SPECTRA2

TMS9900 Arcade Game Library

for the

Texas Instruments TI-99/4A

REFERENCE MANUAL

version 1.2 - March 2018

REVISION

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Revision** |
| 08-MAR-2011 | Filip van Vooren | Initial release |
| 27-DEC-2017 | Filip van Vooren | Started updating manual |
|  |  |  |
|  |  |  |

Introduction 7

License spectra2 8

License Honeycomb Rapture 9

How it all started 10

Compatibility 11

Serviceable parts inside 11

The runtime library 12

Installation 13

Hello world! example 14

Library initialisation 18

Library startup options 21

Reset to TI title screen 22

Scratch-pad memory setup 23

Register usage 31

Equates & constants 35

The config register 36

Introducing the CONFIG register 37

The subroutine state flags 38

The stack 39

Introducing the stack 40

The POPR(0-3) and POPRT subroutine 40

spectra2 stack usage 41

Threads 42

The thread scheduler 43

The timer table 45

Highest slot in use 46

The kernel thread 46

The user hook 47

Support routines 48

Support equates 49

Register usage 49

Exiting a thread 49

Virtual keyboard 52

VIRTKB subroutine 53

The ‘ANY’ key 53

Support routines 54

Support equates 54

Example 55

Memory / Copy subroutines 57

CPYM2M / XPYM2M 58

Copy ROM/RAM to RAM 58

CPYM2V / XPYM2V 59

Copy ROM/RAM to VDP VRAM 59

CPYV2M / XPYV2M 60

Copy VDP VRAM to RAM 60

CPYG2M / XPYG2M 61

Copy GROM to RAM 61

CPYG2V / XPYG2V 62

Copy GROM to VDP VRAM 62

FILM / XFILM 63

Fill RAM with byte 63

FILV / XFILV 64

Fill VDP VRAM with byte 64

VDP low-level subroutines 65

VDWA 66

Setup VDP write address 66

VDRA 67

Setup VDP read address 67

VPUTB / XVPUTB 68

Write a single byte to VDP VRAM 68

VGETB / XVGETB 69

Read a single byte from VDP VRAM 69

VIDTAB / XIDTAB 70

Dump video mode table to VDP registers 70

PUTVR / PUTVRX 72

Load single VDP register with byte 72

PUTV01 73

Load VDP registers #0 and #1 from R14 73

SCROFF 74

Turn screen off 74

SCRON 75

Turn screen on 75

INTOFF 76

Disable VDP interrupt 76

INTON 77

Enable VDP interrupt 77

SMAG1X 78

Set sprite magnification 1X 78

SMAG2X 79

Set sprite magnification 2X 79

S8X8 80

Set sprite size to 8x8 pixels 80

S16X16 81

Set sprite size to 16x16 pixels 81

GTCLMN 82

Get number of columns per row 82

YX2PNT 83

Get VDP Pattern-Name-Table address for cursor YX position 83

YX2PX / YX2PXX 84

Get pixel position for cursor YX position 84

PX2YX 86

Get tile YX position for pixel YX position 86

VDP tiles & patterns subroutines 88

LDFNT 89

Load TI-99/4A character font from GROM into VRAM 89

PUTSTR 90

Put length-byte prefixed string at cursor position 90

PUTAT 92

Put length-byte prefixed string at position Y,X 92

HCHAR 93

Repeat characters horizontally at position Y,X 93

VCHAR 94

Repeat characters vertically at position Y,X 94

FILBOX 95

Fill box with characters at position Y,X 95

PUTBOX 97

Put length-prefixed string in box at position Y,X 97

MKNUM 99

Convert unsigned number to right-justified string 99

PUTNUM 101

Put unsigned number on screen 101

Sound & speech subroutines 103

MUTE 104

Mute all sound generators and clear sound pointer 104

MUTE2 105

Mute all sound generators 105

SDPREP 106

Prepare for playing sound 106

SDPLAY 107

Run the sound player 107

SPSTAT 108

Read status register byte from speech synthesizer 108

SPCONN 109

Check if speech synthesizer is connected 109

SPPREP 110

Prepare for playing speech 110

SPPLAY 111

Run the speech player 111

Keyboard & joystick subroutines 112

VIRTKB 113

The virtual keyboard implementation 113

Thread scheduler subroutines 114

TMGR 115

The thread scheduler 115

MKSLOT 116

Allocate timer slots 116

CLSLOT 118

Clear allocated timer slot 118

KERNEL 119

The kernel thread 119

MKHOOK 120

Allocate the user hook 120

Miscellaneous subroutines 121

POPR(0-3) or POPRT 122

Pop registers & return to caller 122

RND / RNDX 123

Generate random number 123

RUNLIB 124

Initialize spectra2 runtime library 124

Appendix: examples & source code 125

# Introduction

### License spectra2

This software is provided 'as-is', without any expressed or implied warranty. In no event will the author(s) be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose. If you use this software in a product, an acknowledgment in the product documentation would be deeply appreciated but is not required.

In the case of the spectra2 source code, permission is granted to anyone to alter it, subject to the following restrictions:

* The origin of this software must not be misrepresented; you must not claim that you wrote the original software.
* Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
* This notice may not be removed or altered from any source distribution.

### License Honeycomb Rapture

Honeycomb Rapture ©2009

written by Owen Brand

Author does hereby assign conditional rights to Honeycomb Rapture for use in the SPECTRA2 development library. This work may be altered and redistributed if the following guidelines are followed:

* Any re-distribution of Honeycomb Rapture must include reference to the author (Owen Brand) and to the SPECTRA2 libraries.
* Any monetary compensation received by an individual for distributing this or any version of Honeycomb Rapture must be reported to the author (owenbrand@rocketmail.com)

### How it all started

The idea for the initial implementation of SPECTRA was born while I was working on Pitfall!, my first homebrew game for the Texas Instruments TI-99/4A.

During that time I was studying the Colecovision disassembly of the game very closely and I learned that the game called multiple subroutines stored in the consoles’ built-in ROM. Doing some research in the internet revealed that this Colecovision ROM contains a BIOS; a collection of game routines called OS7.

Thanks to the wonderful work of Daniel Bienvenu who documented most of these subroutines, I was able to understand what they were actually for. It inspired me to start working on a similar library for the TI-99/4A Home Computer.

I wanted an open-source library that allows me to concentrate on the development of the game itself, without having to start writing all subroutines from scratch over and over again.

SPECTRA2 takes that approach one step further and acts as a miniature operating system for running homebrew games and software from the cartridge space on the unexpanded TI-99/4A.

The library is designed for minimal memory usage, the main target being the TI-99/4A with its 256 bytes of scratchpad memory.

### Compatibility

SPECTRA2 is a library targeted for cross-development on a PC compatible environment. Even on an older PC, assembly times are so fast that I don’t see much benefit in reusing already assembled object files. I do see some huge benefits in programming TMS9900 assembly on your desktop or netbook:

Besides the fact that you can always carry your development environment with you (e.g. on a USB stick), the biggest advantage for me are the TI-99/4A emulators and their powerful built-in debuggers.

Using such an environment will seriously speed-up your development cycle, while allowing more flexibility.

The source code of SPECTRA2 is compatible with Burrsofts’ Asm994A Assembler V3.008

This great cross-assembler for Windows is not part of SPECTRA2, but can be obtained directly at BurrSoft[1]

The assembler is part of the Win994A emulator package and is considered freeware by the author. For further details and verification please check the license conditions at the mentioned BurrSoft page.

### Serviceable parts inside

The library has been tested to some extent, but comes without any warranties whatsoever. There may still be plenty of bugs inside and if you find any, let me know and I’ll try to fix them.

# The runtime library

### Installation

The installation process is very easy, download the spectra2 zip-file from [**http://www.retroclouds.de/spectra2/spectra2.zip**](http://www.retroclouds.de/spectra2/spectra2.zip) and extract/copy all files to your working directly.

If you want a minimal installation, then it’s sufficient to copy the **runlib.a99** file.

This assembly source file is the core of the library and contains all required equates and subroutines for running your first program.

### Hello world! example

Take a look at the below “Hello World!” program. This is pretty much how the assembly source should be laid out.

We can identify 4 major parts:

* A) The cartridge header
* B) Include required assembly source files
* C) Equates for controlling library startup behaviour
* D) The main program

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

AORG >6000 ; cartridge space >6000 - >7FFF

\*--------------------------------------------------------------

\* A - Cartridge header

\*--------------------------------------------------------------

GRMHDR BYTE >AA,1,1,0,0,0

DATA PROG

BYTE 0,0,0,0,0,0,0,0

PROG DATA 0

DATA RUNLIB

HW BYTE 12 ; # of chars in ‘HELLO WORLD!’

TEXT 'HELLO WORLD!'

\*--------------------------------------------------------------

\* B - Include required files

\*--------------------------------------------------------------

COPY "D:\Projekte\spectra2\tms9900\runlib.a99"

\*--------------------------------------------------------------

\* C - SPECTRA2 startup options

\*--------------------------------------------------------------

SPVMOD EQU GRAPH1 ; Video mode. See VIDTAB for details.

SPFONT EQU FNOPT7 ; Font to load. See LDFNT for details.

SPFCLR EQU >F0 ; Foreground/Background color for font.

SPFBCK EQU >08 ; Screen background color.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* D - Main

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MAIN BL @PUTAT

DATA >0B0A,HW ; "Hello World!" on row >0B, column >0A

B @TMGR ; Handle FCTN-QUIT key, etc.

END

This example is included as file **example1.a99** in the spectra2 samples directory.

**A) The cartridge header**

The TI cartridge space is in the range from >6000 to >7FFF. It’s important to know that the cartridge header must start at >6000 in order to be recognized as a valid header by the TI Operating System.

For most projects it’s sufficient to change the program title for the TI selection screen. This string has to be prefixed with a length byte and may not contain any lower-case characters.

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

AORG >6000 ; cartridge space >6000 - >7FFF

\*--------------------------------------------------------------

\* Cartridge header

\*--------------------------------------------------------------

GRMHDR BYTE >AA,1,1,0,0,0

DATA PROG

BYTE 0,0,0,0,0,0,0,0

PROG DATA 0

DATA RUNLIB

HW BYTE 12 ; # of chars in HELLO WORLD!

TEXT 'HELLO WORLD!'

The TI cartridge selection screen should look as seen in the screenshot below:



**B) Include required files**

Use the COPYdirective to include the spectra2 runtime library **runlib.a99** in your source code. Change the file path so that it matches the directory containing your version of the runlib.a99 file.

\*--------------------------------------------------------------

\* Include required files

\*--------------------------------------------------------------

COPY "D:\Projekte\spectra2\tms9900\runlib.a99"

**C) Equates for controlling library startup behaviour**

The below equate values are used for initializing the TI-99/4A environment. The specified values are inserted in the source code of the spectra2 initialisation routine and video mode table during the assembly process.

\*--------------------------------------------------------------

\* SPECTRA2 startup options

\*--------------------------------------------------------------

SPVMOD EQU GRAPH1 ; Video mode. See VIDTAB for details.

SPFONT EQU FNOPT7 ; Font to load. See LDFNT for details.

SPFCLR EQU >F0 ; Foreground/Background color for font.

SPFBCK EQU >08 ; Screen background color.

* **SPVMOD EQU GRAPH1**

This directive is used for initializing the VDP in graphic mode 1 (32 columns mode). Actually GRAPH1 is the address of the included video mode table. The table is used by the **VIDTAB** subroutine for setting all 7 VDP registers.

See the VIDTAB subroutine on page 70 for further details.

* **SPFONT EQU FNOPT7**

Load the TI-Basic upper and lower case font from GROM and make the font bold. This is handled by the **LDFNT** subroutine.

See the LDFNT subroutine on page 89 for further details.

* **SPCLR EQU >F0**

Set foreground color to white.

* **SPFBCK EQU >08**

Set background color to red.

For further details also refer to the section “Library startup options” on page 21.

**D) The main program**

After the library has completely initialized, it will automatically do a “B @MAIN” for returning control to the main program.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Main

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MAIN BL @PUTAT

DATA >0B0A,HW ; "Hello World!" on row >0B, column >0A

B @TMGR ; Handle FCTN-QUIT key, etc.

END

* **BL @PUTAT**

By calling PUTAT with the specified “DATA >0B0A,HW” statement, the cursor is set to row >0B, column >0A. It then displays the length-byte prefixed string ‘HELLO WORLD!’, which was also used in the cartridge header.

See the PUTAT subroutine on page 92 for details on how to display a string.

* **B @TMGR**

Control is now handed over to TMGR, the thread scheduler. This subroutine is the main-loop for all programs using the spectra2 library. It does many tasks, such as scanning the keyboard, handling FCTN-QUIT, running speech & sound player, etc.

See the thread scheduler section on page 42 for further details.

* **END**

The assembler END directive.

### Library initialisation

The initialisation subroutine **RUNLIB** is the entry point into the spectra2 library. This subroutine is normally called via a “**B @RUNLIB**” upon program start.

If the program is in the cartridge space, then RUNLIB gets called when the corresponding option is chosen from the TI cartridge selection screen. For this to work, it’s required that the address of RUNLIB is used in the cartridge header.

**See the “Hello World!” program on page 14 for an example.**

The tasks done by RUNLIB are:

1) Disable interrupts and set workspace to >8300.

2) Clear CPU scratch-pad memory from >8306->83FF.

3) Set random seed and determine if VDP handles PAL or NTSC.

4) Copy machine code into scratch-pad memory.

5) Determine TI-99/4A operating system version.

6) Initialize used registers, set defaults and mute the sound generators.

7) Setup VDP registers, clear 16K of VRAM, load color table and startup font.

9) Jump into the main program via “**JMP MAIN**”.

Now let’s take a look at all these steps in detail:

**1) Disable interrupts and set workspace to >8300.**

To avoid any conflicts with the ISR routine in the consoles’ OS, interrupts are disabled. The register workspace is then set to the top of scratchpad memory (>8300).

**2) Clear CPU scratchpad memory from >8306 - >83FF.**

In the previous step the workspace was set to >8300. We now clear all scratch-pad memory starting at >8306 (location of register R4).

**3) Set random seed and determine if VDP handles PAL or NTSC.**

The init routine copies the random seed set by the monitor OS into its proper memory location. Additionally the init subroutine now determines if the VDP is a PAL or NTSC version. It does that by continuously checking the VDP interrupt flag while running a loop counter.

The result of the test (PAL or NTSC) is stored in bit 12 of the CONFIG register (R12).

Note that this step uses registers R1-R3 for temporary storage.

**See the VDP Programmers Guide[1] for further details on the VDP interrupt flag.**

**4) Copy machine code into scratch-pad memory.**

In this step 6 bytes of machine code are copied into the scratch-pad memory location >8320. The machine code is mainly used for speeding up the filling and copying of large memory blocks between CPU and VDP memory. Having this code in scratch-pad memory reduces wait-states.

**See Thierry Nouspikel’s Technical pages[2] for further details on scratch-pad memory and the multiplexer.**

**5) Determine TI-99/4A operating system version.**

The GROM memory in the TI-99/4(A) console is scanned to determine the operating system version.

The result is stored in bit 10 of the CONFIG register.

If the OS can’t be determined, then spectra2 assumes it’s running on an unsupported platform.

It’s important to know, that spectra2 doesn’t support the original TI-99/4 (without a) Home Computer.

**This step will exit to the TI title screen if an unsupported system such as the TI-99/4 is detected.**

**6) Initialize used registers, set defaults and mute the  
sound generators.**

* The registers R1-R3 used in the previous steps are now cleared.
* The stack register (R9) is loaded with address >8400 (that’s outside scratch-pad memory. You need to do a “DECT STACK” first).
* The register R15 is loaded with the address of the VDP data write port.
* All sound generators are muted.

**7) Setup VDP registers and clear 16K of VRAM.**

* All VDP registers are set according to the values in the specified video mode table. This is handled by calling the VIDTAB subroutine using the specified equates.
* The 16K of VRAM gets cleared.
* The color table is loaded into VRAM using the specified equates.
* The startup font is loaded into VRAM using the specified equates.

See the Library startup options on page 21 for further details on the equate values to use.

**8) Hand-over control to MAIN**

The initialisation has completed and control is given to the MAIN subroutine by issuing a “B @MAIN”.

Note that register R0 is not cleared during the library initialisation. This can be useful for passing-through a value from your custom pre-init routine to MAIN.

See file **“/samples/example6.a99”** foran example.

## Library startup options

There are a few equates that must be set in the main source file. They control the spectra2 startup options such as:

VDP video mode, font style, etc.

|  |  |
| --- | --- |
| **Equate** | **Description** |
| SPVMOD | Address of video mode table to use on startup.  Use GRAPH1 for 32 columns mode (with sprites). Use TXTMOD for 40 columns mode (no sprites).  It’s also possible to use your own video mode table. |
| SPFONT | Built-in system font to load on startup.  Note that there are no fonts included in RUNLIB.  The fonts are loaded into VDP memory from the GROMs in the TI-99/4A console.  Possible values to use are:  NOFONT ; Do not load font on startup  FNOPT1 ; Load TI title screen font  FNOPT2 ; Load upper case font  FNOPT3 ; Load upper/lower case font  FNOPT4 ; Load lower case font  FNOPT5 ; Load TI title screen font & make fat  FNOPT6 ; Load upper case font & make fat  FNOPT7 ; Load upper/lower case font & make fat  FNOPT8 ; Load lower case font & make fat |
| SPFCLR | Foreground and background color for textmode  This value goes into VDP#register 7 when using a textmode video table. The SPFCLR equate is not used in any of the graphics video mode tables. |
| SPFBCK | Background color for graphic modes.  This value goes into VDP#register 7 when using a graphics video mode table. The SPFBCK equate is not used in the TXTMOD video mode table. |

For further details see documentation on VIDTAB (page 70) and LDFNT (page 89).

### Reset to TI title screen

You can safely exit the program and return to the TI title screen, by setting register R1 to >FFFF and doing a “**B @RUNLI1**”. The advantage over a “BLWP @>0000”, is that scratchpad memory gets properly cleared first.

SETO R1

B @RUNLI1 ; Exit to title screen

### Scratch-pad memory setup

The TI-99/4A has 256 bytes of memory on the motherboard, which is referred to as “scratch-pad” memory. It has the address range >8300 - >83FF and is on the 16-bit bus. It can be accessed without wait states and is really fast compared to the memory on the 8-bit bus (e.g. 32K memory expansion). You should use scratch-pad memory where possible.

Of the 256 bytes available, spectra2 uses 60 bytes for storing the register workspace and all variables it needs for housekeeping tasks, etc.

Depending on the features used, some memory can be recovered and used for other purposes. This is controlled by multiple flags in the CONFIG register.

Let’s take a detailed look at each of the used memory locations.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | |  | | --- | |  | |  |
| >8300 | A | | | | | | | | | | | | | | | | Register workspace >8300 - >8327 | |
| >8310 |  |  |
| >8320 | B | | | | | | | | C | | D | | E | | F | | Machine code & runtime variables  >8328 - >833B | |
| >8330 | G | | H | | I | | J | | K | | L | |  |  |  |  |
| >8340 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >8350 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >8360 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >8370 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >8380 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >8390 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >83A0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >83B0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >83C0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >83D0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >83E0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| >83F0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| A | Register workspace | | | | |  |  |  |  |  | >8300 - >831F | | | |  |  | 32 |  |
| B | Machine code for loops/speech/... | | | | | | | |  |  | >8320 - >8327 | | | |  |  | 8 |  |
| C | PNT BASE address | | | |  |  |  |  |  |  | >8328 - >8329 | | | |  |  | 2 |  |
| D | Cursor YX position | | | | |  |  |  |  |  | >832A - >832B | | | |  |  | 2 |  |
| E | Timers: Address of timer table | | | | | | |  |  |  | >832C - >832D | | | |  |  | 2 |  |
| F | Timers: Address of user hook | | | | | | |  |  |  | >832E - >832F | | | |  |  | 2 |  |
| G | Timers: Internal use | | | | |  |  |  |  |  | >8330 - >8331 | | | |  |  | 2 |  |
| H | Virtual keyboard flags | | | | | | |  |  |  | >8332 - >8333 | | | |  |  | 2 |  |
| I | Sound player: Address of tune | | | | | | |  |  |  | >8334 - >8335 | | | |  |  | 2 |  |
| J | Sound player: Internal use | | | | | |  |  |  |  | >8336 - >8337 | | | |  |  | 2 |  |
| K | Speech player: Address of LPC data | | | | | | | |  |  | >8338 - >8339 | | | |  |  | 2 |  |
| L | Seed for random subroutine | | | | | | | |  |  | >833A- >833B | | | |  |  | 2 |  |
|  | **Size in bytes** | | |  |  |  |  |  |  |  |  |  |  |  |  |  | **60** |  |

**A) Register workspace (>8300 - >8319)**

There are only 3 hardware registers in a 9900 CPU: PC (program counter), WP (workspace pointer), ST (status register). All other registers are stored in CPU memory.

That is why we need 32 bytes of scratchpad-memory for holding the 16 (16-bit) registers R0-R15.

See section “Register usage” on page 31 for further details.

**B) Machine code (>8320 - >8327)**

The below 8 bytes of machine code are copied into scratchpad memory >8320 upon library startup. The machine code is mainly used for speeding up loops. It’s used by several spectra2 low-level routines (e.g. CPYM2V)

You can use the “tight-loop” routine for your own purpose by overwriting 2 bytes of machine code at >8320. Note that the routine should be called with BL @>8320. It expects TMP2 (R6) to contain the number of times the loop should be executed.

\*--------------------------------------------------------------

\* ; Machine code for tight loop.

\* ; The MOV operation at MCLOOP must be injected by the

\* ; calling routine.

\*--------------------------------------------------------------

\* DATA >???? ; \ MCLOOP MOV ...

MCCODE DATA >0606 ; | DEC R6 (TMP2)

DATA >16FD ; | JNE MCLOOP

DATA >045B ; / B \*R11

When running the speech player (SPPLAY), the following 4 bytes of machine code get copied to >8320, overwriting part of the “tight loop” code. The tight loop code is automatically restored upon player exit.

\*--------------------------------------------------------------

\* ; Machine code for reading from the speech synthesizer

\* ; The SRC instruction takes 12 uS for execution in scratchpad RAM.

\* ; Is required for the 12 uS delay. It destroys R5.

\*--------------------------------------------------------------

SPCODE DATA >D114 ; \ MOVB \*R4,R4 (TMP0)

DATA >0BC5 ; / SRC R5,12 (TMP1)

**C) PNT base address (>8328 - >8329)**

This memory location holds the address of the Pattern Name Table (PNT) in VRAM. The PNT table in VRAM contains all tiles to display on screen.

The address is automatically set by spectra2 if a video mode table is loaded with the VIDTAB subroutine and is used by many of the VDP subroutines included in the library (e.g. YX2PNT).

**Equates**

WBASE EQU >8328 ; 02 - PNT base address

You can also manually set the (using the WBASE equate) for creating multiple “virtual” screens. Basically you’d set it to a VDP memory location outside the window addressed by the VDP#2 write-only register.

You can then use all available spectra2 subroutines for drawing the screen.

For instant display, you then only have to switch the VDP#2 write-only register to the new address.

Please refer to the VDP Programmer’s Guide page for further details on the Pattern Names Table.

**D) Cursor YX position (>832A - >832B)**

This is the memory address used for holding the cursor position. There is no real cursor in spectra2, but many VDP routines in the library use this location for calculating the VRAM target address of the corresponding PNT entry.

**Equates**

WYX EQU >832A ; 02 - Cursor YX position

BY EQU WYX ; Cursor Y position

BX EQU WYX+1 ; Cursor X position

Note that the cursor position always starts with Y=0, X=0. So if you want to display something on row 6, column 10 you would load >0509 into memory location @WYX.

Here’s an example on how to use the cursor for displaying the string “Hello World!” on row 6, column 10.

TEST1 LI R0,>0509 ; Row 6, column 10  
 MOV R0,@WYX ; Load cursor

BL @PUTSTR ; Display string

DATA HW ; String to display

JMP $ ; Soft-halt

HW BYTE 12

TEXT ‘HELLO WORLD!’

**E) Timers: Address of timer table (>832C - >832D)**

This memory address points to a table in CPU memory that contains required base data when running timers. You normally fill the timer table by using the MKSLOT routine.

**Equates**

WTITAB EQU >832C ; 02 - Address of timer table

See section “Thread Scheduler” on page 43 for further details.

**F) Timers: Address of user hook (>832E - >832F)**

This memory address contains the address of the user hook, a user-supplied subroutine that is executed **at least** every 1/60th (NTSC) or 1/50th (PAL) of a second.

The idea is that you use the user hook for stuff that isn’t bound to the VDP interrupt.

**Equates**

WTIUSR EQU >832E ; 02 - Address of user hook

See section “Thread Scheduler” on page 43 for further details.

**G) Timers: Internal use (>8330 - >8331)**

Used by the Thread Scheduler subroutine (TMGR) for storing internal variables.

**Equates**

WTITMP EQU >8330 ; 02 – Internal use

See the “Thread Scheduler” section on page 43 for further details.

**H) Virtual keyboard flags (>8332 - >8333)**

This memory location holds 16 1-bit flags, representing the keys pressed on the spectra2 virtual keyboard. That is when calling the VIRTKB subroutine.

**Equates**

WVRTKB EQU >8332 ; 02 – Virtual keyboard flags

See the “Virtual keyboard” section on page 52 for further details.

**I) Sound player: Address of tune (>8334 - >8335)**

Points to a table in CPU memory or VRAM containing the the sound list data for playback with the built-in sound player routine (SPPLAY).

**Equates**

WSDLST EQU >8334 ; 02 – Tune address

See the “Sound & speech subroutines” section on page 103 for further details.

**J) Sound player: Temporary use (>8336 - >8337)**

Contains some internal variables used by the sound player routine (SDPLAY).

**Equates**

WSTMP EQU >8336 ; 02 – Tune address

See the “Sound & speech subroutines” section on page 103 for further details.

**K) Speech player: Address of LPC data (>8338 - >8339)**

The spectra2 library offers the possibility to playback speech samples, when a speech synthesizer is connected to the TI-99/4A console. Speech samples are encoded in LPC format (Linear Predictive Coding) and must be stored in CPU memory for playback with the SPPLAY subroutine.

This memory location holds the address of the LPC data stream.

**Equates**

WSPEAK EQU >8338 ; 02 – Address of LPC data

See the “Sound & speech subroutines” section on page 103 for further details.

**L) Seed for random subroutine (>833A - >833B)**

For generating pseude-random numbers we need a seed value. The WSEED memory location is automatically setup by the spectra2 initialisation routine. It copies the seed value set by the monitor OS.

**Equates**

WSEED EQU >833A ; 02 – Seed for random subroutine

See the RND subroutine on page 123 for further details.

### Register usage

The 16 available registers play a very important role when using the spectra2 library. Some of the registers have a special purpose, e.g. for passing parameters or speeding-up memory access.

Let’s take a detailed look at each of the registers.

* **General purpose registers (R0 … R3)**

The registers R0 – R3 aren’t used by any of the subroutines in the spectra2 library.

With the only exception being that registers R1-R3 are used during the library initialisation. Nonetheless, once your program (MAIN) takes over, you’ll have R0-R3 to your exclusive disposal.

* **Temporary registers (R4 … R8)**

The registers R4 … R8 are registers used for temporary storage of parameters, counters, etc.

These registers should never be addressed with their  
R4 … R8 label.

Instead they should be referred to using the TMP0 … TMP4 label.

**Equates**

TMP0 EQU R4 ; Temp register 0

TMP1 EQU R5 ; Temp register 1

TMP2 EQU R6 ; Temp register 2

TMP3 EQU R7 ; Temp register 3

TMP4 EQU R8 ; Temp register 4

**Keep in mind, that when calling any of the spectra2 subroutines, it is likely that some or all of the temporary registers will be destroyed.**

* **The stack pointer or temporary register TMP5 (R9)**

Now for sure you already know that there is no hardware stack pointer in a TMS9900 CPU. As a workaround the stack pointer can be simulated by using the general purpose register R9.

Depending on your requirement you should use one of the below equates:

**Equates**

STACK EQU R9 ; Stack pointer

TMP5 EQU R9 ; Temp register 5

Note that when the runtime library gets initialized, R9 is loaded with the value >8400.

Please refer to page 40 for further details on stack usage.

If you decide not to use a stack, then you can use R9 as temporary register **TMP5**.

* **Highest slot in use & internal counter for timers (R10)**R10 is exclusively used by the thread scheduler.
  + The high byte of R10 keeps track of the highest slot used in the thread scheduler timer table.
  + The low byte of R10 is the thread scheduler tick counter and is updated every 1/50th (VDP) or 1/60th (NTSC) of a second.

Please refer to page 43 for further details on the thread scheduler.

* **Subroutine return address (R11)**

Contains the subroutine return address when issuing a branch-and-link **"BL xxxx".**

* **The CONFIG register (R12)**

R12 is the spectra2 configuration register. It’s used for storing 16 individual status flags and should be referenced using the CONFIG label.

**Equates**

CONFIG EQU R12 ; SPECTRA configuration register

Please refer to page 37 for further details on the bit flags available in the CONFIG register.

* **Copy of VDP status byte & counter for sound player (R13)**

R13 is exclusively used by spectra2:

* + The high byte of R13 contains a copy of the VDP status register byte. The byte is continuously copied by the TMGR thread scheduler.
  + The low byte of R13 is used as an internal counter when the sound player is running.

**Equates**

BVDPST EQU WS1+26 ; Copy of VDP status register (HI byte R13)

* **Copy of VDP register #0 and VDP register #1 (R14)**R14 is exclusively used by spectra2.
  + The high byte of R14 contains a copy of VDP write-only register #0.
  + The low byte of R14 contains a copy of VDP write-only register #1.

**Equates**

VDPR01 EQU R14 ; Copy of VDP#0 and VDP#1 bytes

VDPR0 EQU WS1+28 ; High byte of R14. Is VDP#0 byte

VDPR1 EQU WS1+29 ; Low byte of R14. Is VDP#1 byte

This register is used for easily doing bit-operations when setting/getting current video mode, sprite magnification, etc.

See the sections “VDP low level subroutines” on page 65 and “VDP tiles & patterns subroutines” on page 88 for further details on the available VDP support routines.

* **VDP write address or temporary register (R15)**

R15 contains the address of the VDP read or write port. By storing the port address in the register, it’s possible to write more compact and faster code. The VDP low-level routines in spectra2 use this register a lot.

**Equates**

VDPRW EQU R15 ; Contains VDP read/write address

TMP6 EQU R15 ; Temp register 6

VDPR EQU >8800 ; VDP read data window address

VDPW EQU >8C00 ; VDP write data window address

VDPS EQU >8802 ; VDP status register

VDPA EQU >8C02 ; VDP address register

Note that when a spectra2 VDP low-level routine is called, it will load R15 with VDPW or VDPR depending if writing or reading.

It’s also possible to use R15 as temporary register TMP6. However you’ll have to ensure, that R15 is reloaded with the correct VDP write or read address before calling any of the VDP subroutines.

### Equates & constants

A large set of equates is included in the library source code.

Please use the equates instead of the corresponding values where possible.

That way, the migration to a new spectra2 release will be less cumbersome.

In particular equates exist for:

* Registers
* Temporary registers (R3-R9)
* Hi- or Lo- byte of all registers (R0-R15)
* Stack pointer (R9)
* Special purpose registers (R10-R15)
* Bit-level operations
  + All flags in the config register (R12)
  + Bit 0-15 of a word
* Spectra2 routines & parameters
  + Virtual keyboard
  + Sound player options
  + Speech player options
* Spectra2 memory
  + Cursor YX position
  + Task scheduler variables
  + Virtual keyboard, Sound/Speech player
  + …
* Hardware
  + VDP & sound addresses
  + GROM, Speech, etc.

The spectra2 library also includes some constants

* + For setting bits 0-15 of a word
  + For loading a byte with decimal value 0-9

For further details please check the runlib.a99 file (spectra2 source code).

# The config register

### Introducing the CONFIG register

Many of the features in spectra2 are controlled by 16 individual bit flags of the configuration (CONFIG) register. This is currently mapped to R12 but that might change in the future. Therefore please use the CONFIG label instead.

**Equates**

CONFIG EQU R12 ; SPECTRA configuration register

The reason why we use a register instead of a memory location, is that a register allows for easy bit compare and manipulation.

0 1 2 3 4 5 6 7 8 9 A B C D E F

| | | | | | | | | | | | | | | |

| | | | | | | | | | | | | | | +-- 15 Sound player: tune source (1=ROM/RAM, 0=VDP)

| | | | | | | | | | | | | | |

| | | | | | | | | | | | | | +---- 14 Sound player: repeat tune

| | | | | | | | | | | | | |

| | | | | | | | | | | | | +------ 13 Sound player: enabled

| | | | | | | | | | | | |

| | | | | | | | | | | | +-------- 12 VDP9918 latched sprite collision flag

| | | | | | | | | | | |

| | | | | | | | | | | +---------- 11 Keyboard: ANY key pressed

| | | | | | | | | | |

| | | | | | | | | | +------------ 10 \*\*free\*\*

| | | | | | | | | |

| | | | | | | | | +-------------- 9 Timer: Kernel thread enabled

| | | | | | | | |

| | | | | | | | +---------------- 8 Timer: Block kernel thread

| | | | | | | |

| | | | | | | +------------------ 7 Timer: User hook enabled

| | | | | | |

| | | | | | +-------------------- 6 Timer: Block user hook

| | | | | |

| | | | | +---------------------- 5 Speech player: speech synthesizer found

| | | | |

| | | | +------------------------ 4 Speech player: busy

| | | |

| | | +-------------------------- 3 Speech player: enabled (\*)

| | |

| | +---------------------------- 2 VDP9918 version (1=PAL/50, 0=NTSC/60) (\*)

| |

| +------------------------------ 1 F18A found

|

+-------------------------------- 0 Subroutine state flag

(\*) = Read-only flag. Set by RUNLIB subroutine

### The subroutine state flags

Bit 0 in the CONFIG register are used to control the behaviour of some of the subroutines in the spectra2 library. They can be seen as toggles that turn certain features on/off.

The MKNUM subroutine for example uses bit 0 in the CONFIG register to determine if the converted number should be displayed on screen.

You can use bit 0 of the CONFIG register for your own purposes. Just keep in mind that they may be overwritten when calling some of the spectra2 subroutines.

For further details please check the runlib.a99 file (spectra2 source code).

# The stack

### Introducing the stack

Now for sure you already know that there is no hardware stack pointer in a TMS9900 CPU. As a workaround the stack pointer can be simulated by using one of general purpose registers.

In its current release spectra2 uses register R9 as stack pointer, that might change in a future release. Please use the STACK equate instead of R9.

**Equates**

STACK EQU R9 ; Stack pointer

TMP5 EQU R9 ; Temp register 5

The stack grows toward low memory. **It means you have to decrease the stack pointer before pushing a value on the stack.**

MYSUB DECT STACK   
 MOV R11,\*STACK ; Push R11

DECT STACK

MOV R0,\*STACK ; Push R0

DECT STACK

MOV R1,\*STACK ; Push R1

...

B @POPR1 ; Pop R1,R0,R11 and return to caller

When the runtime library gets initialized, the STACK pointer (R9) is loaded with >8400, that is just above scratch-pad memory. By issuing a “DECT STACK” before pushing, we get to address >83FE which is the highest address in scratch-pad memory. Did I mention that >8400 is the address of the sound port? You’ll get strange results when trying to push a value to that address...

### The POPR(0-3) and POPRT subroutine

Instead of writing inline code upon subroutine exit, you can branch to the POPR(0-3) subroutine to pop the registers from the stack and return to the calling program.

If for example, you pushed registers R11, R0, R1 & R2 on the stack, you would do a “B @POPR2” to pop the registers and exit your subroutine.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* POPR. - Pop registers & return to caller

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* B @POPRG.

\*--------------------------------------------------------------

\* REMARKS

\* R11 must be at stack bottom

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

POPR3 MOV \*STACK+,R3

POPR2 MOV \*STACK+,R2

POPR1 MOV \*STACK+,R1

POPR0 MOV \*STACK+,R0

POPRT MOV \*STACK+,R11

B \*R11

### spectra2 stack usage

It’s important to know, that none of the routines in the spectra2 library internally make use of the stack. This is a major difference compared to the initial spectra release, which fully relied on the presence of a stack.

The reason for this change, is that spectra2 is targeting the unexpanded TI-99/4A with its 256 bytes of scratch-pad memory. We don’t want to waste any memory and instruction cycles on pushing/popping values from the stack.

That for sure doesn’t mean that a stack is bad. As a matter of fact, based on the complexity of your game project, it’s probably a good idea to use a stack. That is especially true if you have a bunch of nested calls.

You can use R9 as temporary register **TMP5,** if you decide not to use a stack.

# Threads

### The thread scheduler

When writing arcade games, one is faced with the difficulty of having to control different things at the same time. You have to read the keyboard, move sprites, draw the screen, run some game logic, … all at the same time.

For your game to run fluently, you have to ensure that all of this is handled in a short time frame.

Now even though TMS9900 assembly language is lightning fast, it can be very cumbersome writing such routines.

To help with that, a thread scheduler (TMGR) is included in spectra2. Basically the scheduler acts as your programs’ main loop, periodically calling the subroutines you specify.

In order for this to work, the scheduler expects that the called subroutine will end in a timely matter. However, it can’t enforce it. **A poorly designed subroutine may “hang” your game while consuming all of the CPU time for itself.**

The thread scheduler itself synchronizes with the VDP interrupt flag.

It means that -in best case- a thread can be executed every 1/60th (NTSC) or 1/50th (PAL) of a second.

For some tasks (e.g. sprite coincidence detection) this may be too slow.

That is why the thread scheduler offers the possibility to call a “kernel thread” and a “user hook” each time it reads the VDP status register.

The kernel thread is responsible for controlling the built-in music player and virtual keyboard.

The speech player is controlled by code embedded in the thread scheduler itself. This is to obtain the best possible performance. Note that some of the bit-flags in the CONFIG register control the behaviour of the speech player, kernel thread, etc.

The below schema shows the thread scheduler workflow:

(A) +--------------------------+ (B)

................................> | Read VDP status register | <.......................

. +-------------.------------+ .

. | .

. VDP int flag set? .

. | .

. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ .

. Y | | N .

. | | .

. +-------------.------------+ Speech player on? .

. | Do internal housekeeping | | . F

. | | +---------.---------+ . A

. | \* Unblock kernel thread | | Run speech player | . S

. | \* Unblock user hook | +---------.---------+ (\*) . T

. | \* Update tick counter | | .

. +-------------.------------+ Kernel thread unblocked/enabled? .

. | | .

. +---------.--------+ +--------.--------+ .

. | Update slot data | <............ | Set block flag | .

. | counter, etc. | . | and run | .

. +---------.--------+ . | kernel thread | .

. | . +--------.--------+ (\*) .

. Internal counter = target count in slot? . | .

. | . User hook unblocked/enabled? .

. +-----.-----+ . | .

. | Run slot | . +-------.-------+ .

. +-----.-----+ (\*) . | Run user hook | .

. | . +-------.-------| (\*) .

. Last slot done? . . .

. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_ . ........>.........

. Y | | N .

. | | .

. +-----.------+ +------.------+ .

. | Reset tick | | Prepare for | .

. | counter | | next slot | .

. +-----.------+ +------.------+ .

. . . .

....<...... ......>.....

(A) = Executed once per frame (1/60th for NTSC, 1/50th for PAL)

(B) = User hook repeats until blocked from within user hook code.

Kernel thread (sound player, keyboard scan) runs once per frame.

(\*) = Skipped depending on result of previous check

### The timer table

The scheduler requires a work table in CPU memory for keeping track of the threads, when to fire, etc.

It’s the programmers’ responsibility to make sure there is enough free CPU memory for holding this table.

**Make sure the table is properly initialized with >00 bytes, otherwise the thread scheduler may interpret memory as an allocated slot, execute this garbled slot and lock the computer.**

You have to store the address of the table at memory location @WTITAB, as seen in the next example:

...

MOV @MYTAB,@WTITAB ; Setup address of timer table

BL @MKSLOT

DATA >0002,MVBOX,EOL ; Create new timer slot

B @TMGR ; Start thread scheduler

MYTAB DATA >8350 ; Timer table address

For each running thread a timer slot must be allocated.   
A timer slot consists of 4 bytes and the initial setup is normally done by using the MKSLOT subroutine.

|  |  |  |
| --- | --- | --- |
| **BYTE 0-1** | **BYTE 2** | **BYTE 3** |
| Thread address | Interval | Internal tick counter |

**Thread** This is the address of the subroutine that will be called by the thread scheduler when the slot is fired.  
**An empty slot must contain >0000 in BYTE 0-1**.

**Interval**  Determines at what interval the slot should be fired. This interval must be specified in ticks per second.  
**On a NTSC console we have 60 ticks per second.   
On a PAL console we have 50 ticks per second.**

**Internal counter** Is an internal counter used by the thread scheduler to keep track about when the slot should be fired.

### Highest slot in use

The thread scheduler must know how many slots it needs to handle.  
This is controlled by the most signifcant byte of register R10.

Note that register R10 is set to 0 when the library is initialized. It means that by default only slot 0 gets executed.

In the below example, the highest slot in use is set to 2.

START LI R10,>0200 ; Set highest slot to 2

### The kernel thread

Both the built-in music player and the scanning of the virtual keyboard is handled transparantly by a background thread called the “kernel thread”. You do not need to allocate a timer slot for it.

The kernel thread automatically runs once per frame. This is controlled by the “thread block flag”. That’s bit 8 in the CONFIG register. This block flag is set by the kernel thread upon exit and is reset by the thread scheduler once the next frame is reached.

The kernel thread feature can be completely turned off by resetting the “thread enabled” flag, that’s bit 9 in the CONFIG register. However in that case there will be no automatic sound player and keyboard scanning.

Use the below code for turning off the kernel thread:

START SZC @WBIT9,CONFIG ; Turn off kernel thread

B @TMGR

### The user hook

Calling a subroutine once per frame may be insufficient for certain tasks. That’s especially the case if you want to reliably scan some of the VDP status register flags (e.g. coincidence detection, 5th sprite in a row, etc).

This is where the user hook comes to the rescue: Once loaded it will execute (using BRANCH!) each time the VDP status register is read.

The user hook is turned off by default, this is controlled by the “user-hook enabled” flag (bit 7 in the CONFIG register.).

The easiest way to setup a user hook, is by using the MKHOOK subroutine. **Note that spectra2 only supports 1 user hook.**

You can delay the next execution until the next frame is reached, by setting the “block user-hook” flag in your hook code (bit 6 in the CONFIG register).

**To exit the user hook code and return to the thread scheduler, you have to issue a** “**B @HOOKOK**”.

In the below example, a user hook is defined for checking the coincidence status flag. The thread scheduler automatically copies the value of the VDP status register into the high byte of R13.

BL @MKHOOK ; Prepare user hook

DATA COINC

GAME2 B @TMGR

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* User hook - Check for coincidence

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

COINC COC @WBIT2,R13 ; Coincidence bit set ?

JNE COINCZ ; No, exit

…

COINCZ B @HOOKOK ; Back to thread scheduler

### Support routines

Following subroutines are available for dealing with threads:

* **TMGR**The TMGR subroutine is the entry point into the thread scheduler. It should be started with a “**B @TMGR**” after initialisation in the main program has completed.

Make sure you checked the below before initiating TMGR, it will save you a lot of time searching for program crashes:

* + Memory address WTITAB (2 bytes) set with address of your timer table.
  + Timer table initialized with >00 bytes.
  + Memory address BTIHI (1 byte!) set with highest timer slot in use.
* **MKSLOT**The MKSLOT subroutine is used for allocating new timer slots. It allows you to allocate non-sequential slots, e.g. allocate slots 0,3,4,7 (without touching slots 1,2,5,6).If you have many slots to allocate at once, then you could copy a preset slot table from ROM to RAM without using the MKSLOT subroutine.  
  Please refer to page 116 for further details.
* **CLSLOT**Use the CLSLOT subroutine to remove a single running slot.  
  Please refer to page 118 for further details.
* **MKHOOK**The MKHOOK subroutine is used for allocating a user hook.   
  Please refer to page 120 for further details.
* **KERNEL**The KERNEL subroutine runs as a thread and is responsible for running the sound player and reading the virtual keyboard. You should normally not call this subroutine from your program, it’s automatically called by spectra2.

Please refer to page 119 for further details.

### Support equates

Following equates are available for dealing with threads:

\*--------------------------------------------------------------

\* Equates for scratchpad memory locations

\*--------------------------------------------------------------

WTITAB EQU >832C ; 02 - Timers: Address of timer table

WTIUSR EQU >832E ; 02 - Timers: Address of user hook

WTITMP EQU >8330 ; 02 - Timers: Internal use

\*--------------------------------------------------------------

\* Equates for CONFIG register

\*--------------------------------------------------------------

ENUSR EQU >0100 ; bit 7=1 (Enable user hook)

ENKNL EQU >0040 ; bit 9=1 (Enable kernel thread)

### Register usage

R10 is exclusively used by the thread scheduler:

* The high byte of R10 keeps track of the highest slot used in the thread scheduler timer table.
* The low byte of R10 is the thread scheduler tick counter and is updated every 1/50th (VDP) or 1/60th (NTSC) of a second.

### Exiting a thread

There are many ways how one can exit a thread. Let’s look at some of the possibilities:

**B \*R11**

Use “B \*R11” (2 bytes of machine code) to exit a thread and return to the thread scheduler if you didn’t use any BL (Branch-and-link) instruction in your thread code.

THREAD1 BLABLA ; Some statements

…

B \*R11 ; Exit thread (2 bytes)

**B @SLOTOK**

Use “B @SLOTOK” (6 bytes of machine code) to exit a thread and return to the thread scheduler if you use a BL to call a subroutine from your thread.

THREAD1 BLABLA ; Some statements

BL @MYSUB1 ; Call some routine. R11 is overwritten

B @SLOTOK ; Exit thread (6 bytes)

**Save return address in other register**

As an alternative you can save a copy of R11 and work with that. Remember that R0-R3 are not used by spectra2 so they are good candidates.

THREAD1 MOV R11,R0 ; Save copy of R11 (2 bytes)

BL @MYSUB1 ; Call some routine. R11 is overwritten

B \*R0 ; Exit thread (2 bytes)

**Save return address on stack**

If you decide to set-up a return stack, you can do so by using the STACK register (R9). The STACK register is initialised to >8400 upon library initialisation. Use the POPRT subroutine to pop the return address from the stack and return.

THREAD1 DECT STACK ; Set stack pointer (R9)

MOV R11,\*STACK ; Save return address on stack

BL @MYSUB1 ; Call some routine. R11 is overwritten

B @POPRT ; Pop R11 from stack and return to caller

**Example**

In the next example, we start a thread for showing the blinking message ‘HELLO WORLD!’. The thread interval is set to 15 ticks, which means that the text will effectively blink once every ½ second.



At the same time the speech player will be playing back a recorded speech sample and the kernel thread will scan the keyboard and handle FNCTN-QUIT.

This example is included as file **example2.a99** in the spectra2 samples directory.

AORG >6000

\*--------------------------------------------------------------

\* Cartridge header

\*--------------------------------------------------------------

GRMHDR BYTE >AA,1,1,0,0,0

DATA PROG

BYTE 0,0,0,0,0,0,0,0

PROG DATA 0

DATA RUNLIB

HW BYTE 12

TEXT 'HELLO WORLD!'

\*--------------------------------------------------------------

\* Include required files & startup options

\*--------------------------------------------------------------

COPY "D:\Projekte\spectra2\tms9900\runlib.a99"

SPVMOD EQU GRAPH1 ; Video mode. See VIDTAB for details.

SPFONT EQU FNOPT7 ; Font to load. See LDFNT for details.

SPFCLR EQU >F0 ; Foreground/Background color for font.

SPFBCK EQU >01 ; Screen background color.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Main

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MAIN BL @FILV

DATA >0380,>F0,16 ; Set color table

LI R0,>8370

MOV R0,@WTITAB ; Our timer table

BL @MKSLOT

DATA >000F,BLINK,EOL ; Run thread every 15 ticks

BL @SPPREP

DATA ROCK,SPOPT1 ; Speech player on / Speak external

MOVB @BD1,@>8369 ; Set toggle

B @TMGR ; Run scheduler

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Thread

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BLINK NEG @>8368 ; Switch toggle

JLT BLIN2

BLIN1 BL @PUTAT

DATA >0A0A,HW ; Show "Hello World!" message

JMP BLIN3

BLIN2 BL @HCHAR

BYTE >0A,>0A,32,12 ; white space x

DATA EOL

BLIN3 B @SLOTOK ; Exit to Thread Scheduler

ROCK BYTE TALKON ; Speech data

BYTE >00,>E0,>80,>E2,>3B,>13,>50,>DC,>64,>00,>AA,>E9,>3C,>69

...

BYTE TALKOF

END

# Virtual keyboard

### VIRTKB subroutine

The spectra2 runtime supports a virtual TI-99/4A game keyboard controlled by the “kernel” thread. Each time the thread runs, it calls the subroutine VIRTKB for polling the keyboard/joystick status. It then maps the pressed keys as bit flags in the memory location @WVRTKB.

Benefit of the virtual keyboard is that you do not need to check both keyboard and joysticks. If you for example press ‘S’ on the keyboard, it reacts the same as if you pull joystick 1 to the left. They will both set the bit for the virtual key ‘K1LF’ in @WVRTKB to 1.  
  
Note that the virtual keyboard does not support all keys, but it does handle enough keys for supporting an arcade game. The subroutine also checks for FNCTN-QUIT and exits to the TI-99/4A title screen when pressed.

It also handles multiple keys. If you for example pull joystick 1 diagonally up/left, then it will set both virtual keys ‘K1UP’ and ‘K1LF’. Be aware, that the VIRTKB subroutine always scans the full keyboard. It is not possible to only scan left/right half of the keyboard.

### The ‘ANY’ key

As soon as the VIRTKB routine detects that a key is pressed (or joystick pulled), it will set bit 11 (ANYKEY) in the CONFIG register.

Use the below code to check if any key was pressed:

CHECK COC @ANYKEY,CONFIG ; ANY key pressed ?

JEQ MYLABL ; YES

B \*R11 ; NO, exit

MYLABL … ; Process key

### Support routines

Following subroutines in the spectra2 library are available when dealing with the virtual keyboard:

* **VIRTKB**The VIRTKB subroutine handles the scanning of the keyboard and maps it the corresponding bit flags in @WVRTKB. Normally you should not call VIRTKB directly, because this is all handle in the background by the “KERNEL” thread.

Check the “Thread scheduler” section at page 43 for further details on the kernel thread.

### Support equates

Below are the equates for checking virtual keys:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Equates for virtual keyboard (@WVRTKB)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ; bit 0: ALPHA LOCK down 0=no 1=yes

\* ; bit 1: ENTER 0=no 1=yes

\* ; bit 2: REDO 0=no 1=yes

\* ; bit 3: BACK 0=no 1=yes

\* ; bit 4: Pause 0=no 1=yes

\* ; bit 5: \*free\* 0=no 1=yes

\* ; bit 6: P1 Left 0=no 1=yes

\* ; bit 7: P1 Right 0=no 1=yes

\* ; bit 8: P1 Up 0=no 1=yes

\* ; bit 9: P1 Down 0=no 1=yes

\* ; bit 10: P1 Space / fire / Q 0=no 1=yes

\* ; bit 11: P2 Left 0=no 1=yes

\* ; bit 12: P2 Right 0=no 1=yes

\* ; bit 13: P2 Up 0=no 1=yes

\* ; bit 14: P2 Down 0=no 1=yes

\* ; bit 15: P2 Space / fire / Q 0=no 1=yes

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

KALPHA EQU >8000 ; Virtual key alpha lock

KENTER EQU >4000 ; Virtual key enter

KREDO EQU >2000 ; Virtual key REDO

KBACK EQU >1000 ; Virtual key BACK

KPAUSE EQU >0800 ; Virtual key pause

KFREE EQU >0400 ; \*\*\*NOT USED YET\*\*\*

\*--------------------------------------------------------------

\* Keyboard Player 1

\*--------------------------------------------------------------

K1UPLF EQU >0280 ; Virtual key up + left

K1UPRG EQU >0180 ; Virtual key up + right

K1DNLF EQU >0240 ; Virtual key down + left

K1DNRG EQU >0140 ; Virtual key down + right

K1LF EQU >0200 ; Virtual key left

K1RG EQU >0100 ; Virtual key right

K1UP EQU >0080 ; Virtual key up

K1DN EQU >0040 ; Virtual key down

K1FIRE EQU >0020 ; Virtual key fire

\*--------------------------------------------------------------

\* Keyboard Player 2

\*--------------------------------------------------------------

K2UPLF EQU >0014 ; Virtual key up + left

K2UPRG EQU >000C ; Virtual key up + right

K2DNLF EQU >0012 ; Virtual key down + left

K2DNRG EQU >000A ; Virtual key down + right

K2LF EQU >0010 ; Virtual key left

K2RG EQU >0008 ; Virtual key right

K2UP EQU >0004 ; Virtual key up

K2DN EQU >0002 ; Virtual key down

K2FIRE EQU >0001 ; Virtual key fire

### Example

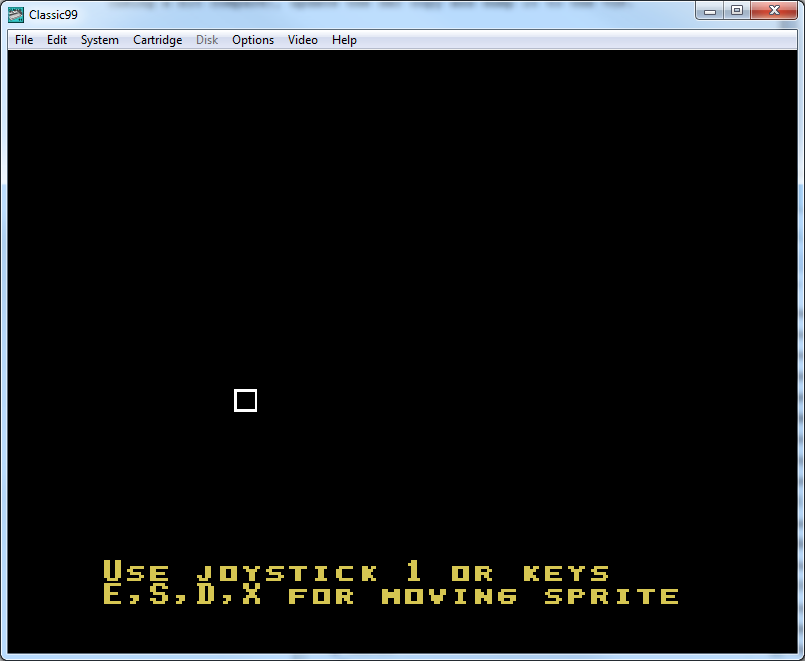
In the next example, we will be moving a sprite using the keyboard or joystick 1.

The main program prepares a copy of the Sprite Attribute Table (SAT) in RAM, displays an information message on screen and allocates a new thread (“MVBOX”) with a 1-tick repeat interval.

After the thread scheduler (TMGR) has taken over, it automatically starts the kernel thread and also execute the MVBOX thread every 1/50th or 1/60th of a second.

The keyboard scanning is automatically done by the kernel thread, the MVBOX subroutine only needs to check what virtual keys got pressed (using a bit compare), update the SAT copy and dump it to the VDP.

This example is included as file **example3.a99** in the spectra2 samples directory.



AORG >6000

\*--------------------------------------------------------------

\* Cartridge header

\*--------------------------------------------------------------

GRMHDR BYTE >AA,1,1,0,0,0

DATA PROG

BYTE 0,0,0,0,0,0,0,0

PROG DATA 0

DATA RUNLIB

HW BYTE 15

TEXT 'MOVE THE SPRITE'

EVEN

\*--------------------------------------------------------------

\* Include required files

\*--------------------------------------------------------------

COPY "D:\Projekte\spectra2\tms9900\runlib.a99"

\*--------------------------------------------------------------

\* SPECTRA2 startup options

\*--------------------------------------------------------------

SPVMOD EQU GRAPH1 ; Video mode. See VIDTAB for details.

SPFONT EQU FNOPT7 ; Font to load. See LDFNT for details.

SPFCLR EQU >A0 ; Foreground/Background color for font.

SPFBCK EQU >01 ; Screen background color.

\*--------------------------------------------------------------

\* Our constans and variables in scratchpad memory

\*--------------------------------------------------------------

RAMSAT EQU >8340 ; Copy of mini-SAT in RAM memory (6 bytes)

RAMTAB EQU >8346 ; Timer table (4 bytes)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Main

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MAIN BL @CPYM2M

DATA SPRITE,RAMSAT,6 ; Copy 6 bytes from ROM into scratchpad RAM

BL @CPYM2V

DATA >1000,PAT1,8 ; Dump sprite pattern

BL @PUTBOX

DATA >1503,>1A02,INFO,EOL ; Show text in box

MOV @MYTAB,@WTITAB ; Setup address of timer table

BL @MKSLOT

DATA >0002,MVBOX,EOL ; Create new timer slot

B @TMGR ; Handle FCTN-QUIT key, timers, etc.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* THREAD Move sprite: This routine is called from TMGR

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MVBOX MOV R11,R0 ; Save R11 in R0

COC @WBIT11,CONFIG ; ANY key pressed ?

JNE MVBOX5 ; No, so exit

MOV @WVRTKB,R1 ; Get keyboard flags

MVBOX1 COC @KEY1,R1 ; Left ?

JNE MVBOX2

SB @BD2,@RAMSAT+1 ; X=X-2

MVBOX2 COC @KEY2,R1 ; Right ?

JNE MVBOX3

AB @BD2,@RAMSAT+1 ; X=X+2

MVBOX3 COC @KEY3,R1 ; Up ?

JNE MVBOX4

SB @BD2,@RAMSAT ; Y=Y-2

MVBOX4 COC @KEY4,R1 ; Down ?

JNE MVBOX5

AB @BD2,@RAMSAT ; Y=Y+2

MVBOX5 BL @CPYM2V ; Dump copy of SAT to VDP SAT

DATA >0300,RAMSAT,6 ; ... R11 is overwritten

B \*R0 ; ... so return using copy in R0

KEY1 DATA K1LF ; Left

KEY2 DATA K1RG ; Right

KEY3 DATA K1UP ; Up

KEY4 DATA K1DN ; Down

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Our constants

\*\*\*\*\*\*\*\*@\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MYTAB DATA RAMTAB ; Location of timer table in scratchpad memory

SPRITE DATA >2020,>000F ; Row >20, col >20, pattern >00, color white

DATA >0D00 ; No more sprites

PAT1 DATA >FF81,>8181,>8181,>81FF

INFO BYTE 52

TEXT 'Use joystick 1 or keys E,S,D,X for moving sprite'

END

# Memory / Copy subroutines

|  |  |
| --- | --- |
| **MEMORY/COPY** |  |

## CPYM2M / XPYM2M

Copy ROM/RAM to RAM

|  |  |
| --- | --- |
| CPYM2M – Parameter in DATA statement | |
| **Call format** | MYTEST BL @CPYM2M  DATA P0,P1,P2 |
| **Input** | P0 = ROM/RAM source address  P1 = RAM destination address  P2 = Number of bytes to copy |
| **Example** | /samples/example3.a99 |

|  |  |
| --- | --- |
| XPYM2M – Parameter in register | |
| **Call format** | MYTEST BL @XPYM2M |
| **Input** | TMP0 = ROM/RAM source address  TMP1 = RAM destination address  TMP2 = Number of bytes to fill |
| **Example** | /samples/????.a99 |

**Description:**

Copy a CPU memory range. Source can be either in RAM or ROM. Destination must be RAM. Note that this subroutine uses some machinecode in scratch-pad memory for obtaining the best possible performance.

**Example:**

Copy 8K of cartridge ROM (>6000 - >7FFF) to high memory (>A000).  
  
MAIN BL @CPYM2M  
 DATA >6000,>A000,8192

LI TMP0,>6000

LI TMP1,>A000

LI TMP2,8192

BL @XPYM2M

|  |  |
| --- | --- |
| **MEMORY/COPY** |  |

## CPYM2V / XPYM2V

Copy ROM/RAM to VDP VRAM

|  |  |
| --- | --- |
| CPYM2V – Parameter in DATA statement | |
| **Call format** | MYTEST BL @CPYM2V  DATA P0,P1,P2 |
| **Input** | P0 = VDP VRAM destination address  P1 = ROM/RAM source address  P2 = Number of bytes to copy |
| **Example** | /samples/example3.a99 |

|  |  |
| --- | --- |
| XPYM2V – Parameter in register | |
| **Call format** | MYTEST BL @XPYM2V |
| **Input** | TMP0 = VDP VRAM destination address  TMP1 = ROM/RAM source address  TMP2 = Number of bytes to fill |
| **Example** | /samples/example6.a99 |

**Description:**

Copy a CPU memory range to VDP VRAM. Source can be either in RAM or ROM. Destination must be VRAM. Note that this subroutine uses some machinecode in scratch-pad memory for obtaining the best possible performance.

Has basically the same functionality as the Editor/Assembler VMBW utility.

**Example:**

Copy a color table from ROM to VDP RAM (>0380).  
  
MAIN BL @CPYM2V  
 DATA >0380,COLTAB,16

JMP $

COLTAB BYTE >03,>03,>03,>03,>05,>05,>07,>0F

BYTE >0F,>0F,>0F,>03,>03,>04,>04,>04

|  |  |
| --- | --- |
| **MEMORY/COPY** |  |

## CPYV2M / XPYV2M

Copy VDP VRAM to RAM

|  |  |
| --- | --- |
| CPYV2M – Parameter in DATA statement | |
| **Call format** | MYTEST BL @CPYV2M  DATA P0,P1,P2 |
| **Input** | P0 = VDP VRAM source address  P1 = RAM destination address  P2 = Number of bytes to copy |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| XPYV2M – Parameter in register | |
| **Call format** | MYTEST BL @XPYV2M |
| **Input** | TMP0 = VDP VRAM source address  TMP1 = RAM destination address  TMP2 = Number of bytes to copy |
| **Example** | /samples/????.a99 |

**Description:**

Copy a memory range from VDP VRAM to RAM. Note that this subroutine uses some machine code in scratch-pad memory for obtaining the best possible performance.

Has basically the same functionality as the Editor/Assembler VMBR utility.

**Example:**

Copy 16 bytes from VDP VRAM >0380 to scratchpad RAM >8370.  
  
MAIN BL @CPYV2M  
 DATA >0380,>8370,16

JMP $

|  |  |
| --- | --- |
| **MEMORY/COPY** |  |

## CPYG2M / XPYG2M

Copy GROM to RAM

|  |  |
| --- | --- |
| CPYG2M – Parameter in DATA statement | |
| **Call format** | MYTEST BL @CPYG2M  DATA P0,P1,P2 |
| **Input** | P0 = GROM source address  P1 = RAM destination address  P2 = Number of bytes to copy |
| **Example** | runlib.a99 |

|  |  |
| --- | --- |
| XPYG2M – Parameter in register | |
| **Call format** | MYTEST BL @XPYG2M |
| **Input** | TMP0 = GROM source address  TMP1 = RAM destination address  TMP2 = Number of bytes to copy |
| **Example** | /samples/????.a99 |

**Description:**

Copy a memory range from GROM to RAM. Note that this subroutine uses some machine code in scratch-pad memory for obtaining the best possible performance.

|  |  |
| --- | --- |
| **MEMORY/COPY** |  |

## CPYG2V / XPYG2V

Copy GROM to VDP VRAM

|  |  |
| --- | --- |
| CPYG2V – Parameter in DATA statement | |
| **Call format** | MYTEST BL @CPYG2V  DATA P0,P1,P2 |
| **Input** | P0 = GROM source address  P1 = VRAM destination address  P2 = Number of bytes to copy |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| XPYG2M – Parameter in register | |
| **Call format** | MYTEST BL @XPYG2V |
| **Input** | TMP0 = GROM source address  TMP1 = VRAM destination address  TMP2 = Number of bytes to copy |
| **Example** | /samples/????.a99 |

**Description:**

Copy a memory range from GROM to VDP VRAM. Note that this subroutine uses some machine code in scratch-pad memory for obtaining the best possible performance.

|  |  |
| --- | --- |
| **MEMORY/COPY** |  |

## FILM / XFILM

Fill RAM with byte

|  |  |
| --- | --- |
| FILM – Parameter in DATA statement | |
| **Call format** | MYTEST BL @FILM  DATA P0,P1,P2 |
| **Input** | P0 = RAM start address  P1 = Byte to fill  P2 = Number of bytes to fill |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| XFILM – Parameter in register | |
| **Call format** | MYTEST BL @XFILM |
| **Input** | TMP0 = RAM start address  TMP1 = Byte to fill  TMP2 = Number of bytes to fill |
| **Example** | /samples/????.a99 |

**Description:**

This routine is used for filling the specified CPU RAM range with a byte value. Note that this subroutine uses some machine code in scratch-pad memory for obtaining the best possible performance.

**Example:**

Fill high-memory range >A000 - >B000 with byte >FF.  
  
MAIN BL @FILM  
 DATA >6000,>FF,4096

|  |  |
| --- | --- |
| **MEMORY/COPY** |  |

## FILV / XFILV

Fill VDP VRAM with byte

|  |  |
| --- | --- |
| FILV – Parameter in DATA statement | |
| **Call format** | MYTEST BL @FILV  DATA P0,P1,P2 |
| **Input** | P0 = VDP VRAM start address  P1 = Byte to fill  P2 = Number of bytes to fill |
| **Example** | /samples/example2.a99 |

|  |  |
| --- | --- |
| XFILV – Parameter in register | |
| **Call format** | MYTEST BL @XFILV |
| **Input** | TMP0 = VDP VRAM start address  TMP1 = Byte to fill  TMP2 = Number of bytes to fill |
| **Example** | /samples/????.a99 |

**Description:**

This routine is used for filling the specified VDP VRAM memory range with a byte value. Note that this subroutine uses some machine code in scratch-pad memory for obtaining the best possible performance.

**Example:**

Clear the 16K of VDP VRAM memory (>0000 - >3FFF).  
  
MAIN BL @FILM  
 DATA >0000,>00,16384

# VDP low-level subroutines

|  |  |
| --- | --- |
| **VDP low-level** |  |

## VDWA

Setup VDP write address

|  |  |
| --- | --- |
| VDWA – Parameter in register | |
| **Call format** | MYTEST BL @VDWA |
| **Input** | TMP0 = VDP VRAM destination address |
| **Example** | Runlib.a99 |

**Description:**

Setup the VDP destination address for writing. Specify the VDP destination address in register TMP0. Useful if you need to insert some inline VSBW/VMBW code in your subroutine.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## VDRA

Setup VDP read address

|  |  |
| --- | --- |
| VDWA – Parameter in register | |
| **Call format** | MYTEST BL @VDRA |
| **Input** | TMP0 = VDP VRAM destination address |
| **Example** | runlib.a99 |

**Description:**

Setup the VDP destination address for reading. Specify the VDP destination address in register TMP0. Useful if you need to insert some inline VSBR/VMBR code in your subroutine.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## VPUTB / XVPUTB

Write a single byte to VDP VRAM

|  |  |
| --- | --- |
| VPUTB – Parameter in DATA statement | |
| **Call format** | MYTEST BL @VPUTB  DATA P0,P1 |
| **Input** | P0 = VDP VRAM destination address  P1 = Byte to write |
| **Example** | /samples/example4.a99 |

|  |  |
| --- | --- |
| XVPUTB – Parameter in register | |
| **Call format** | MYTEST BL @XVPUTB |
| **Input** | TMP0 = VDP VRAM destination address  TMP1 = Byte to write |
| **Example** | /samples/example5.a99 |

|  |  |
| --- | --- |
| **Dependencies** | VDWA |

**Description:**

Write a single byte to VDP VRAM. Has the same functionality as the Editor/Assembler VSBW utility.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## VGETB / XVGETB

Read a single byte from VDP VRAM

|  |  |
| --- | --- |
| VGETB – Parameter in DATA statement | |
| **Call format** | MYTEST BL @VGETB  DATA P0 |
| **Input** | P0 = VDP VRAM source address |
| **Output** | TMP0 = Byte read (in LO-byte) |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| XVGETB – Parameter in register | |
| **Call format** | MYTEST BL @XVGETB |
| **Input** | TMP0 = VDP VRAM source address |
| **Output** | TMP0 = Byte read (in LO-byte) |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | VDRA |

**Description:**

Read a single byte from VDP VRAM. Has the same functionality as the Editor/Assembler VSBR utility. The byte read is returned in the low-byte of register TMP0

|  |  |
| --- | --- |
| **VDP low-level** |  |

## VIDTAB / XIDTAB

Dump video mode table to VDP registers

|  |  |
| --- | --- |
| VIDTAB – Parameter in DATA statement | |
| **Call format** | MYTEST BL @VIDTAB  DATA P0 |
| **Input** | P0 = ROM/RAM address of video mode table |
| **Example** | runlib.a99 |

|  |  |
| --- | --- |
| XIDTAB – Parameter in register | |
| **Call format** | MYTEST BL @XIDTAB |
| **Input** | TMP0 = ROM/RAM address of video mode table |
| **Example** | /samples/????.a99 |

**Description:**

Instead of individually loading each of the VDP write-only registers, you can use this subroutine to load all 7 VDP write-only registers at once. For doing so, you need a table holding the required byte value for each of the registers. There are some default video mode tables bundled with the runtime library (e.g. GRAPH1, TXTMOD).

Note that the subroutine also calculates the base address of the pattern name table by checking the value of VDP register #2. It then stores the calculated address in scratchpad memory location WBASE.

Please refer to the TMS9918 VDP programmer’s guide for details on the 7 VDP registers.

|  |  |
| --- | --- |
|  |  |

See section “scratchpad memory setup” on page 23 (item c) for further details on the PNT base address.

Here’s a sample video mode table (included in the runtime library):

TXTMOD BYTE >00,>F2,>00,>0E,>01,>06,>80,SPFCLR

\*--------------------------------------------------------------

\* Textmode (40 columns)

\*--------------------------------------------------------------

\* ; VDP#0 Control bits

\* ; bit 6=0: M3 | Graphics 1 mode

\* ; bit 7=0: Disable external VDP input

\* ; VDP#1 Control bits

\* ; bit 0=1: 16K selection

\* ; bit 1=1: Enable display

\* ; bit 2=1: Enable VDP interrupt

\* ; bit 3=1: M1 \ TEXT MODE

\* ; bit 4=0: M2 /

\* ; bit 5=0: reserved

\* ; bit 6=1: 16x16 sprites

\* ; bit 7=0: Sprite magnification (1x)

\* ; VDP#2 PNT (Pattern name table) at >0000 (>00 \* >400)

\* ; VDP#3 PCT (Pattern color table) at >0380 (>0E \* >040)

\* ; VDP#4 PDT (Pattern descriptor table) at >0800 (>01 \* >800)

\* ; VDP#5 SAT (sprite attribute list) at >0300 (>06 \* >080)

\* ; VDP#6 SPT (Sprite pattern table) at >0400 (>80 \* >008)

\* ; VDP#7 Set foreground/background color

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Example:**

Switch the TMS9918 VDP into 40 columns mode (text-mode)

MAIN BL @VIDTAB  
 DATA TXTMOD

JMP $

|  |  |
| --- | --- |
| **VDP low-level** |  |

## PUTVR / PUTVRX

Load single VDP register with byte

|  |  |
| --- | --- |
| PUTVR – Parameter in DATA statement | |
| **Call format** | MYTEST BL @PUTVR  DATA P0 |
| **Input** | P0 = MSB contains the VDP target register  LSB contains byte to load |
| **Example** | /samples/game/hc\_source2.a99 |

|  |  |
| --- | --- |
| PUTVRX – Parameter in register | |
| **Call format** | MYTEST BL @PUTVRX |
| **Input** | TMP0 = MSB contains the VDP target register  LSB contains byte to load |
| **Example** | /samples/????.a99 |

**Description:**

Load single VDP write-only register with specified byte. Same functionality as the Editor/Assembler VWTR utility.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## PUTV01

Load VDP registers #0 and #1 from R14

|  |  |
| --- | --- |
| PUTVRX – Parameter in register | |
| **Call format** | MYTEST BL @PUTV01 |
| **Input** | R14 = MSB contains byte for VDP register #0  LSB contains byte for VDP register #1 |
| **Example** | runlib.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTVRX |

**Description:**

The spectra2 library uses CPU register R14 for holding a copy of VDP write-only registers #0 and #1. Basically one first sets/resets the corresponding bit masks in R14 and then uses the PUTV01 subroutine for loading the byte values in VDP register #0 and #1.

The high byte of R14 contains a copy of VDP write-only register #0.

The low byte of R14 contains a copy of VDP write-only register #1.

Various features of the VDP are controlled by bit flags in VDP register #0 and #1, e.g. current video mode, sprite magnification, interupt enabling, etc.

Please refer to the TMS9918 VDP programmer’s guide for further details.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## SCROFF

Turn screen off

|  |  |
| --- | --- |
| SCROFF – No parameter | |
| **Call format** | MYTEST BL @SCROFF |
| **Example** | /samples/game/hc\_source2.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTV01, PUTVRX |

**Description:**

This subroutine sets bit 1 in VDP write-only register #1 to 0.

As a result the VDP will turn off the screen display and will open a permanent window for CPU access.

You normally use this command before drawing a new screen. Once it is fully drawn, you can then use the SCRON subroutine for turning on the screen again.

Please refer to the TMS9918 VDP programmer’s guide for further details.

Note that the corresponding bit in the VDP shadow register (R14) is also set to 0. See page 31 for details on R14.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## SCRON

Turn screen on

|  |  |
| --- | --- |
| SCRON – No parameter | |
| **Call format** | MYTEST BL @SCRON |
| **Example** | /samples/game/hc\_source2.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTV01, PUTVRX |

**Description:**

This subroutine sets bit 1 in VDP write-only register #1 to 1.

As a result the VDP will turn on the screen display again.

You normally call the SCRON subroutine after issuing a SCROFF and doing some screen manipulation.

Please refer to the TMS9918 VDP programmer’s guide for further details.

Note that the corresponding bit in the VDP shadow register (R14) is also set to 1. See page 31 for details on R14.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## INTOFF

Disable VDP interrupt

|  |  |
| --- | --- |
| INTOFF – No parameter | |
| **Call format** | MYTEST BL @INTOFF |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTV01, PUTVRX |

**Description:**

This subroutine sets bit 2 in VDP write-only register #1 to 0.

As a result the VDP will NOT trigger the CPU interrupt line at the end of the active screen area.

Note that the spectra2 thread scheduler (TMGR) continuously checks the VDP interrupt flag. **The scheduler will not work if you use INTOFF to disable VDP interrupts**.

Please refer to the TMS9918 VDP programmer’s guide for further details.

Note that the corresponding bit in the VDP shadow register (R14) is also set to 0. See page 31 for details on R14.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## INTON

Enable VDP interrupt

|  |  |
| --- | --- |
| INTON – No parameter | |
| **Call format** | MYTEST BL @INTON |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTV01, PUTVRX |

**Description:**

This subroutine sets bit 2 in VDP write-only register #1 to 1.

As a result the VDP will trigger the CPU interrupt line at the end of the active screen display area, just before vertical retrace starts.

Note, that you can still mask the CPU interrupt by using the “LIMI 0” instruction.

This is the default setting when spectra2 is initialized.

Please refer to the TMS9918 VDP programmer’s guide for further details.

Note that the corresponding bit in the VDP shadow register (R14) is also set to 1. See page 31 for details on R14.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## SMAG1X

Set sprite magnification 1X

|  |  |
| --- | --- |
| SMAG1X – No parameter | |
| **Call format** | MYTEST BL @SMAG1X |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTV01, PUTVRX |

**Description:**

This subroutine sets bit 7 in VDP write-only register #1 to 0.

As a result the VDP will remove the sprite magnification,

Please refer to the TMS9918 VDP programmer’s guide for further details.

Note that the corresponding bit in the VDP shadow register (R14) is also set to 0. See page 31 for details on R14.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## SMAG2X

Set sprite magnification 2X

|  |  |
| --- | --- |
| SMAG2X – No parameter | |
| **Call format** | MYTEST BL @SMAG2X |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTV01, PUTVRX |

**Description:**

This subroutine sets bit 7 in VDP write-only register #1 to 1.

As a result the VDP will install sprite magnification. This means that 8x8 sprites become 16x16 and 16x16 sprites become 32x32.

Please refer to the TMS9918 VDP programmer’s guide for further details.

Note that the corresponding bit in the VDP shadow register (R14) is also set to 1. See page 31 for details on R14.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## S8X8

Set sprite size to 8x8 pixels

|  |  |
| --- | --- |
| S8X8 – No parameter | |
| **Call format** | MYTEST BL @S8X8 |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTV01, PUTVRX |

**Description:**

This subroutine sets bit 6 in VDP write-only register #1 to 0.

As a result the VDP will set the sprite size to 8x8 pixels. It means that you need 8 bytes to define a sprite pattern.

Please refer to the TMS9918 VDP programmer’s guide for further details.

Note that the corresponding bit in the VDP shadow register (R14) is also set to 0. See page 31 for details on R14.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## S16X16

Set sprite size to 16x16 pixels

|  |  |
| --- | --- |
| S16X16 – No parameter | |
| **Call format** | MYTEST BL @S16X16 |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTV01, PUTVRX |

**Description:**

This subroutine sets bit 6 in VDP write-only register #1 to 1.

As a result the VDP will set the sprite size to 16x16 pixels. It means that you need 32 bytes to define a sprite pattern.

Please refer to the TMS9918 VDP programmer’s guide for further details.

Note that the corresponding bit in the VDP shadow register (R14) is also set to 1. See page 31 for details on R14.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## GTCLMN

Get number of columns per row

|  |  |
| --- | --- |
| GTCLMN – No parameter | |
| **Call format** | MYTEST BL @S16X16 |
| **Output** | TMP0 = Number of columns per row (32, 40, 64) |
| **Example** | runlib.a99 |

**Description:**

This subroutine checks the bit masks of the bytes in CPU register R14 (copy of VDP#0 & VDP#1) to determine how many columns there are in a row. This routine is used by some of the other VDP subroutines in spectra2.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## YX2PNT

Get VDP Pattern-Name-Table address for cursor YX position

|  |  |
| --- | --- |
| YX2PNT – Parameter in memory location | |
| **Call format** | MYTEST BL @YX2PNT |
| **Input** | @WYX |
| **Output** | TMP0 = VDP destination address |
| **Example** | /samples/example4.a99 |

**Description:**

This subroutine calculates the VDP address of the entry in the Pattern Name Table that matches with the cursor YX position (@WYX). The formula used is:

**VDP address = @WBASE + (Y \* columns per row) + X**

Note that the memory location @WBASE holds the VRAM base address of the VDP Pattern Name Table.

The subroutine checks the bit masks of the 2 bytes that make up CPU register R14 (copy of VDP#0 & VDP#1) to determine how many columns there are in a row. This routine is used by some of the other VDP subroutines in spectra2.

You can use multiple “virtual screens” by first loading @WBASE with the address of another PNT table.

Please refer to the TMS9918 VDP programmer’s guide for further details on the Pattern Name Table.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## YX2PX / YX2PXX

Get pixel position for cursor YX position

|  |  |
| --- | --- |
| YX2PX – Parameter in memory location | |
| **Call format** | MYTEST BL @YX2PX |
| **Input** | @WYX = YX value-pair  (CONFIG:0 = 1) = Skip sprite adjustment |
| **Output** | TMP0HB = Y pixel position  TMP0LB = X pixel position |
| **Example** | /samples/example4.a99 |

|  |  |
| --- | --- |
| YX2PXX – Parameter in register | |
| **Call format** | MYTEST BL @YX2PXX |
| **Input** | TMP0 = YX value-pair  (CONFIG:0 = 1) = Skip sprite adjustment |
| **Output** | TMP0HB = Y pixel position  TMP0LB = X pixel position |
| **Example** | /samples/example4.a99 |

**Description:**

This subroutine converts the tile based cursor YX position into the corresponding Y,X pixel coordinates using the below formula:

**Pixel Y = (Tile Y) \* 8**

**Pixel X = (Tile X) \* 8**

On subroutine exit, the most significant byte of register TMP0 will contain the Y pixel position and the least significant byte of register TMP0 will contain the X pixel position.

The functionality is useful for setting the sprite position based on the position of a tile.

Note that for sprites the top of screen is at >FF, not at >00. The subroutine automatically does the necessary adjustment. This feature can be turned off by setting bit 0 in the CONFIG register.

**Also note that the subroutine does not support multicolor and text mode.**

Please refer to the TMS9918 VDP programmer’s guide for further details on the Sprite Attribute Table.

|  |  |
| --- | --- |
| **VDP low-level** |  |

## PX2YX

Get tile YX position for pixel YX position

|  |  |
| --- | --- |
| PX2YX – Parameter in register | |
| **Call format** | MYTEST BL @PX2YX |
| **Input** | TMP0 = YX value-pair  (CONFIG:0 = 1) = Skip sprite adjustment |
| **Output** | TMP0HB = Y tile position  TMP0LB = X tile position  TMP1HB = Y pixel offset  TMP1LB = X pixel offset |
| **Example** | /samples/????.a99 |

**Description:**

This subroutine converts a –sprite- YX pixel position into the corresponding Y,X tile coordinates using the below formula:

**Tile Y = (Pixel Y) / 8**

**Tile X = (Pixel X) / 8**

**Offset Y = (Pixel Y) modulus 8**

**Offset X = (Pixel X) modulus 8**

On subroutine exit, the most significant byte of register TMP0 will contain the Y tile position and the least significant byte of register TMP0 will contain the X tile position.

The most significant byte of register TMP1 contains the Y offset.

The least significant byte of register TMP1 contains the X offset.

Both the Y and X offset are expressed in pixels.

The functionality is useful for setting a character tile based on the position of a sprite.

Note that for sprites the top of screen is at >FF, not at >00. The subroutine automatically does the necessary adjustment. This feature can be turned off by setting bit 0 in the CONFIG register.

**Also note that the subroutine does not support multicolor and text mode.**

Please refer to the TMS9918 VDP programmer’s guide for further details on the Sprite Attribute Table.

# VDP tiles & patterns subroutines

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## LDFNT

Load TI-99/4A character font from GROM into VRAM

|  |  |
| --- | --- |
| LDFNT – Parameter in DATA statement | |
| **Call format** | MYTEST BL @LDFNT  DATA P0,P1 |
| **Input** | P0 = VDP VRAM destination address  P1 = Font options |
| **Example** | runlib.a99 |

**Description:**

The LDFNT subroutine is used to copy the built-in character font from the TI-99/4A operating system GROMs into VDP VRAM memory.

We can save valuable ROM space by using the fonts available in the TI-99/4A itself. Note that it’s also possible to apply a “bold” effect to the fonts. That way you get a new font that looks nice for games.

Parameter P0 must contain the VDP destination address.

Below are the possible values for parameter P1.

FNOPT1 EQU >0000 ; LDFNT => Load TI title screen font

FNOPT2 EQU >0006 ; LDFNT => Load upper case font

FNOPT3 EQU >000C ; LDFNT => Load upper/lower case font

FNOPT4 EQU >0012 ; LDFNT => Load lower case font

FNOPT5 EQU >8000 ; LDFNT => Load TI title screen font & make fat

FNOPT6 EQU >8006 ; LDFNT => Load upper case font & make fat

FNOPT7 EQU >800C ; LDFNT => Load upper/lower case font & make fat

FNOPT8 EQU >8012 ; LDFNT => Load lower case font & make fat

The LDFNT routine is automatically called when the spectra2 library is initalized.

Please also see details on the SPFONT equate in the “Library startup options” section on page 21.

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## PUTSTR

Put length-byte prefixed string at cursor position

|  |  |
| --- | --- |
| PUTSTR – Parameter in DATA statement | |
| **Call format** | MYTEST BL @PUTSTR  DATA P0 |
| **Input** | P0 = Pointer to length-byte prefixed string  @WYX = Cursor YX position |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| XUSTSTR – Parameter in register | |
| **Call format** | MYTEST BL @XUTSTR |
| **Input** | TMP0 = Pointer to length-byte prefixed string  @WYX = Cursor YX position |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | YX2PNT, XPYM2V |

**Description:**

The PUTSTR subroutine is used to display a length-byte prefixed string at the current cursor position (@WYX). Both rows and columns start with 0.

In other words: the 1st row, 1st column is at YX position 0,0.

Parameter P0 must contain the address of the string to display.

The first byte of that string must contain the string length.

The subroutine supports string with a maximum length of 255 characters. There are no boundary checks. It is for example possible to display a string on row 85. That makes it possible to do some cool effects when working with multiple “virtual screens”.

**Example:**

Display string “Hello World” on row 5, column 15

MAIN LI R0,>050F ; Y=5, X=15

MOV R0,@WYX ; Load cursor

BL @PUTSTR ; Display string

DATA HELLOW

B @TMGR ; Handle FCTN-QUIT key, etc.

HELLOW BYTE 12

TEXT ‘HELLO WORLD!’

END

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## PUTAT

Put length-byte prefixed string at position Y,X

|  |  |
| --- | --- |
| PUTAT – Parameter in DATA statement | |
| **Call format** | MYTEST BL @PUTAT  DATA P0 |
| **Input** | P0 = YX position  P1 = Pointer to length-byte prefixed string |
| **Example** | /samples/example1.a99 |

|  |  |
| --- | --- |
| **Dependencies** | PUTSTR, YX2PNT, XPYM2V |

**Description:**

The PUTAT subroutine is used to display a length-byte prefixed string at the cursor position specified in parameter P0.

The most-significant byte of parameter P0 must contain the row value, the least-significant byte of P0 contains the column value. Both rows and columns start with 0.

In other words: the 1st row, 1st column is at YX position 0,0.

**Note that this subroutine overwrites the cursor YX position (@WYX).**

Parameter P1 must contain the address of the string to display.

The first byte of that string must contain the string length.

The subroutine supports strings with a maximum length of 255 characters. There are no boundary checks. It is for example possible to display a string on row 85. That makes it possible to do some cool effects when working with multiple “virtual screens”.

**Example:**

Display string “Hello World” on row 5, column 15

MAIN BL @PUTSTR ; Display string

DATA >050F,HELLOW

B @TMGR ; Handle FCTN-QUIT key, etc.

HELLOW BYTE 12

TEXT ‘HELLO WORLD!’

END

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## HCHAR

Repeat characters horizontally at position Y,X

|  |  |
| --- | --- |
| HCHAR – Parameter in DATA statement | |
| **Call format** | MYTEST BL @HCHAR  DATA P0,P1  …  DATA EOL |
| **Input** | P0 = YX position  P1 = MSB: Character to write  LSB: Number of times to repeat |
| **Example** | /samples/example2.a99 |

|  |  |
| --- | --- |
| **Dependencies** | YX2PNT, XFILV |

**Description:**

The HCHAR subroutine is comparable to the TI-Basic CALL HCHAR statement. It repeats characters horizontally.

The most-significant byte of parameter P0 must contain the row value, the least-significant byte of P0 must contain the column value. Both rows and columns start with 0.

In other words: the 1st row, 1st column is at YX position 0,0.

**Note that this subroutine overwrites the cursor YX position (@WYX).**

The most-significant byte of Parameter P1 must contain the character to write. The least-significant byte of Parameter P1 must contain the number of times the character should be repeated.

The HCHAR subroutine expects a list of parameters. With one HCHAR call you can draw multiple horizontal lines. **You need to specify the End-Of-List marker in the last DATA statement by using the EOL equate.**

Also note that there are no boundary checks. It is for example possible to do a HCHAR on row 85. This feature is especially useful when working with multiple “virtual screens”.

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## VCHAR

Repeat characters vertically at position Y,X

|  |  |
| --- | --- |
| VCHAR – Parameter in DATA statement | |
| **Call format** | MYTEST BL @VCHAR  DATA P0,P1  …  DATA EOL |
| **Input** | P0 = YX position  P1 = MSB: Character to write  LSB: Number of times to repeat |
| **Example** | /samples/game/hc\_source2.a99 |

|  |  |
| --- | --- |
| **Dependencies** | GTCLMN, YX2PNT |

**Description:**

The VCHAR subroutine is comparable to the TI-Basic CALL VCHAR statement. It repeats characters vertically.

The most-significant byte of parameter P0 must contain the row value. The least-significant byte of P0 must contain the column value. Both rows and columns start with 0.

In other words: the 1st row, 1st column is at YX position 0,0.

**Note that this subroutine overwrites the cursor YX position (@WYX).**

The most-significant byte of Parameter P1 must contain the character to write. The least-significant byte of Parameter P1 must contain the number of times the character should be repeated.

The VCHAR subroutine expects a list of parameters. With one VCHAR call you can draw multiple vertical lines. **You need to specify the End-Of-List marker in the last DATA statement by using the EOL equate.**

Also note that there are no boundary checks. It is for example possible to do a VCHAR on row 85. This feature is especially useful when working with multiple “virtual screens”.

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## FILBOX

Fill box with characters at position Y,X

|  |  |
| --- | --- |
| FILBOX – Parameter in DATA statement | |
| **Call format** | MYTEST BL @FILBOX  DATA P0,P1  …  DATA EOL |
| **Input** | P0HB = Upper left corner Y  P0LB = Upper left corner X  P1HB = Width  P1LB = Height  P2HB = >00  P2LB = Character to fill |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | YX2PNT, XFILV |

**Description:**

The FILBOX subroutine fills the specified rectangular area with characters.

The most-significant byte of parameter P0 must contain the row value. The least-significant byte of P0 must contain the column value. Both rows and columns start with 0.

In other words: the 1st row, 1st column is at YX position 0,0.

**Note that this subroutine overwrites the cursor YX position (@WYX).**

The most-significant byte of parameter P1 specifies the width of the area. The least-significant byte of parameter P1 specifies the height of the area.

The most-significant byte of parameter P2 should be set to the byte value >00 and is not used. The least-significant byte of parameter P2 specifies the character for filling the area.

The FILBOX subroutine handles multiple data statements. **You need to specify the End-Of-List marker in the last DATA statement by using the EOL equate.**

Also note that there are no boundary checks. It is for example possible to do a FILBOX call for row 85. This feature is especially useful when working with multiple “virtual screens”.

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## PUTBOX

Put length-prefixed string in box at position Y,X

|  |  |
| --- | --- |
| PUTBOX – Parameter in DATA statement | |
| **Call format** | MYTEST BL @PUTBOX  DATA P0,P1  …  DATA EOL |
| **Input** | P0HB = Upper left corner Y  P0LB = Upper left corner X  P1HB = Width  P1LB = Height  P2 = Pointer to length-prefixed string |
| **Example** | /samples/game/hc\_source2.a99 |

|  |  |
| --- | --- |
| **Dependencies** | YX2PNT, XFILV |

**Description:**

The PUTBOX subroutine fills the specified rectangular area with the length-prefixed string.

The most-significant byte of parameter P0 must contain the row value. The least-significant byte of P0 must contain the column value. Both rows and columns start with 0.

In other words: the 1st row, 1st column is at YX position 0,0.

**Note that this subroutine overwrites the cursor YX position (@WYX).**

The most-significant byte of parameter P1 specifies the width of the area. The least-significant byte of parameter P1 specifies the height of the area.

Parameter P2 must contain the address of the string to display in the area. The first byte of that string must contain the string length. The subroutine supports string with a maximum length of 255 characters.

**Note that if the string is too short for filling the whole rectangular area, it will be automatically repeated until it fits.**

The PUTBOX subroutine handles multiple data statements. **You need to specify the End-Of-List marker in the last DATA statement by using the EOL equate.**

Also note that there are no boundary checks. It is for example possible to do a PUTBOX call for row 85. This feature is especially useful when working with multiple “virtual screens”.

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## MKNUM

Convert unsigned number to right-justified string

|  |  |
| --- | --- |
| MKNUM – Parameter in DATA statement | |
| **Call format** | MYTEST BL @MKNUM  DATA P0,P1,P2 |
| **Input** | P0 = Pointer to 16 bit unsigned number  P1 = Pointer to 5 byte string buffer  P2HB = Offset for ASCII digit  P2LB = Character for replacing leading 0's  Optional  (CONFIG:0 = 1) = Display number at cursor YX  @WYX = Cursor YX position |
| **Output** | 5 byte string buffer will contain converted number |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | XUTSTR |

**Description:**

The MKNUM subroutine converts a 16 bit unsigned number (0-65535) into a right-justified string.

Parameter P0 must contain the address of the memory location holding the 16 bit unsigned number.

Parameter P1 must contain the address of a working buffer in RAM (5 bytes). This buffer will also contain the generated string.

The most-significant byte of parameter P2 must contain the ASCII offset for digit 0. The offset depends on what ASCII characters you use for holding the digits 0-9. If you for example load patterns for 0-9 overriding characters A-J, then you would load the byte value >41 (decimal 65).

This functionality is useful, if you have multiple characters sets for displaying a score (e.g. with different colours) or if you relocated the digits to a more suitable location in the pattern table.

The least-significant byte of parameter P2 must contain the ASCII value of the padding character. This character will be used for replacing the leading 0’s. That could for example be a white-space character or the ASCII value of the character holding digit 0.

Suppose you have the value “123”. Using the MKNUM subroutine you could convert it to the string “00123” or “ 123”.

Following equates are available for parameter P2:

NUM1 EQU >3030 ; MKNUM => ASCII 0-9, leading 0's

NUM2 EQU >3020 ; MKNUM => ASCII 0-9, leading spaces

**You can optionally display the generated string at the current cursor YX position by setting bit 0 in the CONFIG register.**

|  |  |
| --- | --- |
| **VDP tiles & patterns** |  |

## PUTNUM

Put unsigned number on screen

|  |  |
| --- | --- |
| PUTNUM – Parameter in DATA statement | |
| **Call format** | MYTEST BL @PUTNUM  DATA P0,P1,P2,P3 |
| **Input** | P0 = YX position  P1 = Pointer to 16 bit unsigned number  P2 = Pointer to 5 byte string buffer  P3HB = Offset for ASCII digit  P3LB = Character for replacing leading 0's |
| **Output** | 5 byte string buffer will contain converted number |
| **Example** | /samples/example5.a99 |

|  |  |
| --- | --- |
| **Dependencies** | MKNUM, XUTSTR |

**Description:**

The PUTNUM subroutine converts a 16 bit unsigned number (0-65535) into a right-justified string and displays it on screen.

The most-significant byte of parameter P0 must contain the row value. The least-significant byte of P0 must contain the column value.

Both rows and columns start with 0.

In other words: the 1st row, 1st column is at YX position 0,0.

**Note that this subroutine overwrites the cursor YX position (@WYX).**

Parameter P1 must contain the address of the memory location holding the 16 bit unsigned number.

Parameter P2 must contain the address of a working buffer in RAM (5 bytes). This buffer will also contain the generated string.

The most-significant byte of parameter P3 must contain the ASCII offset for digit 0. The offset depends on what ASCII characters you use for holding the digits 0-9. If you for example load patterns for 0-9 overriding characters A-J, then you would load the byte value >41 (decimal 65).

This functionality is useful, if you have multiple characters sets for displaying a score (e.g. with different colors) or if you relocated the digits to a more suitable location in the pattern description table.

The least-significant byte of parameter P3 must contain the ASCII value of the padding character. This character will be used for replacing the leading 0’s. That could for example be a white-space character or the ASCII value of the character holding digit 0.

Suppose you have the value “123”. Using the PUTNUM subroutine you could display the string “00123” or “ 123”.

Following equates are available for parameter P3:

NUM1 EQU >3030 ; MKNUM => ASCII 0-9, leading 0's

NUM2 EQU >3020 ; MKNUM => ASCII 0-9, leading spaces

# Sound & speech subroutines

|  |  |
| --- | --- |
| **SOUND & SPEECH** |  |

## MUTE

Mute all sound generators and clear sound pointer

|  |  |
| --- | --- |
| MUTE – No parameter | |
| **Call format** | MYTEST BL @MUTE |
| **Example** | /samples/game/hc\_source2.a99 |

**Description:**

The MUTE subroutine is used for muting all sound generators. It additionally clears memory location @WSDLST (address of tune currently playing) and turns off the sound player by resetting bit 13 in the CONFIG register.

For further details please refer to the SDPREP and SDPLAY subroutines.

|  |  |
| --- | --- |
| **SOUND & SPEECH** |  |

## MUTE2

Mute all sound generators

|  |  |
| --- | --- |
| MUTE2 – No parameter | |
| **Call format** | MYTEST BL @MUTE2 |
| **Example** | /samples/game/hc\_source2.a99 |

**Description:**

The MUTE2 subroutine is used for muting all sound generators. It additionally turns off the sound player by resetting bit 13 in the CONFIG register.

However, subroutine MUTE2 does not clear memory location (@WSDLST).

So due to this, you basically use MUTE2 for pausing the sound player.

For further details please refer to the SDPREP and SDPLAY subroutines.

|  |  |
| --- | --- |
| **SOUND & SPEECH** |  |

## SDPREP

Prepare for playing sound

|  |  |
| --- | --- |
| SDPREP – Parameter in DATA statement | |
| **Call format** | MYTEST BL @SDPREP  DATA P0,P1 |
| **Input** | P0 = Address of tune  P1 = Option flags for sound player |
| **Example** | /samples/game/hc\_source1.a99 |

**Description:**

The SDPREP subroutine initializes the CONFIG register bits 13-15 and sets some memory addresses (@WSDLST, @WSDTMP) used by the built-in sound player. It also loads the least-significant byte of R13 with 1.

The sound player (SDPLAY) itself is automatically called by the kernel background thread on each VDP interrupt.

Parameter P0 contains the address of the tune to play. Note that the tune data must already be present in either ROM/RAM or VRAM.

The sound table format is compatible to the format supported by the ISR sound routine found in the console ROM.

Parameter P1 contains the option flags for the tune. It specifies if the tune should be played from ROM/RAM or VRAM. Additionally it specifies if the tune should automatically start over when finished.

The below equates are available for parameter P1

SDOPT1 EQU 7 ; SDPLAY => 111 (Player on, repeat, tune in CPU memory)

SDOPT2 EQU 5 ; SDPLAY => 101 (Player on, no repeat, tune in CPU memory)

SDOPT3 EQU 6 ; SDPLAY => 110 (Player on, repeat, tune in VRAM)

SDOPT4 EQU 4 ; SDPLAY => 100 (Player on, no repeat, tune in VRAM)

Please refer to the Editor/Assembler manual page 312 for details on the ISR sound table format.

|  |  |
| --- | --- |
| **SOUND & SPEECH** |  |

## SDPLAY

Run the sound player

|  |  |
| --- | --- |
| SDPLAY – No parameter | |
| **Call format** | MYTEST BL @SDPLAY |
| **Example** | - |

**Description:**

The SDPLAY subroutine is the built-in sound player. It is normally automatically called by the background kernel thread on each VDP interrupt. It means this code is executed 60 times a second on NTSC and 50 times a second on a PAL machine.

The sound format is compatible to the sound format of the ISR sound routine found in the console ROM.

It’s still possible to call the SDPLAY subroutine from your program in case you are not using the background kernel thread. That’d allow for some custom effects like slowing down or speeding up a tune.

The SDPREP subroutine must be used for setting up memory before a tune can be played.

The sound player uses bit 13,14,15 in the CONFIG register. You can turn off the sound player by setting bit 13 to 0. You have to use the MUTE subroutine if a tune is already in progress.

Please refer to the Editor/Assembler manual page 312 for details on the ISR sound table format.

|  |  |
| --- | --- |
| **SOUND & SPEECH** |  |

## SPSTAT

Read status register byte from speech synthesizer

|  |  |
| --- | --- |
| SPSTAT – No parameter | |
| **Call format** | MYTEST LI TMP2,MYRET  B @SPSTAT |
| **Output** | MSB TMP0 = speech synth status code |
| **Example** | runlib.a99 |

**Description:**

The SPSTAT subroutine is used for checking the speech synth FIFO buffer fill grade. You normally do not need to run this subroutine in your program, as it’s automatically handled by the built-in speech player (SPPLAY).

Nonetheless, should you need to call the SPSTAT subroutine, you’ll have to use “B @SPSTAT” after loading register TMP2 with the return address to branch to upon subroutine exit.

Upon exit register TMP0 will contain the speech synthesizer status code.

Note that the 32K memory expansion is not available when the speech synthesizer status register is accessed. Therefore the SPSTAT subroutine loads and executes some machine code in scratchpad memory (>8320 - >8327).

Please refer to the Editor Assembler manual, section 22 page 352 for further details on using speech on the TI-99/4A.

Also see the TMS5220 Speech Synthesizer Data Manual, section 5.2 (FIFO Buffer) and section 5.4 (Status Register)

|  |  |
| --- | --- |
| **SOUND & SPEECH** |  |

## SPCONN

Check if speech synthesizer is connected

|  |  |
| --- | --- |
| SPCONN – No parameter | |
| **Call format** | MYTEST BL @SPCONN |
| **Output** | LSB TMP0 = AA if speech synthesizer found  CONFIG#10 = 1/0 |
| **Example** | /samples/????.a99 |

|  |  |
| --- | --- |
| **Dependencies** | SPSTAT |

**Description:**

The SPCONN subroutine is used for checking if the speech synthesizer is connected. Upon exit the least-significant byte of register TMP0 will contain the speech synthesizer status code.

The latter will equal >AA if a speech synthesizer is connected.

You normally do not need to call this subroutine in your program. The RUNLIB subroutine does that for you upon library initialisation and stores the results in bit 10 of the CONFIG register.

For further details please refer to section 22.1.6 page 354 in the Editor/Assembler manual.

Please refer to the library initialisation section for further details on RUNLIB and the CONFIG register usage.

|  |  |
| --- | --- |
| **SOUND & SPEECH** |  |

## SPPREP

Prepare for playing speech

|  |  |
| --- | --- |
| SPPREP – Parameter in DATA statement | |
| **Call format** | MYTEST BL @SPPREP  DATA P0 |
| **Input** | P0 = Address of LPC speech data |
| **Example** | /samples/example2.a99 |

**Description:**

The SPPREP subroutine prepares memory and the CONFIG register for playing speech. It loads the value of parameter P0 into memory location (@WSPEAK).

The speech player (SPPLAY) itself is automatically called by the thread scheduler routine (TMGR).

Parameter P0 specifies the memory address (ROM/RAM) where the LPC encoded speech data can be found.

|  |  |
| --- | --- |
| **SOUND & SPEECH** |  |

## SPPLAY

Run the speech player

|  |  |
| --- | --- |
| SPPLAY – No parameter | |
| **Call format** | MYTEST BL @SPPLAY |
| **Example** | - |

**Description:**

The SPPLAY subroutine is the built-in speech player. You normally do not need to call the SPPLAY subroutine from your program. This is automatically handled in the background by the thread scheduler (TMGR).

Communicating with the speech synthesizer device is very critical as far as timing is concerned. That is why the speech player code is called from inside the thread scheduler code itself.

The SPPREP subroutine must be used for setting up memory before speech can be activated.

The speech player (SPPLAY) included in spectra2 supports:

* **Playback recorded speech from an external source**

In that case the LPC encoded voice data is either available in cartridge ROM or loaded into RAM.

Note that the speech player uses bit 3,4,5 in the CONFIG register.

The speech player can be turned off by setting bit 3 in the CONFIG register to 0.

# Keyboard & joystick subroutines

|  |  |
| --- | --- |
| **KEYBOARD & JOYSTICKS** |  |

## VIRTKB

The virtual keyboard implementation

|  |  |
| --- | --- |
| KBSCAN – No parameter | |
| **Call format** | MYTEST BL @KBSCAN |
| **Output** | @WVRTKB |
| **Example** | runlib.a99 |

**Description:**

Spectra2 knows the concept of a “virtual keyboard”. It basically maps most game keys and joysticks 1 and 2 on a bit mask. The concept used to accomplish this is explained in the “Virtual Keyboard” section. Check there for examples, etc.

Normally there is no need to call the KBSCAN from your program. It’s automatically handled by the background kernel thread (KERNEL) which is part of the runtime library.

The VIRTKB subroutine uses bit 11 in the CONFIG register. Upon completion, the keys presses are stored as bit flags in the memory word @WVRTKB.

# Thread scheduler subroutines

|  |  |
| --- | --- |
| **Thread Scheduler** |  |

## TMGR

The thread scheduler

|  |  |
| --- | --- |
| TMGR – No parameter | |
| **Call format** | MYTEST B @TMGR |
| **Example** | /samples/example2.a99 |

**Description:**

The TMGR subroutine is the spectra2 thread scheduler. It’s pretty much the main subroutine responsible for running background jobs such as the kernel thread and any additional threads started by the user.

The “Thread Scheduler” section explains in detail how the scheduler works and how to use it. Check there for examples, etc.

You can start the scheduler with “**B @TMGR**” after initialisation in the main program has completed.

Make sure you checked the below before initiating TMGR, it will save you a lot of time searching for program crashes:

* Memory address WTITAB (2 bytes) set with address of your timer table.
* Timer table initialized with >00 bytes.
* Memory address BTIHI (1 byte!) set with highest timer slot in use.

|  |  |
| --- | --- |
| **Thread Scheduler** |  |

## MKSLOT

Allocate timer slots

|  |  |
| --- | --- |
| TMGR – Parameter in DATA statement | |
| **Call format** | MYTEST BL @MKSLOT  DATA P0,P1  …  DATA EOL |
| **Input** | P0HB = Slot number  P0LB = Repeat interval  P1 = Subroutine to call |
| **Example** | /samples/example5.a99 |

**Description:**

The MKSLOT subroutine is used to allocate timer slots for running threads. The subroutine allows you to allocate non-sequential slots, e.g. allocate slots 0,3,4,7 (without touching slots 1,2,5,6).

See the “Threads” sections for details on timer table layout.

The most significant byte of parameter P0 is the slot number to use. The amount of available slots is determined by the size of the timer table in RAM memory.

The least significant byte of parameter P0determines the interval at which the task scheduler should run the subroutine specified in parameter P1. The value for the interval is defined in ticks per second.

Parameter P1contains the address of the subroutine to call via BL when the slot is fired.

The MKSLOT subroutine handles multiple data statements. **You need to specify the End-Of-List marker in the last DATA statement by using the EOL equate.**

Make sure that you set the memory word @WTITAB (2 bytes) with the address of your timer table before calling MKSLOT the first time.

Don’t forget to update the memory location @WBTIHI (1 byte!) with the highest slot in use.

Note that if you have many slots to allocate at once, you could copy a preset slot table from ROM into RAM without using the MKSLOT subroutine

|  |  |
| --- | --- |
| **Thread Scheduler** |  |

## CLSLOT

Clear allocated timer slot

|  |  |
| --- | --- |
| CLSLOT – Parameter in DATA statement | |
| **Call format** | MYTEST BL @CLSLOT  DATA P0 |
| **Input** | P0 = Slot number |
| **Example** | /samples/game/hc\_source2.a99 |

|  |  |
| --- | --- |
| XLSLOT – Parameter in register | |
| **Call format** | MYTEST BL @XLSLOT |
| **Input** | TMP0 = Slot number |
| **Example** | /samples/????.a99 |

**Description:**

Use the CLSLOT subroutine to remove a single running slot. It means that the subroutine marked in the specified slot will no longer be executed.

Note that using CLSLOT does not re-arrange the remaining slots in the timer table. Due to this you can get holes in the timer table over time. It’s pretty much the responsibility of the programmer to keep track of what slots can be reused for new threads.

Parameter P0 must contain the slot number of the slot to clear.

|  |  |
| --- | --- |
| **Thread Scheduler** |  |

## KERNEL

The kernel thread

|  |  |
| --- | --- |
| KERNEL – No parameter | |
| **Call format** | MYTEST B @KERNEL |
| **Input** | P0 = Slot number |
| **Example** | - |

**Description:**

The KERNEL subroutine is used for doing certain basic background tasks such as running the sound player (SDPLAY) and scanning the virtual keyboard (VIRTKB). You can’t call the KERNEL subroutine directly from your program. It’s completely controlled by the Thread Scheduler code (TMGR).

The kernel thread can be deactivated by resetting bit 9 in the CONFIG register.

Please refer to the “Threads” section for further details on the kernel thread.

|  |  |
| --- | --- |
| **Thread Scheduler** |  |

## MKHOOK

Allocate the user hook

|  |  |
| --- | --- |
| MKHOOK – Parameter in DATA statement | |
| **Call format** | MYTEST BL @MKHOOK  DATA P0 |
| **Input** | P0 = Address of user hook |
| **Example** | /samples/game/hc\_source2.a99 |

**Description:**

The MKHOOK subroutine is responsible for allocating the user hook.

The idea is that you use the user hook for stuff that isn’t bound to the VDP interrupt and that needs to be executed very often (more than 50 or 60 times a second), e.g. checking the coincidence flag in the VDP status register.

Parameter P0 contains the address of the user hook, a user-supplied subroutine that is executed each time the VDP status register is read.

The MKHOOK routine will move the address in P0 to memory location @WHOOK. It then sets bit 7 and resets bit 8 in the CONFIG register.

Note that the user hook code must always exit with a "B @HOOKOK" for returning to the thread scheduler.

Please refer to the “Threads” section for the full details on the user hook concept.

# Miscellaneous subroutines

|  |  |
| --- | --- |
| **Miscellaneous** |  |

## POPR(0-3) or POPRT

Pop registers & return to caller

|  |  |
| --- | --- |
| POPR(0-3) or POPRT – No parameter | |
| **Call format** | MYTEST B @POPR3  MYTEST B @POPR2  MYTEST B @POPR1  MYTEST B @POPR0  MYTEST B @POPRT |
| **Example** | /samples/????.a99 |

**Description:**

These routines pop the specified registers from the stack and then returns to the caller. It expects that the return address (R11) is at the bottom.

Use POPRT if you only want to pop R11 and return.

**Note that –by default- STACK is an equate for R9.**

See the “Stack” section on page 40 for further details.

**Example:**

Suppose you have a subroutine MYTEST that changes R0 and R1. You want to make sure that upon subroutine exit R0 and R1 keep their original values.

MAIN LI R0,15  
 LI R1,22

BL @MYTEST ; Upon return; R0=15, R1=22  
 JMP $ ; Soft halt  
MYTEST DECT STACK

MOV R11,\*STACK ; Push R11 (return address)

DECT STACK

MOV R0,\*STACK ; Push R0 on stack (value 15)

DECT STACK

MOV R1,\*STACK ; Push R1 on stack (value 22)

LI R0,99 ; Overwrite R0

CLR R1 ; Overwrite R1

B @POPR1 ; Pop R1,R0,R11 from stack and return

|  |  |
| --- | --- |
| **Miscellaneous** |  |

## RND / RNDX

Generate random number

|  |  |
| --- | --- |
| RND – Parameter in DATA statement | |
| **Call format** | MYTEST BL @RND  DATA P0 |
| **Input** | P0 = Highest random number allowed |
| **Output** | TMP0 = Random number |
| **Example** | /samples/example5.a99 |

|  |  |
| --- | --- |
| RNDX – Parameter in register | |
| **Call format** | MYTEST BL @RNDX |
| **Input** | TMP0 = Highest random number allowed |
| **Output** | TMP0 = Random number |
| **Example** | /samples/example5.a99 |

**Description:**

The RND subroutine generates a new random number in the range between 0 and P0. The subroutine uses and updates the seed value stored in memory location @WSEED.

Parameter P0 must contain the highest number allowed.

The generated random number is returned in register TMP0.

The seed value in memory location @WSEED is populated for the first time when the library gets initialized. The value is copied from scratch-pad memory location @>83C0 which is set by the monitor OS.

The original seed value is based on a counter waiting for a key-press in the TI selection screen.

|  |  |
| --- | --- |
| **Miscellaneous** |  |

## RUNLIB

Initialize spectra2 runtime library

|  |  |
| --- | --- |
| RND – No parameter | |
| **Call format** | MYTEST B @RUNLIB |
| **Example** | /samples/example1.a99 |

|  |  |
| --- | --- |
| **Dependencies** | CPYM2M, CPYG2M, FILV, MUTE, VIDTAB, LDFNT |

**Description:**

The RUNLIB subroutine initializes the spectra2 runtime library. It must be the first thing that gets executed when a program is started.

It does many tasks as clearing RAM and VDP VRAM memory, setting the VDP in a defined state, checking the console it’s running on, etc.

It will jump to the main program (label MAIN), once it has completed the initialisation process.

For the full details please refer to the “Runtime library initialisation” section.

# Appendix: examples & source code