (void) { start_sound (shoot_sound,1); }
```

```
/* To initialize the location and direction of the ball */
static void initialize (void)
{
        ball_y = 0; /* top */
        ball_x = 0; /* left */
        ball_dx = 1; /* moving to the right */
        ball_dy = 1; /* moving to the bottom */
        paddle_x = 20;

        /* BALL */
        change_pattern ( 128, ball_pattern,1 );
        change_multicolor ( 128, ball_colors );
        /* PADDLE */
        change_pattern ( 'x', paddle_pattern,3 );
        change_multicolor ( 'x', paddle_bouncer_colors );
        change_multicolor ( 'y', paddle_block_colors );
        change_multicolor ( 'z', paddle_bouncer_colors );
}

/* Change the direction of the ball when bouncing horizontally on the border of the screen */
static void bounce_on_walls_in_X (void)
{
        if ( ball_x == 0 ) { ball_dx = -ball_dx; pop(); }
        if ( ball_x == 31 ) { ball_dx = -ball_dx; pop(); }
}

/* Change the direction of the ball when bouncing vertically on the border of the screen */
static void bounce_on_walls_in_Y (void)
{
        char temp_x;
        if ( ball_y == 0 ) { ball_dy = 1; pop(); }
        /* To make the ball bounce on the paddle */
        if ( ball_y == paddle_y-1 )
        {
                if ( ball_x >= paddle_x && ball_x < paddle_x+paddle_len )
                { ball_dy = -1; pop(); }
                else
                {       temp_x =  ball_x + ball_dx;
                        if ( temp_x >= paddle_x && temp_x < paddle_x+paddle_len )
                        {
                                ball_dx = -ball_dx;  ball_dy = -1; pop();
                                bounce_on_walls_in_X();
                        }
                }
        }
}
```

```
/* To update the location and direction of the ball. */
static void update_ball_location (void)
{
	ball_x += ball_dx;
	ball_y += ball_dy;
	bounce_on_walls_in_X();
	bounce_on_walls_in_Y();
}

/* This part of the program is the game engine and it's used to rebound a ball on screen */
static void smash(void)
{
	/* Initialize the ball and the paddle */
	initialize ();
	/* Clear screen */
	cls();

	/* The animation will stop when pressing on fire1 on joystick#1 */
	while ( ball_y < paddle_y )
	{
		if ( joypad_1 & ( LEFT | RIGHT ) )
		{
			/* Erase the paddle */
			print_at ( paddle_x, paddle_y, "    " ); /* four(4) spaces */
		/* Change the X position of the paddle */
			if ( joypad_1 & LEFT ) paddle_x--;
			if ( joypad_1 & RIGHT ) paddle_x++;
		/* Keep the paddle in the limit of the screen */
			if (paddle_x < 0 ) paddle_x = 0;
			if (paddle_x > 32-paddle_len ) paddle_x = 32-paddle_len;
		}

		put_char ( ball_x, ball_y, 32 ); /* 32 is the value of the SPACE character */
		/* Update the value of the location of the ball */
		update_ball_location ();
		/* Print the ball at the new location */
		put_char ( ball_x, ball_y, 128 );
		/* Print the ball at the new location */
		print_at ( paddle_x, paddle_y, "xyyz" ); /* paddle */
		/* Slowdown the animation */
		delay(3);
	}
}
```

```
/* NMI routine: update sound at every vertical retrace */
void nmi(void) { update_sound(); }

void gameover(void)
{
        /* Play shoot sound */
        shoot();
        /* Print "GAME OVER" on screen */
        center_string ( 11, "GAME  OVER" );
        /* Waiting for FIRE button */
        pause();
}

void main(void)
{
        /* Initialize the VDP to the screen mode 2 */
        screen_mode_2_text();
        /* Set the default ascii character set */
        upload_default_ascii (BOLD);
        /* Start the bouncing ball video game */
        smash();
        /* Print "GAME OVER" on screen */
        gameover();
}
```

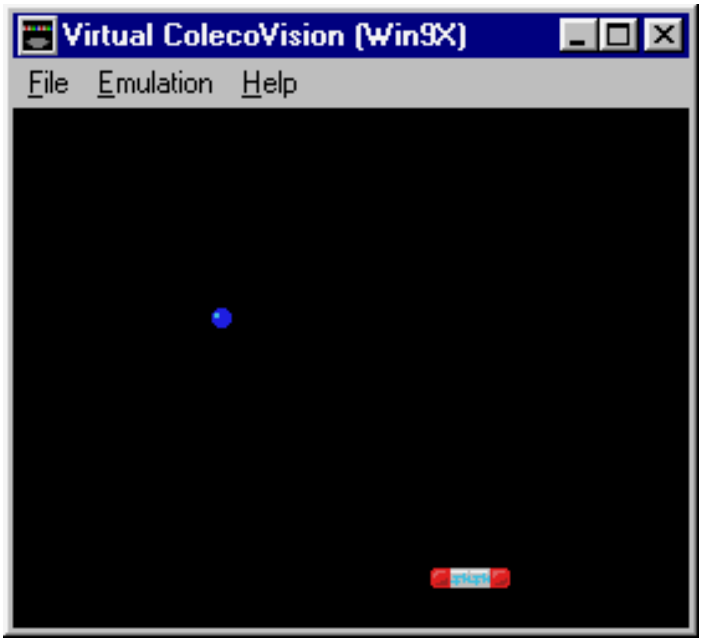


Figure 17 Smash game running with VirtualColeco emulator

DEBUG EXERCISE

A TEST PROGRAM TO DEBUG

Try to find the bugs in the following code. What is missing? What is typed in a wrong way?
You can try to compile this code to help you find the bugs.

```
#include <coleco.h>
#include <getpput.h>

byte a=1;

static void test(void)
{
        a--;
}

void main(void)
{
        screen_mode_2_text();
        cls()
        print(5,6,"PRESS FIRE TO CONTINUE..?");
        pause();
        test();
        if (a==0)
        {
                print(5,6,"THIS PROGRAM IS BUG FREE");
                pause();
        }
}
```

SOLUTION

This is the corrected version of the previous test program. Read the remarks for information.

```
#include <coleco.h>
/* getpput.h is not the right name to use for the getput 1 library */
#include <getput1.h>

/* Oops, we can't initialise a global variable like this: byte a=0; */
byte a;

static void test(void)
{
	a--;
}

/* Oops, we forgot the NMI process. This process is necessary and can't be omitted */
void nmi()
{
	/* do nothing */
}

void main(void)
{
	/* This is a good place to initialise the global variable "a" */
	a=1;
	screen_mode_2_text();
	/* Set the default ascii character set. Otherwise, we see nothing on screen. */
	upload_default_ascii (BOLD);
	/* Oops, we forgot the ";" to separate the instructions */
	cls();
	/* The print instruction is not valid. We have to use the print_at instruction */
	print_at(5,6,"PRESS FIRE TO CONTINUE..?");
	pause();
	/* Oops, an error of logic, we forgot to clear the screen */
	cls();
	test();
	if (a==0)
	{
	/* The print instruction is not valid. We have to use the print_at instruction */
		print_at(5,6,"THIS PROGRAM IS BUG FREE");
	pause();
	}
}
```

OPTIMIZATION TRICKS

Before trying these tricks, save your project and make sure it compiles and link perfectly.

Trick #1 : divide and multiply by using bit shifting

Use bit shifting to avoid multiplication and division by power of 2 (2,4,8,...).

Before	***After***
byte x_char = x_sprite / 8; byte y_sprite = (y_char * 8) - 1;	byte x_char = x_sprite >> 3; byte y_sprite = (y_char << 3) - 1;

Trick #2 : a useful pointer

Keep a pointer to the memory location you are using again and again.

Before	***After***

byte a[9];
 	
void not_optimized(byte index)
{
  a[index] += 5;
  if (a[index] > 25) a[index] = 0;
}
	byte a[9];
	
	void optimized(byte index)
	{
	 byte *ptr_a = &a[index];
	  (*ptr_a) += 5;
	  if ((*ptr_a) > 25) (*ptr_a) = 0;
	}

typedef struct {
  char x,y; /* coordinates */
  char dx,dy; /* direction speed */
} coorxy;
 	
coorxy ghost[10];
 	
void not_optimized(byte index)
{
  ghost[index].x += ghost[index].dx;
  if (ghost[index].x<0) ghost[index].x=31;
  if (ghost[index].x>31) ghost[index].x=0;
 	
  ghost[index].y += ghost[index].dy;
  if (ghost[index].y<0) ghost[index].y=23;
  if (ghost[index].y>23) ghost[index].y=0;
}
	typedef struct {
	  char x,y; /* coordinates */
	  char dx,dy; /* direction speed */
	} coorxy;
	
	coorxy ghost[10];
	
	void optimized(byte index)
	{
	  coorxy *this_ghost = &ghost[index];
	  char *this_x = &this_ghost->x;
	  char *this_y = &this_ghost->y;
	
	  (*this_x) += this_ghost->dx;
	  if ((*this_x)<0) (*this_x)=31;
	  if ((*this_x)>31) (*this_x)=0;
	
	  (*this_y) += this_ghost->dy;
	  if ((*this_y)<0) (*this_y)=23;
	  if ((*this_y)>23) (*this_y)=0;
	}

Trick #3 : fill up RAM with memset

Use memset function rather than a loop to fill up tables in RAM with the same value.

Before	*After*

char a[32];
byte b[10][10];
 	
byte i,j;
for (i=0;i<32;i++) a[i]='A';
for (i=0;i<10;i++) for (j=0;j<10;j++) b[i][j]=0;
	char a[32];
	char b[10][10];
	
	memset (a,32,'A');
	memset (b,100,0);

Trick #4 : load_ascii VS upload_ascii VS upload_default_ascii

To free up space in your project, don't add an ASCII charset in the project, simply use the Coleco ASCII font.

load_ascii : calls the Coleco BIOS load_ascii routine to load the entire ASCII table in VRAM, no extra data needed.

upload_ascii : more flexible than the Coleco BIOS load_ascii routine, it loads a part of the ASCII table in VRAM with an italic and/or bold effect.

upload_default_ascii : This routine from the Getput library simply calls Marcel's upload_ascii routine with specific parameters. Use directly upload_ascii to free up a couple of bytes in ROM.

Trick #5 : do your own sprite routines

Marcel's sprite routines are too big, too general for your needs. You can declare a shorter sprite attributes table in your project and use put_vram to update sprite attributes in VRAM. By doing this, you free up RAM and ROM space in your project, but you have to add a byte 208 (D0) after the last sprite attributes in VRAM.

Before	*After*

#include <coleco.h>
 	
/* sprites table already defined in Marcel's library */
 	
void show_sprites(void)
{
  update_sprites(2,0x1B00); /* show 2 sprites */
}
	#define NO_SPRITES
	#include <coleco.h>
	
	typedef struct
	{
	 byte y;
	 byte x;
	 byte pattern;
	 byte colour;
	} sprite_t;
	sprite_t sprites[3]; /* 2 sprites + 1 dummy (0xD0) */
	
	void show_sprites(void)
	{
	  sprites[2].y = 0xD0;
	  put_vram(0x1B00,sprites,9); /* 9 = 2 sprites + 0xD0 */
	}

THAT's ALL?

Yes, that’s all!

You are ready to create your own ColecoVision projects.

Make your first ColecoVision project as simple as possible. Do "learning" tests first to gain programming skills.

Words of wisdom that all new ColecoVision games designers should adhere to:

"You're very first effort should be something for YOU, not something you plan to release. Have fun with it, don't try to bite off more than you can chew... trust me on that one."

When it's time to do a big ColecoVision game project, takes some important notes and sketch a storyboard with key-situations. Code your project one part at a time, start with graphics and game parameters. Compile your project after each modification to see if there is no mistake in your code. Find bugs by testing, testing and testing again or ask some beta-testers to seek bugs in your project. Share your new game with ColecoVision fans and ask for feedback and comments. Tune up your game and fix all the bugs before thinking of releasing it in cartridge. When the final version is ready, release your game in a cartridge format or ask someone to do it for you.

Good Luck in your future ColecoVision projects! ☺

Best regards,

Daniel Bienvenu

APPENDIX A - MORE TECHNICAL INFORMATION

HARDWARE SPECIFICATIONS

Note: This information came from the ColecoVision FAQ.

Resolution:	256 x 192
CPU:	Z-80A
Bits:	8
Speed:	3.58 MHz
RAM:	1K (7000-73FF but copied at different addresses)
Video RAM:	16K = 8x 2K (4116 RAM chip)
Video Display Processor:	Texas Instruments TMS9928A
Sprites:	32
Colors:	16 (15 colors plus one invisible)
Sound:	Texas Instruments SN76489AN; 3 tone channels, 1 noise
Cartridge ROM:	8K/16K/24K/32K

CARTRIDGE (ROM) HEADER

Note: This information came from a ColecoVision technical documentation.

8000 - 8001: If AA and 55, the CV will show the CV title screen.
If 55 and AA, the CV will jump directly to the start address.

The following bytes always point to 0000 or RAM (7xxx)
8002 - 8003: Pointer to Sprites table (table sprites properties: Y, X, pattern, colour?)
8004 - 8005: Pointer to Sprites table (in which order the coleco bios show the sprites?)
8006 - 8007: Pointer to RAM space to let ColecoVision BIOS to temporary stock data
8008 - 8009: Pointer to Joysticks: 12 bytes: 2 control bytes, 5 bytes for joystick port#1, 5 bytes for joystick port#2.

control byte : [READ JOYSTICK?][][][KEYPAD][RIGHT][SPINNER][JOYPAD][LEFT])

5 bytes for the joystick attribute (bits set to one when pressed): left fire, direction [left,down, right, up], spinner, right fire, keypad.

800A - 800B: Start address of the game

800C - 800E: Jump to: RST 08h
800F - 8011: Jump to: RST 10h
8012 - 8014: Jump to: RST 18h
8015 - 8017: Jump to: RST 20h
8018 - 801A: Jump to: RST 28h
801B - 801D: Jump to: RST 30h
801E - 8020: Jump to: RST 38h
8021 - 8023: Jump to: NMI (Vertical Retrace Interrupt)

8024 - ????: Title screen data:

COLECO VISION
PRESENTS
LINE 1
LINE 2
YEAR

The title screen data is stored as one string with the '/' character (2Fh) used as a delimiter. It signals the end of a line, and isn't printed.

"LINE 2/LINE 1/YEAR"

Note: There isn't an end-of-line delimiter, because the year is always 4 characters long.

SOUND GENERATION HARDWARE

Note: This information came from a ColecoVision sound documentation.

The ColecoVision uses the Texas Instruments SN76489A sound generator chip as the output port ffh. It contains three programmable tone generators, each with its own programmable attenuator, and a noise source with its own attenuator.

TONE GENERATORS

Each tone generator consists of a frequency synthesis section requiring 10 bits of information to define half the period of the desired frequency (n). F0 is the most significant bit and F9 is the least significant bit. The information is loaded into a 10 stage tone counter, which is decremented at a N/16 rate where N (3.579MHz) is the input clock frequency. When the tone counter decrements to zero, a borrowed signal is produced. This borrowed signal toggles the frequency flip-flop and also reloads the tone counter. Thus, the period of the desired frequency is twice the value of the period register.

The frequency can be calculated by the following:

$$f = 3.579\text{MHz}/(32n)$$

NOISE GENERATOR

The noise generator consists of a noise source that is a shift register with an exclusive OR feedback network. The feedback network has provisions to protect the shift register from being locked in the zero state.

Noise Feedback Control

Feedback (FB)	Configuration
0	“Periodic” Noise
1	“White” Noise

Noise Generator Frequency Control

NF0	NF1	Shift Rate
0	0	N/512
0	1	N/1024
1	0	N/2048
1	1	Tone gen. #3 output

CONTROL REGISTERS

The SN76489A has 8 internal registers which are used to control the 3 tone generators and the noise source. During all data transfers to the SN76489A, the first byte contains a 3 bits field which determines the channel and the control/attenuation. The channel codes are shown below.

Register Address Field

R0	R1	Destination Control Register
0	0	Tone 1
0	1	Tone 2
1	0	Tone 3
1	1	Noise

The output of the frequency flip-flop feeds into a 4 stage attenuator. The attenuator values, along with their bit position in the data word, are shown below. Multiple attenuation control
bits may be true simultaneously. Thus, the maximum attenuation is 28 db.

A0	A1	A2	A3	Weight
0	0	0	1	2 db
0	0	1	0	4 db
0	1	0	0	8 db
1	0	0	0	16 db
1	1	1	1	OFF

SOUND DATA FORMATS

The formats required to transfer data to sound port ffh are shown below.

Frequency

1	Reg. Addr			Data			
	R0	R1	0	F6	F7	F8	F9

0	X	Data					
		F0	F1	F2	F3	F4	F5

Noise Control

1	Reg. Addr			X	FB	Shift	
	1	1	0			NF0	NF1

Attenuator

1	Reg. Addr			Data			
	R0	R1	1	A0	A1	A2	A3

NOTES TABLE CONVERSION: FREQUENCIES (Hz) <-> HEX values

	Hz	HEX	Hz	HEX	Hz	HEX	Hz	HEX	Hz	HEX
A	110.00	3F8	220.00	1FC	440.00	0FE	880.00	07F	1760.0	03F
A#/B^b	116.54	3BF	233.08	1DF	466.16	0EF	932.33	077	1864.6	03B
B	123.47	389	246.94	1C4	493.88	0E2	987.77	071	1975.5	038
C	130.81	356	261.63	1AB	523.25	0D5	1046.5	06A	2093.0	035
C#/D^b	138.59	327	277.18	193	554.36	0C9	1108.7	064	2217.5	032
D	146.83	2F9	293.66	17C	587.33	0BE	1174.7	05F	2349.3	02F
D#/E^b	155.56	2CE	311.13	167	622.25	0B3	1244.5	059	2489.0	02C
E	164.81	2A6	329.63	153	659.25	0A9	1318.5	054	2637.0	02A
F	174.61	280	349.23	140	698.46	0A0	1396.9	050	2793.8	028
F#/G^b	185.00	25C	370.00	12E	739.99	097	1480.0	04B	2960.0	025
G	196.00	23A	391.99	11D	783.99	08E	1568.0	047	3136.0	023
G#/A^b	207.65	21A	415.30	10D	830.61	086	1661.2	043	3322.4	021

Remark: The frequency of the medium C is 523.25 Hz. The frequency of the same note C but an octave higher is 1046.5 Hz.

SCALES

A Scale is a series of notes which we define as "correct" or appropriate for a song.

Examples of various Scales (Root = "C"):

Name	C	D^b	D	E^b	E	F	G^b	G	A^b	A	B^b	B
Major	1		2		3	4		5		6		7
Minor	1		2	3		4		5	6		7	
Harmonic Minor	1		2	3		4		5	6			7
Melodic Minor (asc)	1		2	3		4		5		6		7
Melodic Minor (desc)	1		2	3		4		5	6		7	
Enigmatic	1	2			3		4		5		6	7
Flamenco	1	2		3	4	5		6	7		8	
Major Triad	1				2			3				
Minor Triad	1			2				3				

MARCEL's SOUND DATA FORMAT

(Source from Dale wick and tests)

Sound Header

0 = melody
1 = noise

Sound Body

HEX CODE	EXTRA BYTES	INSTRUCTIONS
00	2 bytes: address in 2 bytes	**Jump** Continue sound at a specific address Note: END is address 0000
01 - 3F		**Delay**
4X	2 bytes: speed, increment	**Frequency sweep*** Speed = [1 (=fast), FF (=slow)] Increment = [80 (=-128),FF (=-1)] U [1,7F] Note: If increment > 0, frequency is going down Note: If speed or increment = 0, no effect
6X	2 bytes: speed, increment	**Attenuation sweep** Speed = [1 (=fast), FF (=slow)] Increment = [F1 (=-15),FF (=-1)] U [1,F] Note: If increment > 0, volume is going down Note: If speed or increment = 0, no effect
8X	1 byte: low part of the frequency	**Frequency*** Format: 8X YZ, Frequency = XYZ
9X		**Attenuation*** Note: 90 = loud, 9F = silence
AX	1 byte	Same as 8X
BX		Same as 9X
CX	1 byte	Same as 8X
DX		Same as 9X
EX		**Noise sound**** If X=0,1 or 2, play a BUZZ sound If X=3, play a BUZZ sound with output ch3 If X=4,5 or 6, play a NOISE sound If X=7, play a NOISE sound with output ch3
F0 - FF		**Attenuation**** Note: F0 = loud, FF = silence

*: Only for melodic mode
**: Only for noise mode

VDP - VIDEO DISPLAY PROCESSOR

(from Texas Instrument documentation)

VDP has 8 control registers (0-7) and one status register.

REGISTERS

Control registers

Register	***Bits***							
	7	6	5	4	3	2	1	0
0	-	-	-	-	-	-	M2	EXT
1	4/16K	BL	GINT	M1	M3	-	SI	MAG
2	-	-	-	-	PN13	PN12	PN11	PN10
3	CT13	CT12	CT11	CT10	CT9	CT8	CT7	CT6
4	-	-	-	-	-	PG13	PG12	PG11
5	-	SA13	SA12	SA11	SA10	SA9	SA8	SA7
6	-	-	-	-	-	SG13	SG12	SG11
7	TC3	TC2	TC1	TC0	BD3	BD2	BD1	BD0

M1, M2, M3	Select screen mode
EXT	Enables external video input
4/16K	Selects 16K Video RAM if set
BL	Blank screen if reset
SI	16x16 sprites if set; 8x8 if not
MAG	Sprites enlarged if set (double sized: sprite pixels are 2x2)
GINT	Generate interrupts if set
PN*	Address for pattern name table (screen)
CT*	Address for colour table (special meaning in M2)
PG*	Address for pattern generator table (special meaning in M2)
SA*	Address for sprite attribute (y, x, pattern, colour) table
SG*	Address for sprite generator table
TC*	Text colour (foreground)
BD*	Back drop (background)

Status register

INT	5S	C	FS4	FS3	FS2	FS1	FS0

FS*	Fifth sprite (first sprite not displayed). Valid if 5S is set
C	Sprite collision detected
5S	Fifth sprite (not displayed) detected
INT	Set at each screen update (refresh)

VDP register access

The status register can't be write. After reading the status register, INT (bit#7) and C (bit#5) are reset.

```
In ASM:     in      a,(bfh) ; get register value (COLECO BIOS: call 1fdch)
In C:       byte a = vdp_status; /* vdp_status is updated after the NMI routine */
```

The control registers can't be read. Two bytes must be writen:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	V7	V6	V5	V4	V3	V2	V1	V0
Byte 1	1	-	-	-	-	R2	R1	R0

Legend

```
V*          Value to be writen in the register.
R*          Register number.
```

```
In ASM:     ld      a, value
            out     (bfh),a ; set value
            ld      a, register_number
            add     a,80h
            out     (bfh),a ; write value in register
In C:       vdp_out (register_number, value); /* (COLECO BIOS: call 1fd9h) */
```

NMI Non maskable interrupt

After the vertical retrace (refresh is done), the bit 7 of the status register is set.
If GINT (bit 5 of control register#1) is set, the NMI interrupts the normal execution.
When it's time again to refresh, the bit 7 of the status register is reset.

NMI can be used to execute something again and again at a regular speed like updating sounds. Some games use NMI to call the game engine.

Screen modes

Mode 0 - Graphic I

Description: 32x24 characters, two colors per 8 characters, sprites active.

Mode 1 - Text

Description: 40x24 characters (6x8), colors set in control register#7, sprites inactive.

Mode 2 - Graphic II

Description: 32x24 characters, 256x192 pixels, two colors per line of 8 pixels, sprites active.

Special meaning for CT* and PG*:

At control register#3, only bit 7 (CT13) sets the address of the color table (address: 0000 or 2000). Bits 6 - 0 are an AND mask over the top 7 bits of the character number.

At control register#4, only bit 2 (PG13) sets the address of the pattern table (address: 0000 or 2000). Bits 1 and 0 are an AND mask over the top 2 bits of the character number. If the AND mask is:

- 00, only one set (the first one) of 256 characters is used on screen.
- 01, the middle of the screen (8 rows) use another set (the second one) of 256 characters.
- 10, the bottom of the screen (8 rows) use another set (the third one) of 256 characters.
- 11, three set of 256 characters are used on screen: set one at the top (8 rows), set two in the middle (8 rows), and set three at the bottom (8 rows). This particular mode is normally used as a bitmap mode screen. The bitmap mode screen is in fact all the three characters set (top, middle and bottom) showed on screen at the same time by filling the screen with all the characters.

Mode 3 - Multicolor

Description: 64x48 big pixels (4x4), sprites active.

COLECO SCREEN MODE 1 (TEXT MODE)

M	O	D	E	1								1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
		0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
	0																																								
	1																																								
	2																																								
	3																																								
	4																																								
	5																																								
	6																																								
	7																																								
	8																																								
	9																																								
1	0																																								
1	1																																								
1	2																																								
1	3																																								
1	4																																								
1	5																																								
1	6																																								
1	7																																								
1	8																																								
1	9																																								
2	0																																								
2	1																																								
2	2																																								
2	3																																								

Not usefull except in a text adventure game, this screen mode is not supported in getput library but can be set by doing the right vdp_out calls. Because there are 40 characters per line, all the print functions cannot be used properly in this screen mode, including get_char and put_char.

COLECO SCREEN MODE 0 & 2 (GRAPHIC I & II MODE)

M	O	D	E	0	&	2						1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3
		0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
	0																																
	1																																
	2																																
	3																																
	4																																
	5																																
	6																																
	7																																
	8																																
	9																																
1	0																																
1	1																																
1	2																																
1	3																																
1	4																																
1	5																																
1	6																																
1	7																																
1	8																																
1	9																																
2	0																																
2	1																																
2	2																																
2	3																																

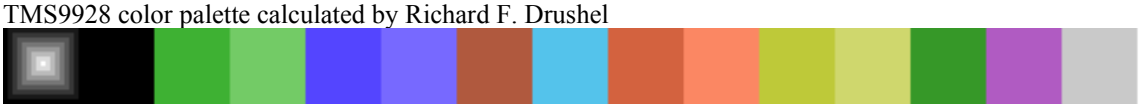
Screen mode 2 is fully supported in getput library. It's the most colorful screen mode and used mostly to show nice full screen bitmap pictures.

Screen mode 0 can be set with the right vdp_out calls and can be used with all the print functions in getput library.

COLOR PALETTE

COLOR #	*COLOR*	*Y*	*R-Y*	*B-Y*
0	Invisible	-	-	-
1	Black	0.00	0.47	0.47
2	Medium Green	0.53	0.07	0.20
3	Light Green	0.67	0.17	0.27
4	Dark blue	0.40	0.40	1.00
5	Light blue	0.53	0.43	0.93
6	Dark Red (brown)	0.47	0.83	0.30
7	Cyan	0.73	0.00	0.70
8	Medium Red	0.53	0.93	0.27
9	Light Red (Pink/orange)	0.67	0.93	0.27
10 (A)	Dark Yellow (Yellow)	0.73	0.57	0.07
11 (B)	Light Yellow (Yellow + Light Grey)	0.80	0.57	0.17
12 (C)	Dark Green	0.47	0.13	0.23
13 (D)	Magenta	0.53	0.73	0.67
14 (E)	Grey (Light Grey)	0.80	0.47	0.47
15 (F)	White	1.00	0.47	0.47

TMS9928 color palette calculated by Richard F. Drushel



TMS9938 color palette calculated by Marat Fayzullin

TMS9928 color palette used in MESS emulator

The default color palette used in ADAMEM is the one calculated by Richard F. Drushel.
The color palette used in COLEM is the one calculated by Marat Fayzullin.
The color palette I see in my Commodore monitor model 1802 looks like the one used in MESS emulator.

More information about Texas Instruments TMS99n8 color palette.

URL:
http://junior.apk.net/~drushel/pub/coleco/twwmca/wk961118.html
http://junior.apk.net/~drushel/pub/coleco/twwmca/wk961201.html
http://junior.apk.net/~drushel/pub/coleco/twwmca/wk970202.html

COLECO ASCII TABLE

DEC: 0-63, HEX: 00-3F

DEC	*HEX*	*CHARACTER*	*DEC*	*HEX*	*CHARACTER*
0	00	(null)	32	20	Space
1	01		33	21	!
2	02		34	22	"
3	03		35	23	#
4	04		36	24	$
5	05		37	25	%
6	06		38	26	&
7	07		39	27	'
8	08		40	28	(
9	09		41	29	)
10	0A		42	2A	*
11	0B		43	2B	+
12	0C		44	2C	,
13	0D		45	2D	-
14	0E		46	2E	.
15	0F		47	2F	/
16	10		48	30	0
17	11		49	31	1
18	12		50	32	2
19	13		51	33	3
20	14		52	34	4
21	15		53	35	5
22	16		54	36	6
23	17		55	37	7
24	18		56	38	8
25	19		57	39	9
26	1A		58	3A	:
27	1B		59	3B	;
28	1C		60	3C	<
29	1D	©	61	3D	=
30	1E	™	62	3E	>
31	1F	™	63	3F	?

DEC: 64-127, HEX: 40-7F

DEC	*HEX*	*CHARACTER*	*DEC*	*HEX*	*CHARACTER*
64	40	@	96	60	`
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z
91	5B	[	123	7B	{ (brace left)
92	5C	\	124	7C	| (broken vertical)
93	5D	]	125	7D	} (brace right)
94	5E	^	126	7E	~ (tilde)
95	5F	_ (underline)	127	7F	▒ (deleted)

APPENDIX B - ORIGINAL OS7' BIOS INFORMATION

JUMP TABLE

Legend:
P (at the end): function specifically done for Pascal programs.

```
1F61 > 0300 : PLAY_SONGS              1FB2 > 0203 : SOUND_INITP
1F64 > 0488 : ACTIVATEP               1FB5 > 0251 : PLAY_ITP
1F67 > 06C7 : PUTOBJP                 1FB8 > 1B08 : INIT_TABLE
1F6A > 1D5A : REFLECT_VERTICAL        1FBB > 1BA3 : GET_VRAM
1F6D > 1D60 : REFLECT_HORIZONTAL      1FBE > 1C27 : PUT_VRAM
1F70 > 1D66 : ROTATE_90               1FC1 > 1C66 : INIT_SPR_ORDER
1F73 > 1D6C : ENLARGE                 1FC4 > 1C82 : WR_SPR_NM_TBL
1F76 > 114A : CONTROLLER_SCAN         1FC7 > 0FAA : INIT_TIMER
1F79 > 118B : DECODER                 1FCA > 0FC4 : FREE_SIGNAL
1F7C > 1979 : GAME_OPT                1FCD > 1053 : REQUEST_SIGNAL
1F7F > 1927 : LOAD_ASCII              1FD0 > 10CB : TEST_SIGNAL
1F82 > 18D4 : FILL_VRAM               1FD3 > 0F37 : TIME_MGR
1F85 > 18E9 : MODE_1                  1FD6 > 023B : TURN_OFF_SOUND
1F88 > 116A : UPDATE_SPINNER          1FD9 > 1CCA : WRITE_REGISTER
1F8B > 1B0E : INIT_TABLEP             1FDC > 1D57 : READ_REGISTER
1F8E > 1B8C : GET_VRAMP               1FDF > 1D01 : WRITE_VRAM
1F91 > 1C10 : PUT_VRAMP               1FE2 > 1D3E : READ_VRAM
1F94 > 1C5A : INIT_SPR_ORDERP         1FE5 > 0664 : INIT_WRITER
1F97 > 1C76 : WR_SPR_NM_TBLP          1FE8 > 0679 : WRITER
1F9A > 0F9A : INIT_TIMERP             1FEB > 11C1 : POLLER
1F9D > 0FB8 : FREE_SIGNALP            1FEE > 0213 : SOUND_INIT
1FA0 > 1044 : REQUEST_SIGNALP         1FF1 > 025E : PLAY_IT
1FA3 > 10BF : TEST_SIGNALP            1FF4 > 027F : SOUND_MAN
1FA6 > 1CBC : WRITE_REGISTERP         1FF7 > 04A3 : ACTIVATE
1FA9 > 1CED : WRITE_VRAMP             1FFA > 06D8 : PUTOBJ
1FAC > 1D2A : READ_VRAMP              1FFD > 003B : RAND_GEN
1FAF > 0655 : INIT_WRITERP
```

OTHER OS SYMBOLS

(IN ALPHABETIC ORDER)

These OS symbols are declared as global symbols except those in red. They can be directly used by Coleco programmers but normally used when calling functions in the jump table.
Note: CTRL_PORT_PTR and DATA_PORT_PTR are used into the Marcel's Coleco library.

```
01B1 : ADD816                Add signed 8bit value A to 16bit [HL]
0069 : AMERICA               60 = NTSC, 50 = PAL
006A : ASCII_TABLE           Pointer to uppercase ASCII pattern
012F : ATN_SWEEP             Attenuation sweep
0000 : BOOT_UP               Initializes stack and jump to POWER_UP
08C0 : CALC_OFFSET           Calculates offset in name table =32*y+x
8000 : CARTRIDGE             Cartridge starting address
8008 : CONTROLLER_MAP        Pointer to controller memory map
1D43 : CTRL_PORT_PTR         (in READ_VRAM and equal I/O port# BF)
1D47 : DATA_PORT_PTR         (in READ_VRAM and equal I/O port# BE)
0190 : DECLSN                (in sound sweep functions)
019B : DECMSN                (in sound sweep functions)
73C6 : DEFER_WRITES          Boolean flag to defer writes to vram
02EE : EFXOVER               (in PROCESS_DATA_AREA to get next note)
1D6C : ENLRG                 It's the local function name for ENLARGE
00FC : FREQ_SWEEP            Frequency sweep
8024 : GAME_NAME             String of ASCII characters
0898 : GET_BKGRND            Get names from name table in vram
801E : IRQ_INT_VECT          Maskable interrupt soft vector (RST 38H)
01D5 : LEAVE_EFFECT          Called by a special sound effect function when
                             it's finished
8002 : LOCAL_SPR_TABLE       Pointer to sprite name table
01A6 : MSNTOLSN              (in sound sweep functions)
73C7 : MUX_SPRITES           Boolean flag to sprite multiplexing
8021 : NMI_INT_VECT          NMI soft vector
006C : NUMBER_TABLE          Pointer to numbers 0-9 pattern
006E : POWER_UP              Start game if a cartridge is plugged in
02D6 : PROCESS_DATA_AREA     Update sound counters or call effect
080B : PUT_FRAME             Put a frame of names to name table in vram
73C9 : RAND_NUM              Pointer to pseudo random number value
800F : RST_10H_RAM           Reset 10 soft vector
8012 : RST_18H_RAM           Reset 18 soft vector
8015 : RST_20H_RAM           Reset 20 soft vector
8018 : RST_28H_RAM           Reset 28 soft vector
801B : RST_30H_RAM           Reset 30 soft vector
800C : RST_8H_RAM            Reset 8 soft vector
8004 : SPRITE_ORDER          Pointer to sprite order table
73B9 : STACK                 Stack pointer address
800A : START_GAME            Pointer to game start code
73C3 : VDP_MODE_WORD         Copy of the first two VDP registers
73C5 : VDP_STATUS_BYTE       Contents of default NMI handler
8006 : WORK_BUFFER           Pointer to temporary storage in RAM
```

MEMORY MAP

From ADAM™ Technical Reference Manual

Note for ADAM users: The ADAM computer can be reset in either computer mode or in game mode. When the cartridge (or ColecoVision) reset switch is pressed, ADAM resets to game mode. In this mode, 32K of cartridge ROM are switched into the upper bank of memory, and OS 7' plus 24K of intrinsic RAM are switched into the lower bank of memory. So, it's possible to create a ColecoVision game with additional options if plugged into an ADAM computer and then use the extra RAM space and the ADAM peripherics.

COLECOVISION GENERAL MEMORY MAP

From ColecoVision FAQ

ADDRESS	DESCRIPTION
0000-1FFF	ColecoVision BIOS OS 7'
2000-5FFF	Expansion port
6000-7FFF	1K RAM mapped into 8K. (7000-73FF)
8000-FFFF	Game cartridge

GAME CARTRIDGE HEADER

From The Absolute OS 7' Listing

ADDRESS	NAME	DESCRIPTION
8000-8001	CARTRIDGE	Test bytes. Must be AA55 or 55AA.
8002-8003	LOCAL_SPR_TABLE	Pointer to RAM copy of the sprite name table.
8004-8005	SPRITE_ORDER	Pointer to RAM sprite order table.
8006-8007	WORK_BUFFER	Pointer to free buffer space in RAM.
8008-8009	CONTROLLER_MAP	Pointer to controller memory map.
800A-800B	START_GAME	Pointer to the start of the game.
800C-800E	RST_8H_RAM	Restart 8h soft vector.
800F-8011	RST_10H_RAM	Restart 10h soft vector.
8012-8014	RST_18H_RAM	Restart 18h soft vector.
8015-8017	RST_20H_RAM	Restart 20h soft vector.
8018-801A	RST_28H_RAM	Restart 28h soft vector.
801B-801D	RST_30H_RAM	Restart 30h soft vector.
801E-8020	IRQ_INT_VECTOR	Maskable interrupt soft vector (38h).
8021-8023	NMI_INT_VECTOR	Non maskable interrupt (NMI) soft vector.
8024-80XX	GAME_NAME	String with two delemiters "/" as "LINE2/LINE1/YEAR"

COMPLET OS 7' RAM MAP

ADDRESS	NAME	DESCRIPTION
7020-7021	PTR_LST_OF_SND_ADDRS	Pointer to list (in RAM) of sound addrs
7022-7023	PTR_TO_S_ON_0	Pointer to song for noise
7024-7025	PTR_TO_S_ON_1	Pointer to song for channel#1
7026-7027	PTR_TO_S_ON_2	Pointer to song for channel#2
7028-7029	PTR_TO_S_ON_3	Pointer to song for channel#3
702A	SAVE_CTRL	CTRL data (byte)
73B9	STACK	Beginning of the stack
73BA-73BF	PARAM_AREA	Common passing parameters area (PASCAL)
73C0-73C1	TIMER_LENGTH	Length of timer
73C2	TEST_SIG_NUM	Signal Code
73C3-73C4	VDP_MODE_WORD	Copy of data in the 1st 2 VDP registers
73C5	VDP_STATUS_BYTE	Contents of default NMI handler
73C6	DEFER_WRITES	Defered sprites flag
73C7	MUX_SPRITES	Multiplexing sprites flag
73C8-73C9	RAND_NUM	Pseudo random number value
73CA	QUEUE_SIZE	Size of the defered write queue
73CB	QUEUE_HEAD	Indice of the head of the write queue
73CC	QUEUE_TAIL	Indice of the tail of the write queue
73CD-73CE	HEAD_ADDRESS	Address of the queue head
73CF-73D0	TAIL_ADDRESS	Address of the queue tail
73D1-73D2	BUFFER	Buffer pointer to defered objects
73D3-73D4	TIMER_TABLE_BASE	Timer base address
73D5-73D6	NEXT_TIMER_DATA_BYTE	Next available timer address
73D7-73EA	DBNCE_BUFF	Debounce buffer. 5 pairs (old and state) of fire, joy, spin, arm and kbd for each player.
73EB	SPIN_SW0_CT	Spinner counter port#1
73EC	SPIN_SW1_CT	Spinner counter port#2
73ED	-	(reserved)
73EE	S0_C0	Segment 0 data, Controller port #1
73EF	S0_C1	Segment 0 data, Controller port #2
73F0	S1_C0	Segment 1 data, Controller port #1
73F1	S1_C1	Segment 1 data, Controller port #2
73F2-73FB	VRAM_ADDR_TABLE	Block of VRAM table pointers
73F2-73F3	SPRITENAMETBL	Sprite name table offset
73F4-73F5	SPRITEGENTBL	Sprite generator table offset
73F6-73F7	PATTERNNAMETBL	Pattern name table offset
73F8-73F9	PATTERNGENTBL	Pattern generator table offset
73FA-73FB	COLORTABLE	Color table offset
73FC-73FD	SAVE_TEMP	(no more used - in VRAM routines)
73FE-73FF	SAVED_COUNT	Copy of COUNT for PUT_VRAM and GET_VRAM