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.

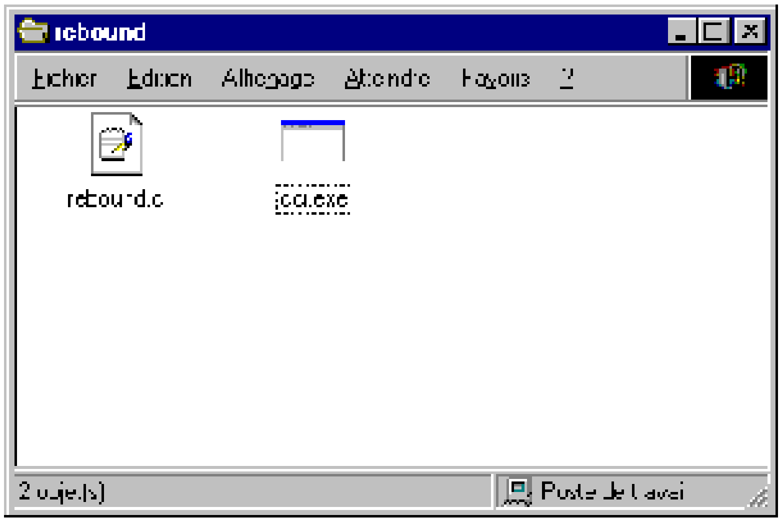
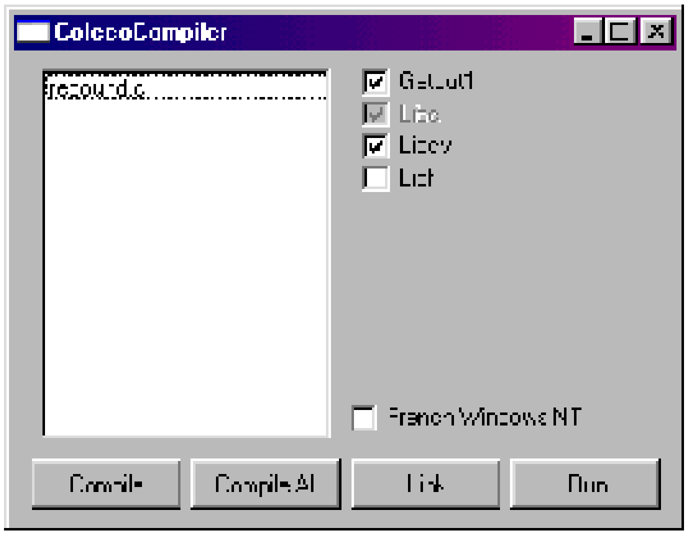


Figure 11 Rebound directory before compiling and linking the project

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.

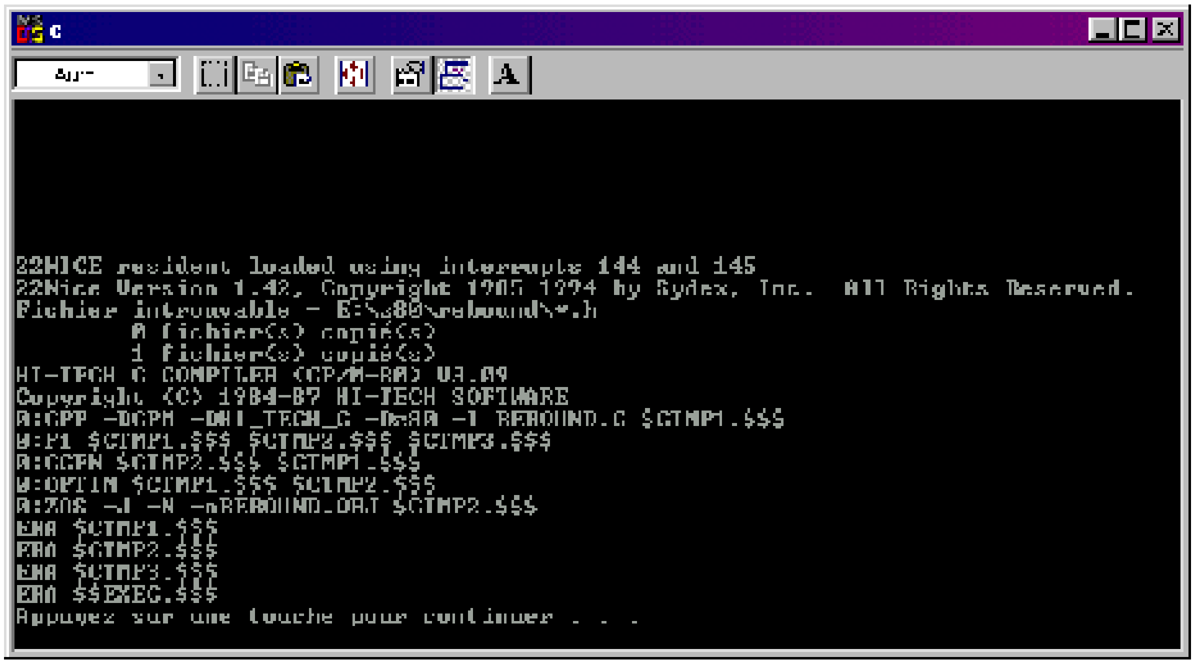


rebound.c

[✓] Getput1
[✓] Libc
[✓] Libcv
[ ] List
[ ] French Windows NT

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.

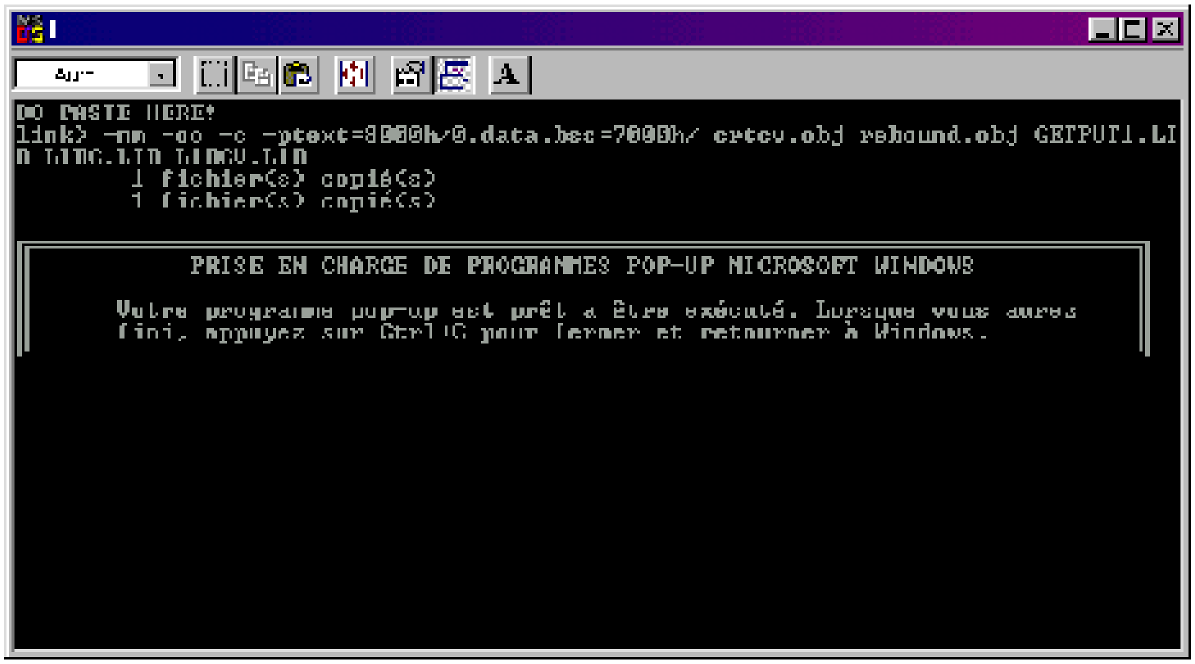


```
22NICE resident loaded using interrupts 144 and 145
22Nice Version 1.42, Copyright 1985 1994 by Sydex, Inc.  All Rights Reserved.
Fichier introuvable - E:\z80\rebound\*.h
        0 fichier(s) copié(s)
        1 fichier(s) copié(s)
HI-TECH C COMPILER (CP/M-80) V3.09
Copyright (C) 1984-87 HI-TECH SOFTWARE
A:CPP -DCPM -DHI_TECH_C -Dz80 -I REBOUND.C $CTMP1.$$$
A:P1 $CTMP1.$$$ $CTMP2.$$$ $CTMP3.$$$
A:CGEN $CTMP2.$$$ $CTMP1.$$$
A:OPTIM $CTMP1.$$$ $CTMP2.$$$
A:ZAS -J -N -oREBOUND.OBJ $CTMP2.$$$
ERA $CTMP1.$$$
ERA $CTMP2.$$$
ERA $CTMP3.$$$
ERA $$EXEC.$$$
Appuyez sur une touche pour continuer . . .
```

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.



```
DO PASTE HERE!
link> -mn -oo -c -ptext=8000h/0.data.bss=7000h/ crtcv.obj rebound.obj GETPUT1.LI
B LIBC.LIB LIBCV.LIB
        1 fichier(s) copié(s)
        1 fichier(s) copié(s)

           PRISE EN CHARGE DE PROGRAMMES POP-UP MICROSOFT WINDOWS

      Votre programme pop-up est prêt à être exécuté. Lorsque vous aurez
      fini, appuyez sur Ctrl+C pour fermer et retourner à Windows.
```

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.

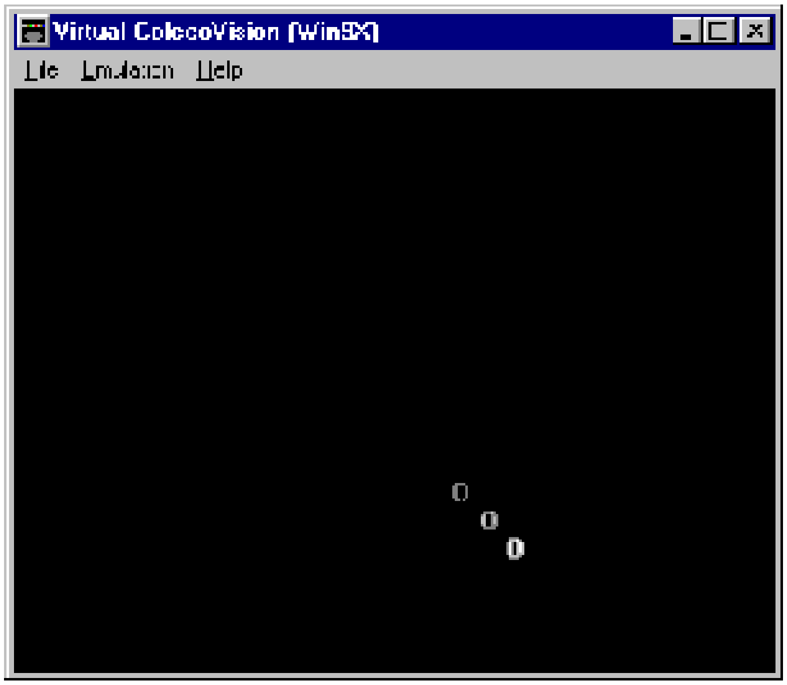


Figure 15 Running Rebound in the Virtual ColecoVision emulator for Windows

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.

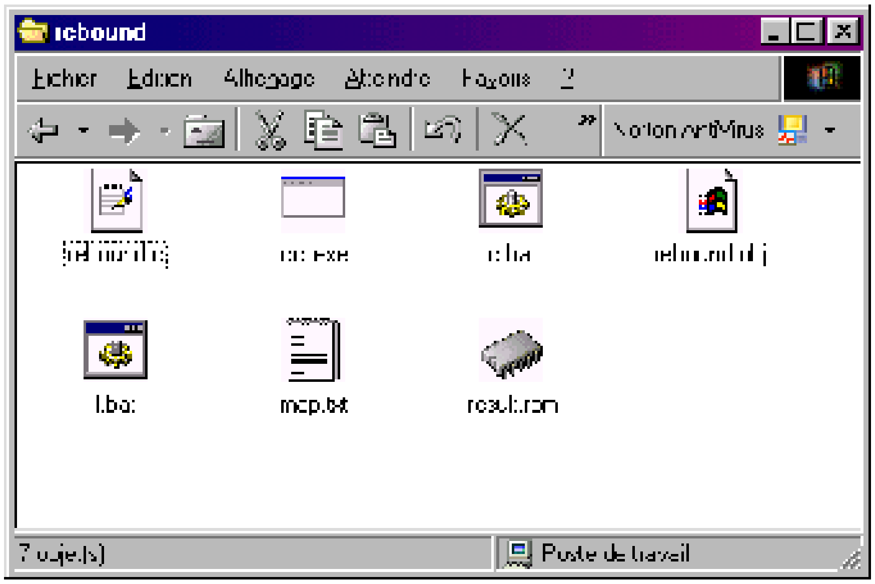


Figure 16 Rebound directory after compiling and linking the project

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();
}
```

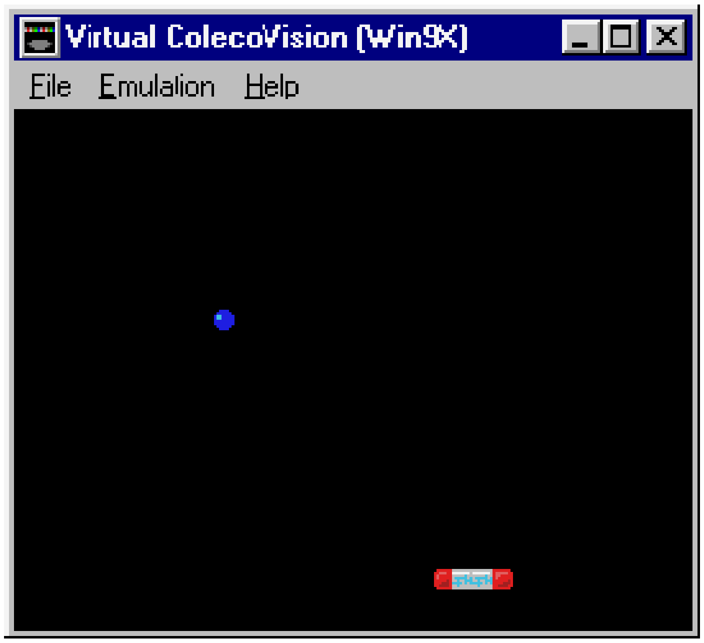


Figure 17 Smash game running with VirtualColeco emulator

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																																

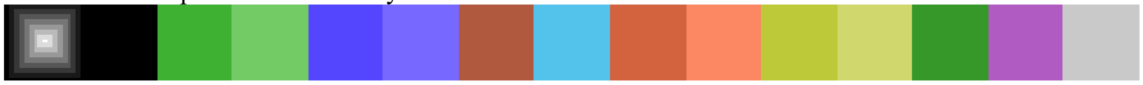
Screen mode 2 is fully supported in getput library. It's the most colorful screen mode and used mostly to show nice full screen bitmap pictures.

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

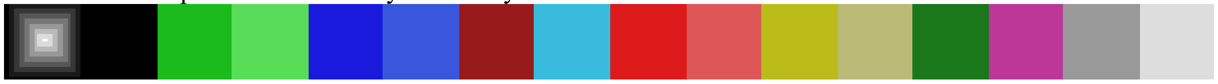
COLOR PALETTE

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

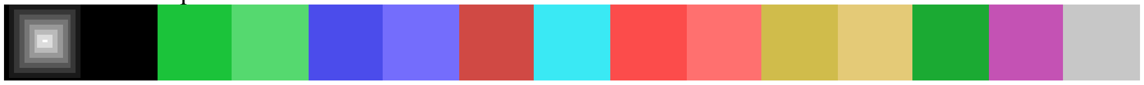
TMS9928 color palette calculated by Richard F. Drushel



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**	**T**	**62**	**3E**	**>**
31	**1F**	**M**	**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^{tm} 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