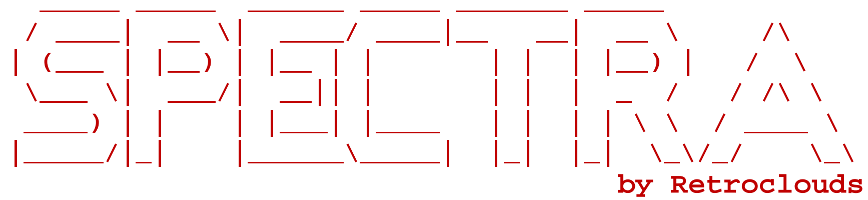
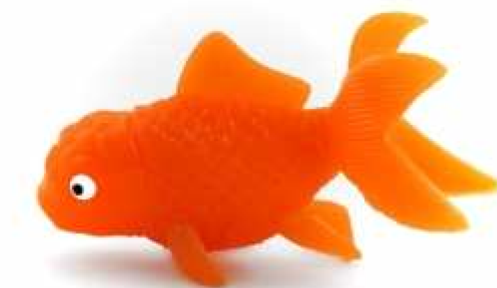


TMS9900 Arcade Game Library

for the

Texas Instruments TI-99/4A



REFERENCE MANUAL

August 2009

version 0.1.5

REVISION

Date	Author	Revision
August 1st 2009	Filip Van Vooren (Retroclouds)	Initial version SPECTRA version 0.1.5 (“Small Fish”)

TABLE OF CONTENTS

Introduction 7

What I need to know 8
License 8
License EPSGMOD player 8
How it all started 9
Compatibility 9
Serviceable parts inside 10
Graphics 10
Sound 10

Base 11

What I need to know 12
Required files 12
Stack 13
Memory layout 14
Variables in block 2 15

POPRG(0-9) 16
Pop registers & return to caller 16

POPRX(0-9) 17
Pop registers & return to caller (R11 already set) 17

FILMEM 19
Fill RAM memory range with byte 19

FILMEX 20
Fill RAM memory range with byte (register variant) 20

CPYM 21
Copy ROM/RAM memory range 21

CPYMX 22
Copy ROM/RAM memory range (register variant) 22

G2VDP 23
Copy GROM memory range to VDP memory 23

G2VDPX 24
Copy GROM memory range to VDP memory (register variant) 24

G2MEM 25
Copy GROM memory range to RAM memory 25

G2MEMX 26
Copy GROM memory range to RAM memory (register variant) 26

VDP low-level 27
FVRAM *28*
Fill VDP memory with byte 28
FVRAMX *29*
Fill VDP memory with byte (register variant) 29
PVRAM *30*
Copy memory range to VDP memory 30
VSBR *31*
Read single byte from VDP 31
VSBW *32*
Write single byte to VDP 32
VMBR *33*
Read multiple bytes from VDP 33
VMBW *34*
Write multiple bytes to VDP 34
VWTR *35*
Write to VDP register 35
LVDPSH *36*
Load VDP shadow registers in RAM with video mode table 36
WVDPSH *38*
Write VDP shadow registers from RAM to VDP write-only registers 38
VDPADR *39*
Calculate VDP table start address 39
VIDOFF *40*
Disable screen display 40
VIDON *41*
Enable screen display 41
XY2OF *42*
Calculate screen offset of X/Y character position 42

VDP Sprites 43
What I need to know 44
Copy of Sprite Attribute Table in RAM (Shadow SAT) 44
PUTSAT 46
Write shadow SAT (Sprite Attribute Table) from RAM to VDP memory 46
SPRORD 48
Initialize sprite order table to default sprite order 0..31 48
S8X8 49
Sprites with 8 x 8 pattern 49
S16X16 50
Sprites with 16 x 16 pattern 50
SMAG1X 51
Sprite magnification 1X 51
SMAG2X 52
Sprite magnification 2X 52
SPRITE 53
Create new sprite 53

VDP Tiles & Patterns 55
FILLSCR 56
Fill screen with character 56
FIBOX 57
Fill rectangular area with character 57
FIBOXX 58
Fill rectangular area with character (register variant) 58
PUTTX 59
Put length-byte prefixed string on screen 59
MIRRV 60
Mirror tile/sprite patterns in RAM memory buffer around vertical axis 60

Sound & Speech 61
EPSGMD 62
Setup memory for playing EPSGMOD tune 62

Timers 64

What I need to know *65*
Timer table 65
Timer slot format 65
Highest slot in use 66
Equates for accessing timer slots 66

MKSLOT *67*
Allocate specified timer slot 67

TMGR *68*
Timer Manager – the Spectra task scheduler 68

KBSCAN *71*
Scan the virtual TI-99/4A keyboard 71

Appendix 75

Overview video mode tables *76*

Introduction

What I need to know

Below you find some details regarding SPECTRA you definitely should know about before you start programming your next favourite assembler game.

License

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or at your option any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

License EPSGMOD player

Below I've included Tursi's conditions dated July 14th 2009 for the usage of the EPSGMOD tracker in SPECTRA:

> You may use my version of the EPSGMod tracker, and any future versions I release for the TI-99/4A and compatible machines, in your library, with any license of your choice (including any version of the GPL), subject to the following conditions:
>
> 1) You must get permission from KonTechs - you'll find his contact form on the webpage I linked above. If you can't reach him let me know and I'll forward a note on your behalf. You may include this acceptance on my part.
>
> 2) This does not license KonTech's own code, tracker, or any property not created by myself personally or HarmlessLion LLC.
>
> 3) You keep a credit to both myself and KonTechs, with web links, in your distribution both in documentation and source.
>
> 4) It is always free to download and use your library, and to distribute software based on said library.
>
> 5) I have the right to cancel this permission for future releases of the code, if I have to, by including a note in the distribution to that effect.

I also got the permission from Martin Konrad (Kontechs) for the usage of Tursi's EPSGMOD tracker in SPECTRA. I received the below at July 16th 2009:

> You have my permission for including Tursi's EPSGMOD-player in your distribution, as long as the distribution is free. But please inform me about further usages of the player. I've sent a copy of this to Tursi, too.

How it all started

The idea for the 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 Pitfall! very closely and I learned that the game called some subroutines that are stored in the console's built-in ROM. Doing some research in the internet revealed that the built-in Colecovision ROM contains a BIOS; in this case 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. Surprisingly the OS7 functions are not used that often in homebrew Colecovision games, as they are reported to be slow.

Nonetheless, it inspired me to start working on a similar library for the TI-99/4A. I wanted an open-source library that allows me to concentrate on the actual homebrew game without having to start writing all subroutines from scratch over and over again.

I believe in free software and therefore it is my wish that SPECTRA is free and stays free in the future. This is the reason I opted for distribution with a GPL license.

One size doesn't fit all

SPECTRA is perhaps a bit different compared to already existing libraries. First of all I believe in source code. As a matter of fact I don't like libraries where I don't have any access to the source code. For sure I'm not pretending to understand all, but at least I want to have the option to take a peek in the source code if something isn't working the way I expect it.

So the concept behind SPECTRA is that instead of loading the full library, you just include the subroutines that you need for getting the job done. It will save you some memory….

The library is targeted for cross-development on a PC. Even on an old PC assembly times are so fast that I don't see a big benefit in only using the already assembled object files. Actually there are some big benefits on programming TMS9900 assembly games on a PC. Besides the fact that you can always have your DEV environment with you, the biggest advantage is that there are actually some cool emulators around, with at least one having a good debugger. Last but not least I'm planning on writing a new TMS9900 cross-assembler especially for writing games on the TI-99/4A supporting SPECTRA.

Compatibility

The source code in SPECTRA is compatible with Burrsofts' Asm994A Assembler V3.008 This great cross-assembler for Windows is not part of the SPECTRA library, but can be obtained directly at BurrSoft (http://www.99er.net/win994a.shtml).

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 extend, but comes without any warranties whatsoever. There are still plenty of bugs inside and if you find any, let me know and I'll try to fix them. Better still, why don't you fix them yourself and send me the updated version.

This not only counts for corrections: I would like to see SPECTRA become a community project. So if you find this library useful, then why don't you contribute by adding some new functionality. We can then all benefit from it.

Visit the project home page at sourceforge (http://spectra99,sourceforge.net) for obtaining the newest version. You can also report bugs there, discuss SPECTRA in the forum and bring in some new ideas.

Graphics

Writing a homebrew game is not only about programming, a really big part has to do with implementing graphics and sound effects. For creating graphics I have bundled SPECTRA with a small utility called "Tile Studio Converter" that allows you to easily generate sprite and character patterns of graphics created with Tile Studio.

Tile Studio (http://tilestudio.sourceforge.net) by Wiering Software is a very powerful open source development utility for graphics of tile-based games.
This windows software is NOT included in SPECTRA, but is available for free via the mentioned web link.

Sound

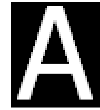
With the kind permission of both Tursi and Kontechs I have included a slightly modified version of Tursi's EPSGMOD (http://www.harmlesslion.com) for playing sophisticated sounds and tunes on the TI-99/4A.

Basically the idea is that you create the tune using Kontechs Mod2PSG2 (http://mod2psg2.kontechs.de). Mod2PSG2 is a very powerful music tracker for the SN76489 sound chip that is used in the TI-99/4A, Colecovision, SEGA Master System, …
This windows software is NOT included in SPECTRA, but is available for free via the mentioned web link.

See section "Sound & Speech" for details about the player and refer to the license section for further details on the EPSGMOD usage conditions.

Base

BASE



What I need to know

Below you find some details you definitely should know about before you start programming your next favourite assembler game.

Required files

The idea is that you only include those parts of SPECTRA that are useful in your game implementation . However, there is one file called **spectra_base.a99** that always must be included in your source code. Reason is that it contains a header section that is used for setting up memory and stack, has some basic equates, etc.
As such this file is a prerequisite for all of the subroutines in the library.

Below is the preferred order for including SPECTRA in your game source.

File	
your_game_header.a99	
your_game1.a99	
your_game2.a99	
….	
spectra_base.a99	Always required
spectra_memcpy.a99	Optional. Depends on your game requirements
spectra_vdp.a99	
spectra_tiles.a99	
spectra_sprites.a99	
spectra_timers.a99	
spectra_epsgmod.a99	
spectra_ctrl_virtkb.a99	
your_game_end.a99	

Stack

All subroutines in SPECTRA are called via the branch-and-link (BL) instruction. When a subroutine is called, it first pushes the registers it is about to change on the stack. On subroutine exit they are then popped from the stack again by branching to the appropriate utility routines (POPRGx or POPRXx).
This means that your registers will NOT be destroyed after calling any of the subroutines in the library.

STACK equate

Now for sure you already know that there is no hardware stack pointer in a TMS9900 CPU. As a workaround the stack pointer is simulated by using one of the 16 registers in the workspace.

In the default configuration SPECTRA uses register R8 as stack pointer.

You can use another register for this purpose by changing the STACK equate in **spectra_base.a99**. However, keep in mind to store/restore the correct values when fiddling with this register.

Be aware that the stack in SPECTRA grows from low memory towards high memory.

STKBUF equate

This equate must contain the address of the memory location that is at the bottom of the stack. The value for this equate depends on the used SPECTRA memory layout. When using the default configuration this will be BLOCK 4 in scratch-pad memory.

By changing STKBUF in **spectra_base.a99** you can decide where the stack will be allocated in memory.

Make sure that on program start STKBUF points to a memory area that is properly initialized.
The stack size can grow to 64 bytes when using the default memory configuration.

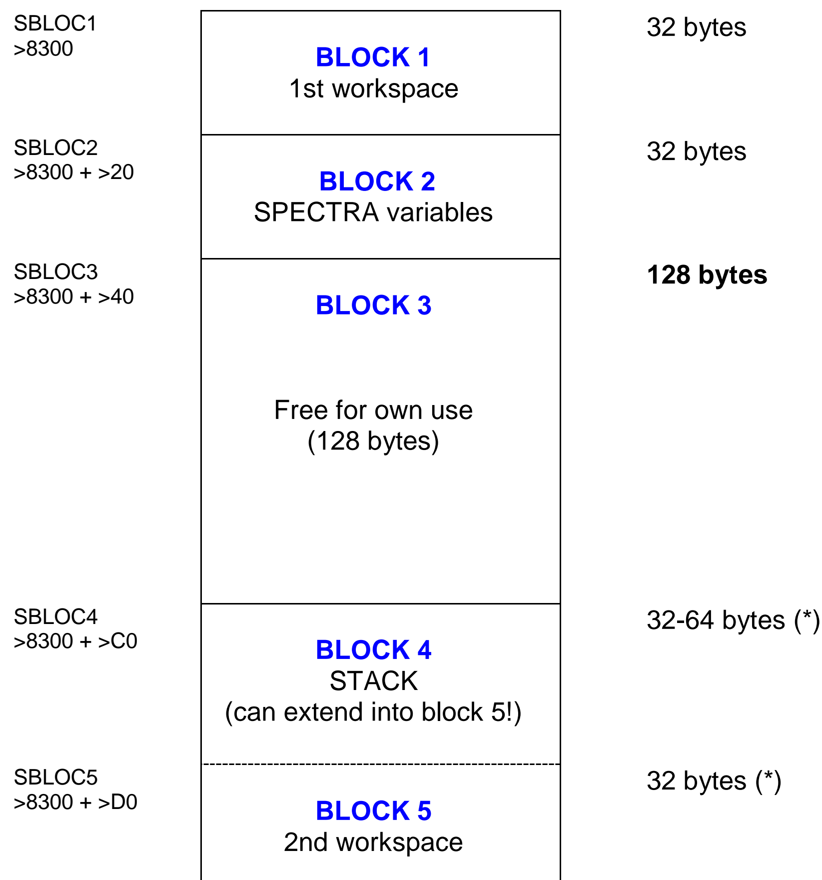
Memory layout

The memory layout used by SPECTRA is quite flexible. By default it uses all of scratchpad memory. However you could reconfigure it in such way that none or only part of the scratchpad memory is used. That could be helpful if you want to setup your own ISR hook.

Memory configuration is done in **spectra_base.a99** and is basically divided in 5 blocks:

- Block 1: Contains the main workspace (R0-R15)
- Block 2: Contains all variables used by SPECTRA
- Block 3: **Free for own use**
- Block 4: Subroutine stack
- Block 5: Contains second workspace (R0-R15). Is normally not used but is available as workspace for your BLWP subroutines. **Keep in mind however that block 4 can grow into block 5 so you will have to check subroutine dependencies first when using block 5 (*)**

Note that blocks 1, 2 and 4 are crucial for SPECTRA to work properly.



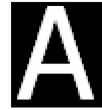
Below you find the memory header section as included in the file **spectra_base.a99**

```
*//////////////////////////////////////////////////////////////
*                SPECTRA SCRATCH-PAD MEMORY USAGE
*//////////////////////////////////////////////////////////////
SCRPAD  EQU   >8300                  ;   256 bytes scratch-pad
SPCMEM  EQU   SCRPAD
SBLOC1  EQU   SPCMEM                 ;   \ Length   32  bytes
SBLOC2  EQU   SPCMEM+>20             ;   | Length   32  bytes
SBLOC3  EQU   SPCMEM+>40             ;   | Length  128  bytes     -> FREE FOR OWN USE
SBLOC4  EQU   SPCMEM+>C0             ;   | Length   32  bytes  \  -> SBLOC4 can extend
SBLOC5  EQU   SPCMEM+>D0             ;   / Length   32  bytes  /     to 64 bytes
*//////////////////////////////////////////////////////////////
*                            BLOCK 1
*//////////////////////////////////////////////////////////////
WSSPC1  EQU   SPCMEM                 ;   32: SPECTRA workspace 1
*//////////////////////////////////////////////////////////////
*                            BLOCK 2
*//////////////////////////////////////////////////////////////
VIRTKB  EQU   SBLOC2                 ;    2: Virtual keyboard
VDPREG  EQU   SBLOC2+2               ;   16: VDP shadow registers
VDPSTA  EQU   SBLOC2+18              ;    2: VDP shadow status register
VDPCOL  EQU   SBLOC2+20              ;    2: VDP screen columns
TCOUNT  EQU   SBLOC2+22              ;    2: Timer Tick counter
THIGH   EQU   SBLOC2+24              ;    2: Timer highest slot in use
OUTP0   EQU   SBLOC2+26              ;    2: Return parameter 0
FUTU1   EQU   SBLOC2+28              ;    2: RESERVED FOR FUTURE USE
FUTU2   EQU   SBLOC2+30              ;    2: RESERVED FOR FUTURE USE
*//////////////////////////////////////////////////////////////
*                            BLOCK 3
*//////////////////////////////////////////////////////////////
FREE    EQU   SBLOC3                 ;  128: Free for own use
*//////////////////////////////////////////////////////////////
*                          BLOCK 4 - 5
*//////////////////////////////////////////////////////////////
STKBUF  EQU   SBLOC4                 ;   32: Stack (extends to 64!)
WSSPC2  EQU   SBLOC5                 ;   32: SPECTRA workspace 2
```

Variables in block 2

Please keep in mind that the SPECTRA memory layout is subject to change in future versions. This particularly applies to the variables used in block 2 (SBLOC2).

EQUATE	SIZE (bytes)	Description / Used by …
VIRTKB	2	Virtual keyboard controller - Virtual keyboard status **spectra_ctrl_keyb.a99**
VDPREG	16	VDP shadow registers (VDP#R0-R7) **spectra_vdp.a99, spectra_tiles.a99, spectra_sprites.a99**
VDPSTA	2	VDP shadow status register **spectra_vdp.a99, spectra_tiles.a99, spectra_sprites.a99**
VDPCOL	2	Number of columns in a row **spectra_vdp.a99, spectra_tiles.a99, spectra_sprites.a99**
TCOUNT	2	Timer Manager - Internal tick counter **spectra_timers.a99**
THIGH	2	Timer Manager - Highest timer slot in use **spectra_timers.a99**
OUTP0	2	Subroutine output parameter 0
FUTU1	2	RESERVED FOR FUTURE USE
FUTU2	2	RESERVED FOR FUTURE USE



POPRG(0-9)

Pop registers & return to caller

Main Category: **RAM**

File **spectra_base.a99**
Keywords RAM, LOW-LEVEL

Call format	MYTEST B @POPRG2
Input	-
Output	-
Stack usage	-

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.

Note that –by default- STACK is an equate for R8.
See the "what I need to know – Stack" section for details on stack usage.

Example:

Suppose you have a subroutine MYTEST defined that changes R0-R2 and you want to make sure that when you call MYTEST upon return R0-R2 have their original values again.

```
MAIN            LI      R0,15
                LI      R1,22
                LI      R2,8
                BL      @MYTEST        ; Upon return R0=15, R1=22 and R2=8
                JMP     $              ; Soft halt
MYTEST          INCT    STACK
                MOV     R11,*STACK+    ; Push R11 (return address)
                MOV     R0,*STACK+     ; Push R0
                MOV     R1,*STACK+     ; Push R1
                MOV     R2,*STACK      ; Push R2
                LI      R0,99
                MOV     @MYVAR,R1
                CLR     R2
                B       @POPRG2        ; Pop R2,R1,R0,R11 from stack and return
```

Remarks:

You don't want to do this if you are changing too many registers. In that case a LWPI will be more effective.

POPRX(0-9)

Pop registers & return to caller (R11 already set)

Main Category: RAM

File **spectra_base.a99**
Keywords RAM, LOW-LEVEL

Call format	MYTEST B @POPRX2
Input	-
Output	-
Stack Usage	-

Description:

These routines are similar to the POPRG(0-9) subroutines in that they pop the specified registers from the stack and then returns to the caller.
This version expects that R11 is already set in the subroutine itself.
This is useful if passing parameters to the subroutine via DATA statements

Note that –by default- STACK is an equate for R8.
See the “what I need to know – Stack” section for details on stack usage.

Example:

Suppose you have a subroutine MYTEST defined that changes R0-R2 and you want to make sure that when you call MYTEST upon return R0-R2 have their original values again.

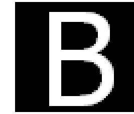
```
MAIN            LI      R0,15
                LI      R1,22
                LI      R2,8
                BL      @MYTEST
                DATA    4711            ; Upon return R0=15, R1=22 and R2=8
                JMP     $               ; Soft halt
MYTEST          INCT    STACK
                MOV     R0,*STACK+      ; Push R0
                MOV     R1,*STACK+      ; Push R1
                MOV     R2,*STACK       ; Push R2
                MOV     *R11+,R0        ; Get 4711 from caller and put in R0
                MOV     @MYVAR,R1
                CLR     R2
                B       @POPRX2         ; Pop R2,R1,R0 from stack and return
```

Remarks:

You don’t want to do this if you are changing too many registers. In that case a LWPI will be more effective.

Memory / Copy

MEMORY/COPY



FILMEM

Fill RAM memory range with byte

Main Category: **RAM**

File **spectra_memcpy.a99**
Keywords RAM, LOW-LEVEL, data-variant

Call format	`MYTEST BL @FILMEM` `DATA P0,P1,P2`
Input	P0 = Memory start address P1 = Memory end address P2 = LSB must contain byte to fill
Output	-
Stack Usage	6 bytes (R0,R1,R2)

Description:

This routine is used to fill a memory range with the specified byte.

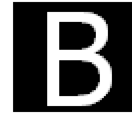
Example:

Fill memory >6000 until >7FFF with value >FF

```
MAIN            BL      @FILMEM
                DATA    >6000,>7FFF,>00FF
                JMP     $               ; Soft halt
```

Remarks:

-



FILMEX
Fill RAM memory range with byte (register variant)

Main Category: **RAM**

File **spectra_memcpy.a99**
Keywords RAM, LOW-LEVEL, register-variant

Call format	`MYTEST    BL  @FILMEM`
Input	R0 = Memory start address R1 = Memory end address R2 = LSB must contain byte to fill
Output	**-**
Stack Usage	

Description:

This routine is used to fill a memory range with the specified byte.
Parameters are passed via registers.

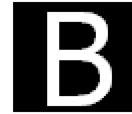
Example:

Fill memory >6000 until >7FFF with value >FF

```
MAIN            LI      R0,>6000        ; Start
                LI      R1,>7FFF        ; End
                LI      R2,>00FF
                BL      @FILMEM
                JMP     $               ; Soft halt
```

Remarks:

NOT YET IMPLEMENTED



CPYM

Copy ROM/RAM memory range

Main Category: **RAM**

File **spectra_memcpy.a99**
Keywords RAM, LOW-LEVEL, data-variant

Call format	`MYTEST BL @CPYM` `DATA P0,P1,P2`
Input	P0 = Memory source address P1 = Memory target address P2 = Number of bytes to copy
Output	**-**
Stack Usage	6 bytes (R0,R1,R2)

Description:

This routine is used to copy a ROM/RAM memory range to RAM.
Parameters are passed via registers.

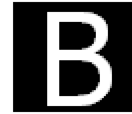
Example:

Copy 500 bytes from high-memory >A000 to low-memory >8000

```
MAIN             BL      @CPYM
                 DATA    >A000,>8000,500
                 JMP     $             ; Soft halt
```

Remarks:

-



CPYMX

Copy ROM/RAM memory range (register variant)

Main Category: RAM

File **spectra_memcpy.a99**
Keywords RAM, LOW-LEVEL, register-variant

Call format	`MYTEST     BL  @CPYMX`
Input	R0 = Memory source address R1 = Memory target address R2 = Number of bytes to copy
Output	**-**
Stack Usage	8 bytes (R0,R1,R2,R11)

Description:

This routine is used to copy a ROM/RAM memory range to RAM.
Parameters are passed via registers.

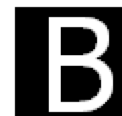
Example:

Copy 500 bytes from high-memory >A000 to low-memory >8000

```
MAIN            LI      R0,>A000    ; Start RAM
                LI      R1,>8000    ; Start VDP RAM
                LI      R2,500
                BL      @CPYM
                JMP     $           ; Soft halt
```

Remarks:

-



G2VDP

Copy GROM memory range to VDP memory

Main Category: **GROM**

File **spectra_memcpy.a99**
Keywords GROM, VDP, LOW-LEVEL, data-variant

Call format	MYTEST BL @G2VDP DATA P0,P1,P2
Input	P0 = GROM source address P1 = VDP target address P2 = Number of bytes to copy
Output	-
Stack Usage	6 bytes (R0,R1,R2)

Description:

This routine is used to copy a memory range from GROM memory directly into VDP memory.

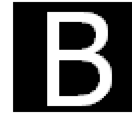
Example:

Copy 300 bytes from GROM >0200 to VDP >0000

```
MAIN            BL      @G2VDP
                DATA    >0200,>0000,300
                JMP     $           ; Soft halt
```

Remarks:

This subroutine overwrites the current GROM address.



G2VDPX

Copy GROM memory range to VDP memory (register variant)

Main Category: **GROM**

File **spectra_memcpy.a99**
Keywords GROM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST     BL  @G2VDPX`
Input	R0 = GROM source address R1 = VDP target address R2 = Number of bytes to copy
Output	-
Stack usage	6 bytes (R0,R1,R2)

Description:

This routine is used to copy a memory range from GROM memory directly into VDP memory. Parameters are passed via registers

Example:

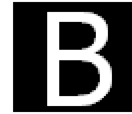
Copy 300 bytes from GROM >0200 to VDP >0000

```
MAIN            LI      R0,>0200     ; GROM start
                CLR     R1           ; VDP start
                LI      R2,300
                BL      @G2VDPX
                JMP     $            ; Soft halt
```

Remarks:

This subroutine overwrites the current GROM address.

MEMORY/COPY



G2MEM

Copy GROM memory range to RAM memory

Main Category: **GROM**

File **spectra_memcpy.a99**
Keywords GROM, RAM, LOW-LEVEL, data-variant

Call format	`MYTEST BL @G2MEM` `DATA P0,P1,P2`
Input	P0 = GROM source address P1 = RAM target address P2 = Number of bytes to copy
Output	-
Stack usage	6 bytes (R0,R1,R2)

Description:

This routine is used to copy a memory range from GROM memory to RAM memory.

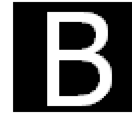
Example:

Copy 300 bytes from GROM >0200 to RAM >6000

```
MAIN            BL      @G2MEM
                DATA    >0200,>6000,300
                JMP     $             ; Soft halt
```

Remarks:

This subroutine overwrites the current GROM address.



G2MEMX

Copy GROM memory range to RAM memory (register variant)

Main Category: **GROM**

File **spectra_memcpy.a99**
Keywords GROM, RAM, LOW-LEVEL, register-variant

Call format	`MYTEST    BL  @G2MEMX`
Input	R0 = GROM source address R1 = RAM target address R2 = Number of bytes to copy
Output	-
Stack usage	6 bytes (R0,R1,R2)

Description:

This routine is used to copy a memory range from GROM memory to RAM memory. Parameters are passed via registers.

Example:

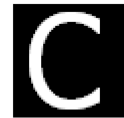
Copy 300 bytes from GROM >0200 to RAM >6000

```
MAIN            LI      R0,>0200     ; GROM start
                LI      R1,>6000     ; RAM start
                LI      R2,300
                BL      @G2MEMX
                JMP     $            ; Soft halt
```

Remarks:

This subroutine overwrites the current GROM address.

VDP
low-level



FVRAM
Fill VDP memory with byte

Main Category: VDP

File **spectra_vdp.a99**
Keywords VDP, LOW-LEVEL, data-variant

Call format	`MYTEST BL @FVRAM` `DATA P0,P1,P2`
Input	P0 = VDP start address P1 = Byte to fill P2 = Number of bytes to fill
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

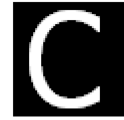
This routine fill the specified VDP range with the specified byte

Example:

Fill memory range VDP >0000 - >0300 with ASCII character 32.
This example clears the screen assuming that (VDP#2 PNT pointer is set to >00) and graphics mode 1 is active.

```
MAIN            BL      @FVRAM
                DATA    >0000,>20,768
                JMP     $           ; Soft halt
```

Remarks:
-



FVRAMX

Fill VDP memory with byte (register variant)

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords VDP, LOW-LEVEL, register-variant

Call format	`MYTEST     BL  @FVRAMX`
Input	R0 = VDP start address R1 = Byte to fill R2 = Number of bytes to fill
Output	**-**
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

This routine fill the specified VDP range with the specified byte
Parameters are passed via registers.

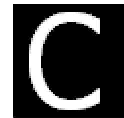
Example:

Fill memory range VDP >0000 - >0300 with ASCII character 32.
This example clears the screen assuming that VDP#2 PNT register is set to >00 and graphics mode 1 is active.

```
MAIN            CLR     R0          ; VDP start
                LI      R1,>20      ; ASCII character 32
                LI      R2,768      ; Bytes to fill
                BL      @FVRAMX
                JMP     $           ; Soft halt
```

Remarks:

-



PVRAM

Copy memory range to VDP memory

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, data-variant

Call format	`MYTEST     BL  @PVRAM` `           DATA  P0,P1,P2`
Input	P0 = VDP start address P1 = RAM/ROM start address P2 = Number of bytes to copy
Output	**-**
Stack usage	6 bytes (R0,R1,R2)

Description:

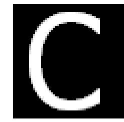
This routine copies the specified memory range (RAM or ROM) to VDP memory.
Same functionality as the VMBW command.

Example:

Copy 15 sprites (16x16 size) from RAM address with label FGHT1 to VDP >0400
This example assumes that VDP#6 SPT (Sprite Pattern Table) register is set to >80 and graphics mode 1 is active.

```
MAIN            BL      @PVRAM
                DATA    >0400,FGHT1,15 * 32
                JMP     $           ; Soft halt
```

Remarks:
There is no PVRAMX routine. You have to use VMBW for this.



VSBR

Read single byte from VDP

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST     BL  @VSBR`
Input	R0 = VDP source address
Output	R1 = MSB, the byte read from VDP
Stack usage	**-**

Description:

This routine reads a single byte from VDP memory and stores it in the Most-Significant Byte of register R1.

Parameters are passed via registers.

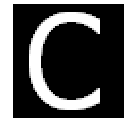
Example:

The below example reads the character displayed on row 0 column 0 from VDP screen table assuming that VDP#2 PNT (Pattern Name Table) register is set to >00
After the routine is executed the Most-Significant Byte of register R1 contains the character value.

```
MAIN            CLR     R0
                BL      @VSBR         ; Upon return MSB of R1 contains character value
                JMP     $             ; Soft halt
```

Remarks:

Same functionality as in the Editor Assembler module.



VSBW

Write single byte to VDP

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST    BL  @VSBW`
Input	R0 = VDP target address R1 = MSB, the byte to write to VDP
Output	-
Stack usage	4 bytes (R0,R11)

Description:

This routine writes a single byte to VDP memory.
The byte to write must be set as Most-Significant Byte of register R1.

Parameters are passed via registers.

Example:

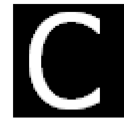
The below example writes the character A (>41) to row 0 column 0 in VDP screen table assuming that VDP#2 PNT (Pattern Name Table) register is set to >00

```
MAIN            CLR     R0              ; VDP start
                LI      R1,>4100        ; Hex value >41 (=ASCII 65) in MSB
                BL      @VSBW
                JMP     $               ; Soft halt
```

Remarks:

Same functionality as in the Editor Assembler module.

VDP LOW-LEVEL



VMBR

Read multiple bytes from VDP

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST     BL  @VMBR`
Input	R0 = VDP source address R1 = RAM target address R2 = Number of bytes to read
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

This routine reads the specified amount of bytes from VDP memory into RAM memory

Parameters are passed via registers.

Example:

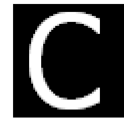
The below example reads 10 characters from VDP screen table starting at row 0 column 0 assuming that VDP#2 PNT (Pattern Name Table) register is set to >00.
The bytes read will be stored in RAM memory in the memory buffer labelled MYBUFF.

```
MAIN            CLR    R0            ; VDP start
                LI     R1,MYBUFF     ; buffer in RAM
                LI     R2,10         ; amount of bytes to read
                BL     @VMBR
                JMP    $             ; Soft halt
MYBUFF          BSS    10            ; Buffer in RAM for storing 10 chars
```

Remarks:

Same functionality as in the Editor Assembler module.

VDP LOW-LEVEL



VMBW

Write multiple bytes to VDP

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST BL @VMBW`
Input	R0 = VDP target address R1 = RAM/ROM source address R2 = Number of bytes to write
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

This routine writes the specified amount of bytes from RAM/ROM memory to VDP memory.

Parameters are passed via registers.

Example:

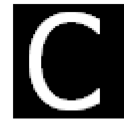
The below example writes 11 characters from RAM address with label MYTXT to VDP screen table starting at row 0 column 0 assuming that VDP#2 PNT (Pattern Name Table) register is set to >00.

```
MAIN        CLR    R0            ; VDP start
            LI     R1,MYTXT      ; Text in RAM/ROM
            LI     R2,11         ; Number of bytes to write
            BL     @VMBW
            JMP    $             ; Soft halt
MYTXT       TEXT   "HELLO WORLD"
```

Remarks:

Same functionality as in the Editor Assembler module.

VDP LOW-LEVEL



VWTR

Write to VDP register

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST     BL  @VWTR`
Input	R0 = MSB is the VDP target register LSB is the value to write
Output	-
Stack usage	4 bytes (R0,R11)

Description:

This routine writes the value in the least-significant byte of R0 to the VDP write-only register addressed by the most-significant byte of R0.

Parameters are passed via registers.

Example:

The below example sets the SPT (Sprite Pattern Table) pointer in VDP#6 to address >3000.

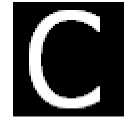
```
MAIN            LI      R0,>0606        ; Note that >06 * >800 = >3000
                MOV     R0,@VDPR6       ; Sync shadow register
                BL      @VWTR
                JMP     $               ; Soft halt
```

Remarks:

Same functionality as in the Editor Assembler module.

The SPECTRA library knows the concept of VDP shadow registers in RAM.
If you use VWTR you need to make sure the shadow registers are up-to-date as well.

For details check out functions LVDPSH (Load VDP shadow registers) and WVDPSH (Write shadow registers to VDP).



LVDPSH

Load VDP shadow registers in RAM with video mode table

Main Category: **VDP**

File	**spectra_vdp.a99**
Keywords	RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST BL @LVDPSH`
Input	R0 = Address of video mode table
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

The SPECTRA library knows the concept of VDP shadow registers in RAM.
They are basically a copy of the write-only VDP registers.
This concept allows some more flexibility (e.g. read register value or set certain bits in the copy).

The LVDPSH subroutine is used to load all shadow registers (@VDPREG0-@VDPREG7) with the values of the specified video mode table.

Additionally it writes the amount of columns in a row (32, 40, 64) to the memory location @VDPCOL and clears the shadow status register (@VDPSTA).

Note that LVDPSH itself does not dump the registers to the VDP. You have to use the WVDPSH subroutine for that.

See section **"APPENDIX – Overview video mode tables"** for the default video mode tables provided with SPECTRA.

See section **"BASE – What I need to know"** for details about SPECTRA memory layout.

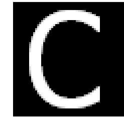
Example:

The below example sets up the VDP for dealing with graphics mode 1. The table VMODE1 is already provided as part of the SPECTRA library (spectra_vdp.a99) and is included in this example for reference purposes only.

```
MAIN            LI      R0,VMODE1   ; "graphics mode 1"
                BL      @LVDPSH     ; Load shadow registers

                ....

                BL      @WVDPSH     ; Write shadow registers to VDP
                JMP     $           ; Soft halt

VMODE1          DATA  >0000,>01E2,>0200,>030E,>0401,>0506,>0680,>0700,32
***************************************************************
* 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=0: M1 \ Graphics 1 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 Background color to black
*
* 32 Columns in a row
***************************************************************
```

Remarks:

-



WVDPSH

Write VDP shadow registers from RAM to VDP write-only registers

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST BL @LVDPSH`
Input	R0 = Address of video mode table
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

The SPECTRA library knows the concept of VDP shadow registers in RAM.
They are basically a copy of the write-only VDP registers. This concept allows some more flexibility (e.g. read register value or set certain bits in the copy).

The WVDPSH subroutine itself is used to dump all shadow registers (@VDPREG0-@VDPREG7) to the VDP write-only registers after they have been loaded from a video mode table by the LVDPSH subroutine.

See section **"APPENDIX – Overview video mode tables"** for the default video mode tables provided with SPECTRA.

See section **"BASE – What I need to know"** for details about SPECTRA memory layout.

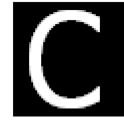
Example:

The below example sets up the VDP for dealing with graphics mode 1.
The table VMODE1 is provided as part of the SPECTRA library.

```
MAIN            LI      R0,VMODE1   ; "graphics mode 1"
                BL      @LVDPSH     ; Load shadow registers
                ....
                BL      @WVDPSH     ; Write shadow registers to VDP
                JMP     $           ; Soft halt

VMODE1          DATA   >0000,>01E2,>0200,>030E,>0401,>0506,>0680,>0700,32
```

Remarks:

-



VDPADR

Calculate VDP table start address

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST BL @VDPADR` `DATA P0`
Input	P0 = Table identifier
Output	@OUTP0 = VDP address of table
Stack usage	10 bytes (R0,R1,R2,R3,R11)

Description:

This subroutine calculates the start address of the specified table based on the value stored in the corresponding VDP shadow register and returns it in the memory location OUTP0.

Please refer to subroutines LVDPSH and WVDPSH for details about what shadow registers are.

Below are the valid values (equates) available for input register R0:

GETPNT (Pattern name table)
GETPCT (Pattern color table)
GETPDT (Pattern descriptor table)
GETSAT (Sprite attribute table)
GETSPT (Sprite pattern table)

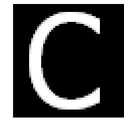
Example:

The below example determines the start address of the Pattern descriptor table and then copies 10 character patterns to that table, starting with character 0

```
MAIN        BL      @VDPADR      ; Get start address of Pattern descriptor table
            DATA    GETPDT       ; Result is in @OUTP0
            MOV     @OUTP0,R0
            LI      R1,DIGITS
            LI      R2,10*8      ; For displaying score
            BL      @VMBW
            JMP     $            ; Soft halt
DIGITS      BYTE    >00,>1C,>22,>63,>63,>63,>22,>1C        ; 0
            BYTE    >00,>18,>38,>18,>18,>18,>18,>7E        ; 1
            ...
```

Remarks:

-



VIDOFF

Disable screen display

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST     BL  @VIDOFF`
Input	-
Output	-
Stack usage	-

Description:

This subroutine sets bit 1 in VDP shadow register #1 to 0 which means that the VDP stops displaying the screen image and opens a permanent window (interval) for CPU access.

Note that VIDOFF only sets the VDP shadow register, you still need to write it to the VDP using the WVDPSH subroutine or a similar command.

You normally use this command at the start of a game if a new screen is to be displayed once it is completely built.

Example:

The below example loads video mode1, disables the screen display and writes the shadow registers to VDP. Then you have a permanent window for accessing VDP memory. Finally the screen display is enabled again by VIDON and with writing all shadow registers to the VDP.

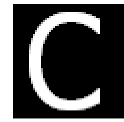
```
MAIN            LI      R0,VMODE1   ; "graphics mode 1"
                BL      @LVDPSH     ; Load shadow registers
                BL      @VIDOFF     ; Disable screen display
                BL      @WVDPSH     ; Write shadow registers to VDP

                ....                ; Your VDP commands here

                BL      @VIDON      ; Enable screen display
                BL      @WVDPSH     ; Write shadow registers to VDP
                JMP     $           ; Soft halt
```

Remarks:

In performance crucial code, you can achieve the same by doing an inline:

```
                SZCB    @BIT1,@VDPR1 ; bit 1=0 (Disable screen display)
```

VDP LOW-LEVEL



VIDON

Enable screen display

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	`MYTEST BL @VIDON`
Input	-
Output	-
Stack usage	-

Description:

This subroutine sets bit 1 in VDP shadow register #1 to 1 which means that the VDP starts displaying the screen image again. This closes the permanent window (interval) that was available for CPU access (see VIDOF subroutine)

Note that VIDON only sets the VDP shadow register, you still need to write it to the VDP using the WVDPSH subroutine or a similar command.

You normally use this command at the start of a game if a new screen is to be displayed once it is completely built.

Example:

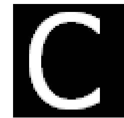
The below example loads video mode1, disables the screen display and writes shadow registers to VDP. Then you have an unlimited window for doing all your VDP stuff . Finally the screen display is enabled again by VIDON and with writing all shadow registers to the VDP.

```
MAIN            LI     R0,VMODE1   ; "graphics mode 1"
                BL     @LVDPSH     ; Load shadow registers
                BL     @VIDOFF     ; Disable screen display
                BL     @WVDPSH     ; Write shadow registers to VDP

                ....               ; Your VDP commands here

                BL     @VIDON      ; Enable screen display
                BL     @WVDPSH     ; Write shadow registers to VDP
                JMP    $           ; Soft halt
```

Remarks:

In performance crucial code, you can achieve the same by doing an inline:

```
                SOCB   @BIT1,@VDPR1 ; bit 1=1 (Enable screen display)
```



XY2OF

Calculate screen offset of X/Y character position

Main Category: **VDP**

File **spectra_vdp.a99**
Keywords RAM, VDP, LOW-LEVEL, register-variant

Call format	MYTEST BL @XY2OF
Input	R0 = X Column (0-31, 0-39 for text-mode) R1 = Y row (0-23) or R0 =MSB X Column (0-31, 0-39 for text-mode) LSB Y row (0-23)
Output	@OUTP0 = Calculated screen offset
Stack usage	10 bytes (R0,R1,R2,R3,R11)

Description:

This subroutine calculates the VDP screen offset based on the provided XY coordinates by using the formula **offset = (ROWS * @VDPCOL + COL)**

Note that the memory location @VDPCOL holds the numbers of columns in a row and is normally set by the @LVDPSH subroutine.

Be aware that you must add the PNT base address to the offset yourself to get the proper VDP target address.

Example:

The below example puts the text 'HELLO WORLD' at row 4, column 11 assuming that the Pattern Name Table (PNT) is located at address >0000.

```
MAIN        LI      R0,>0A03    ; X=11 Y=4
            BL      @XY2OF      ; Result is in @OUTP0
            MOV     @OUTP0,R0
            LI      R1,HELLO
            LI      R2,11       ; Text length
            BL      @VMBW
            JMP     $           ; Soft halt
HELLO       TEXT    'HELLO WORLD'
```

Remarks:

-

VDP
Sprites

VDP SPRITES – MEMORY SETUP

What I need to know

Below you find some information that may be helpful when using sprites in SPECTRA.

Copy of Sprite Attribute Table in RAM (Shadow SAT)

Before using the SPECTRA sprite functionality, you need to allocate 128 bytes of RAM. This is required for maintaining a work copy of the Sprite Attribute Table (SAT). Basically it allows you to easily do sprite manipulation in memory and then dump the SAT to the VDP with a single call (PUTSAT).

Label	RAMSAT
Size	128 bytes
Remarks	Allocate in scratch-pad for optimal speed

For most sprite handling subroutines you can specify the address of the shadow SAT. For easy sprite manipulation it is advised to define the shadow SAT with the label RAMSAT. It will allow you to use the pre-defined sprite equates (See reference table).

It is advised to store the shadow SAT in scratch-pad RAM for getting optimal speed.

Example 1:

In the main program you define that SPECTRA will be using 128 bytes of scratch-pad memory for maintaining a work copy of the Sprite Attribute Table.

```
RAMSAT          EQU    >8340       ; 128 bytes in scratch-pad RAM
```

Example 2:

If scratch-pad memory is not available for use, you would need to allocate it in normal memory, e.g. high-memory

```
RAMSAT          BSS    128         ; 128 bytes in RAM
```

Remarks:

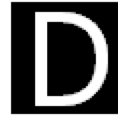
SPECTRA has a set of equates (SPR1-SPR32) that allows easy sprite manipulation in memory.

```
SPR1    EQU   RAMSAT                  ; Sprite 1
SPR2    EQU   RAMSAT+4                ; Sprite 2
SPR3    EQU   RAMSAT+8                ; Sprite 3
SPR4    EQU   RAMSAT+12               ; Sprite 4
SPR5    EQU   RAMSAT+16               ; Sprite 5
SPR6    EQU   RAMSAT+20               ; Sprite 6
SPR7    EQU   RAMSAT+24               ; Sprite 7
SPR8    EQU   RAMSAT+28               ; Sprite 8
......
SPR29   EQU   RAMSAT+112              ; Sprite 29
SPR30   EQU   RAMSAT+116              ; Sprite 30
SPR31   EQU   RAMSAT+120              ; Sprite 31
SPR32   EQU   RAMSAT+124              ; Sprite 32
```

Format of Sprite Attribute Table Entry:

BYTE 0	BYTE 1	BYTE 2	BYTE 3
Vertical position	Horizontal position	Sprite name pointer	Color & Early clock flag

VDP SPRITES



PUTSAT

Write shadow SAT (Sprite Attribute Table) from RAM to VDP memory

Main Category: **VDP**

File **spectra_sprites.a99**
Keywords RAM, VDP, LOW-LEVEL, SPRITE, register-variant

Call format	MYTEST BL @PUTSAT
Input	R0=VDP target address or >FFFF if address must be calculated R1=Address of shadow SAT in RAM/ROM R2=Address of sprite order table in RAM/ROM or >FFFF for default 0..31 order
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

For increased speed and more easy sprite manipulation, SPECTRA can use one or more shadow SAT tables in memory. The goal is to minimize the amount of read & writes from/to the VDP by first doing the appropriate manipulation in memory and then doing a block write to VDP.

You have to set R0 with the SAT address in VDP or you let the system calculate the address by setting R0 to >FFFF. In that case the VDP SAT address is automatically calculated, based on the value in VDP shadow register VDPR5.

SPECTRA expects you to define a shadow SAT (128 bytes) in RAM. It is advised that this memory gets the label RAMSAT. So normally you would do a “LI R1,RAMSAT” first.

If R2 is set to >FFFF then the SAT will be written using the default sprite order 0..31
If you want to use a rotating custom sprite order table (32 bytes), e.g. for avoiding invisible 5th sprite, you can do so by setting R2 to the address of that table.

Example:

The next example, assumes that you have a game cartridge with a SAT dumped in ROM at address >6200. The example basically does the following steps:

- Setup graphics mode 1
- Load shadow SAT in RAM from SAT stored in a game ROM.
- Put sprite 5 at Y=100,X=30
- Dump shadow SAT to VDP using the default sprite order 0..31

```
RAMSAT          BSS     128
MAIN            LI      R0,VMODE1           ; "graphics mode 1"
                BL      @LVDPSH             ; Load shadow registers
                BL      @WVDPSH             ; Dump it to the VDP
                BL      @CPYM
                DATA    >6200,RAMSAT,32*4   ; Put sprites from ROM in shadow SAT
                LI      R0,>641E
                MOV     R0,@SPR5            ; Set Y=100, X=30
                SETO    R0                  ; Let SPECTRA calculate VDP SAT address
                LI      R1,RAMSAT
                SETO    R1                  ; >FFFF - No sprite order table
                BL      @PUTSAT
                JMP     $                   ; Soft halt
```

Remarks:

There is no custom sprite order table provided with SPECTRA. You have to set it up yourself.

VDP SPRITES

SPRORD

Initialize sprite order table to default sprite order 0..31

Main Category: **VDP**

File **spectra_sprites.a99**
Keywords RAM, VDP, SPRITE, register-variant

Call format	`MYTEST     BL  @SPRORD`
Input	R0=Address of sprite order table in RAM
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

When writing the shadow SAT to VDP memory using the PUTSAT subroutine one can set the order in which the entries are written. This allows you to for example have a rotating sprite list for avoiding an invisible 5th sprite, however this comes with the cost of "flickering".

There is no sprite order table provided with SPECTRA, you have to set it up yourself (32 bytes).

The SPRORD subroutine fills the sprite order table with the values 0..31 which in other words is the default sprite order

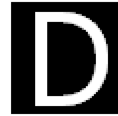
Example:

The below example assumes that shadow SAT RAMSAT is already is setup and that it needs to be written to VDP with default sprite order 0..31, but this time with possibility of having rotating sprite order.

```
VDPSAT          EQU     >0300
MAIN            LI      R0,MYORDR
                BL      @SPRORD     ; Setup Sprite Order Table with values 0..31
                LI      R0,VDPSAT
                LI      R1,RAMSAT
                LI      R2,MYORDR   ; Custom Sprite Order Table
                BL      @PUTSAT
                JMP     $           ; Soft halt
MYORDR          BSS     32          ; Custom sprite order
```

Remarks:

There is no custom sprite order table provided with SPECTRA. You have to set it up yourself .



S8X8

Sprites with 8 x 8 pattern

Main Category: **VDP**

File	**spectra_sprites.a99**
Keywords	RAM, VDP, SPRITE

Call format	`MYTEST    BL  @S8X8`
Input	-
Output	-
Stack usage	-

Description:

This subroutine resets bit 6 in VDP shadow register #1 to 0 which means that the sprite size is 8 x 8. In other words you need 8 bytes for defining a sprite pattern.

Note that S8X8 only set the VDP shadow register, you still need to write it to the VDP using the WVDPSH subroutine or a similar command.

Example:

The below example loads video mode1, set sprite size to 8x8 and sets all VDP read-only registers.

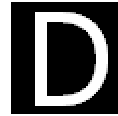
```
MAIN            LI      R0,VMODE1   ; "graphics mode 1"
                BL      @LVDPSH     ; Load shadow registers
                BL      @8X8        ; Set sprite size 8x8
                BL      @WVDPSH     ; Write shadow registers to VDP
                JMP     $           ; Soft halt
```

Remarks:

In performance crucial code, you can achieve the same by doing an inline:

```
                SZCB    @PLS2+1,@VDPR1 ; bit 6=0 (Sprite size 8x8)
                BL      @WVDPSH        ; Write shadow registers to VDP
```

VDP SPRITES



S16X16

Sprites with 16 x 16 pattern

Main Category: **VDP**

File **spectra_sprites.a99**
Keywords RAM, VDP, SPRITE

Call format	`MYTEST     BL  @S16X16`
Input	-
Output	-
Stack usage	-

Description:

This subroutine sets bit 6 in VDP shadow register #1 to 1 which means that the sprite size is 16 x 16. In other words you need 32 bytes for defining a sprite pattern.

Note that S16X16 only sets the VDP shadow register, you still need to write it to the VDP using the WVDPSH subroutine or a similar command.

Example:

The below example loads video mode1, set sprite size to 16x16 and sets all VDP write-only registers.

```
MAIN            LI      R0,VMODE1  ; "graphics mode 1"
                BL      @LVDPSH    ; Load shadow registers
                BL      @16X16     ; Set sprite size 16X16
                BL      @WVDPSH    ; Write shadow registers to VDP
                JMP     $          ; Soft halt
```

Remarks:

In performance crucial code, you can achieve the same by only dumping VDP shadow register 1 after doing an inline:

```
                SOCB    @PLS2+1,@VDPR1 ; bit 6=1 (Sprite size 16x16)
```

SMAG1X

Sprite magnification 1X

Main Category: VDP

File **spectra_sprites.a99**
Keywords RAM, VDP, SPRITE

Call format	`MYTEST     BL  @SMAG1X`
Input	-
Output	-
Stack usage	-

Description:

This subroutine resets bit 7 in VDP shadow register #1 to 0 which means that the sprite magnification is 1. In other words sprites are not magnified..

Note that SMAG1X only sets the VDP shadow register, you still need to write it to the VDP using the WVDPSH subroutine or a similar command.

Example:

The below example loads video mode1, set sprite magnification to 1X and sets all VDP write-only registers.

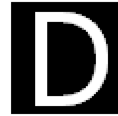
```
MAIN            LI      R0,VMODE1   ; "graphics mode 1"
                BL      @LVDPSH     ; Load shadow registers
                BL      @SMAG1X     ; Set sprite magnification 1X
                BL      @WVDPSH     ; Write shadow registers to VDP
                JMP     $           ; Soft halt
```

Remarks:

In performance crucial code, you can achieve the same by only dumping VDP shadow register 1 after doing an inline:

```
                SZCB    @BIT7,@VDPR1 ; bit 7=0 (Sprite magnification 1x)
```

VDP SPRITES



SMAG2X
Sprite magnification 2X

Main Category: **VDP**

File **spectra_sprites.a99**
Keywords RAM, VDP, SPRITE

Call format	MYTEST BL @SMAG2X
Input	-
Output	-
Stack usage	-

Description:

This subroutine sets bit 7 in VDP shadow register #1 to 1 which means that the sprite magnification is 2x. In other words sprites are twice the size they would normally be

Note that SMAG2X only sets the VDP shadow register, you still need to write it to the VDP using the WVDPSH subroutine or a similar command.

Example:

The below example loads video mode1, set sprite magnification to 2X and sets all VDP write-only registers.

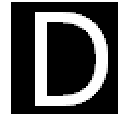
```
MAIN            LI      R0,VMODE1   ; "graphics mode 1"
                BL      @LVDPSH     ; Load shadow registers
                BL      @SMAG2X     ; Set sprite magnification 2X
                BL      @WVDPSH     ; Write shadow registers to VDP
                JMP     $           ; Soft halt
```

Remarks:

In performance crucial code, you can achieve the same by only dumping VDP shadow register 1 after doing an inline:

```
                SOCB    @BIT7,@VDPR1 ; bit 7=1 (Sprite magnification 2x)
```

VDP SPRITES



SPRITE

Create new sprite

Main Category: **VDP**

File **spectra_sprites.a99**
Keywords RAM, VDP, SPRITE

Call format	`MYTEST    BL  @SPRITE`
Input	R0=Pointer to sprite table entry in ROM/RAM -- Format for sprite table entry: DATA P0,P1,P2,P3,P4 P0=VDP target address for entry in VDP SAT or shadow SAT P1=MSB is Y pixel position LSB is X pixel position P2=MSB is sprite character code LSB is sprite colour P3=ROM/RAM source address of sprite pattern data P4=Address of subroutine to call via BL @statement.
Output	-
Stack usage	????

Description:

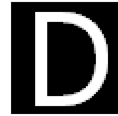
This subroutine is used to create a new sprite.

MSB (Most Significant Byte) of P1 is the Y pixel position
LSB (Least Significant Byte) of P2 is the X pixel position

MSB of P2 is the sprite character code
LSB of P2 is the sprite colour.

P3 is a pointer to the memory location that holds the sprite pattern data.
Set P3 to >0000 (equate NULL) if no pattern data is to be defined.

You can set **P4** with the address of the subroutine that initializes the control logic for this particular sprite. Normally this subroutine is then used to do things like variable initialisation, set sprite order & create a timer slot using (BL @MKSLOT) for handling the sprite animation (movement, etc.).
Set P4 to >0000 (equate NULL) if no setup subroutine needs to be called.



Example:

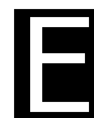
Assuming that location SCORP1 holds the pattern data for a scorpion, the below example will load the pattern data (character >A4) and put a white scorpion sprite on screen at Y position >A2 and X position >80. It then calls the subroutine MYSUB using (BL @MYSUB)

```
MAIN            LI      R0,ABC      ; Sprite table entry
                BL      @SPRITE     ; Put sprite on screen
                JMP     $           ; Soft halt

MYSUB           CLR     @BLABLA     ; Do something
                RET                 ; Exit

ABC             DATA    VDPSAT+40,>A280,>A40F,SCORP1,MYSUB
```

Remarks:

-

VDP
Tiles & Patterns



FILLSCR

Fill screen with character

Main Category: **VDP**

File **spectra_tiles.a99**
Keywords RAM, VDP, TILES, PATTERNS

Call format	MYTEST BL @FILLSCR
Input	R0=Character to fill
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

Fill the screen with the character specified in R0.

The subroutine uses the VDPADR subroutine to determine the address of the PNT table.
Uses memory location @VDPCOL (columns in a row) which is set by the LVDPSH subroutine.
In other words, it also works in text mode, etc.

Example:

Fill the screen with letter A.

```
MAIN            LI     R0,65        ; ASCII of letter A
                BL     @FILLSCR     ; Fill screen
                JMP    $            ; Soft halt
```

Remarks:

-

FIBOX

Fill rectangular area with character

Main Category: **VDP**

File **spectra_tiles.a99**
Keywords RAM, VDP, TILES, PATTERNS

Call format	`MYTEST BL @FIBOX` `DATA P0,P1,P2,P3,P4`
Input	P0 = Upper left corner X P1 = Upper left corner Y P2 = Width P3 = Height P4 = Character to fill
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

Fill the rectangular screen area (box) with the character specified in P4.

The subroutine uses the VDPADR subroutine to determine the address of the PNT table.
Uses memory location @VDPCOL (columns in a row) which is set by the LVDPSH subroutine.
In other words, it also works in text mode, etc.

A right screen-column boundary check is done to prevent that the box spans the right and left part of the screen.

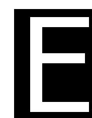
Example:

Fill a box with the letter ‘A’. The box is 15 columns wide and 4 rows long and is positioned at row 8 column 10.

```
MAIN            BL      @FIBOX
                DATA    10,8,15,4,65 ; Fill box with letter A
                JMP     $            ; Soft-halt
```

Remarks:

-



FIBOXX

Fill rectangular area with character (register variant)

Main Category: **VDP**

File **spectra_tiles.a99**
Keywords RAM, VDP, TILES, PATTERNS

Call format	`MYTEST    BL  @FIBOXX`
Input	R0 = Upper left corner X R1 = Upper left corner Y R2 = Width R3 = Height R4 = Character to fill
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

Fill the rectangular screen area (box) with the character specified in P4.
Parameters are passed via registers.

The subroutine uses the VDPADR subroutine to determine the address of the PNT table.
Uses memory location @VDPCOL (columns in a row) which is set by the LVDPSH subroutine.
In other words, it also works in text mode, etc.

A right screen-column boundary check is done to prevent that the box spans the right and left part of the screen.

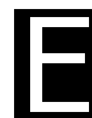
Example:

Fill a box with the letter ‘A’. The box is 15 columns wide and 4 rows long and is positioned at row 8 column 10.

```
MAIN            LI      R0,10       ; left corner X
                LI      R1,8        ; left corner Y
                LI      R2,15       ; width
                LI      R3,4        ; height
                LI      R4,65       ; letter ‘A’
                BL      @FIBOXX     ; Fill box with letter A
                JMP     $           ; Soft-halt
```

Remarks:

-



PUTTX

Put length-byte prefixed string on screen

Main Category: VDP

File	**spectra_tiles.a99**
Keywords	RAM, VDP, TILES, PATTERNS

Call format	`MYTEST BL @PUTTX`
	`DATA P0,P1,P2`
	---------------- 	
	`MYTEST BL @PUTTX`
	`DATA P0,P1`

Input	P0 = X column (0-31, 0-39, ...) P1 = Y column (0-23) P2 = Pointer to string ---------------- P0 = MSB (X column 0-31, 0-39, ...) LSB (Y column 0-23) P1 = Pointer to string
Output	-
Stack usage	8 bytes (R0,R1,R2,R3)

Description:

Display string specified in P2 on screen at column P0 and row P1.
Display string specified in P1 on screen at column (HI-byte P0) and row (LO-byte P0).

The first byte of the string to display must contain the string length.

The subroutine uses the VDPADR subroutine so it knows where the PNT table is in VDP memory. No need to add offset.

Example:

Assuming that the video mode table is already setup, this example displays the text HELLO WORLD on row 12, column 10.

```
MAIN            BL      @PUTTX
                DATA    >0A0C,TXT1  ; X=10 Y=12
                JMP     $           ; Soft-halt
TXT1            BYTE    11
                TEXT    'HELLO WORLD'
```

Remarks:

-

MIRRV

Mirror tile/sprite patterns in RAM memory buffer around vertical axis

Main Category: **VDP**

File **spectra_tiles.a99**
Keywords RAM, VDP, TILES, PATTERNS

Call format	`MYTEST BL @MIRRV` `DATA P0,P1,P2`
Input	P0 = Pointer to pattern data in RAM P1 = Number of bytes to mirror P2 = 'SWAP' or 'NOSWAP'
Output	-
Stack usage	16 bytes (R0,R1,R2,R3,R4,R5,R6,R7)

Description:

Mirror tile/sprite patterns in RAM around the vertical axis.

P0 is a pointer to the memory location in RAM that is holding the actual pattern data.

P1 is the number of bytes that need to be mirrored.

P2 should be set to the pre-defined equates SWAP or NOSWAP.
SWAP means that the pattern sequence 0123 will be swapped to 2301.
This is useful for mirroring sprites having 16 x 16 pattern.

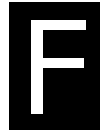
Note that this subroutine overwrites the original patterns while mirroring.

Example:

Assuming that the video mode table is already setup, this example mirrors the sprite patterns of a 16x16 space ship (SHIP) around the vertical axis. In this example the space ship is made out of 3 overlapping sprites (one 16x16 sprite for each colour, 3 * 32 bytes = 96 bytes).
Finally the sprite pattern data gets dumped to the VDP by the @PVRAM subroutine.

```
MAIN            BL      @CPYM
                DATA    SHIP,RAMBUF,96          ; Copy sprite patterns to RAM
                BL      @MIRRV
                DATA    RAMBUF,96,SWAP          ; Mirror patterns around vertical axis
                BL      @PVRAM
                DATA    >0400,RAMBUF,96         ; Dump the pattern data to the VDP
                JMP     $                       ; Soft-halt
RAMBUF          BSS     96                      ; Work buffer

SHIP            BYTE  >00,>01,>01,>01,>01,>01,>00,>00        ; RED
                BYTE  >00,>00,>08,>19,>3D,>01,>01,>00
                BYTE  >00,>00,>00,>80,>80,>80,>C0,>C0
                BYTE  >40,>40,>F0,>B8,>8C,>80,>00,>00
                ....                                         ; GREEN
                ....                                         ; WHITE
```

Sound & Speech



EPSGMD

Setup memory for playing EPSGMOD tune

Main Category: **SOUND**

File **spectra_epsgmod.a99**
Keywords RAM, SOUND, EPSGMOD

Call format	`MYTEST     BL  @EPSGMD`
Input	R0 = Address of tune
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

This subroutine is the interface to the built-in EPSGMOD player from Tursi (http://www.harmlesslion.com). The version included in SPECTRA is slightly modified so that the player runs as a task (PSGTCK) along with your game tasks. Other than that no modifications were done.

Note that the actual player requires 128 bytes of RAM which are allocated in spectra_epsgmod.a99

Basically the idea is that you create the tune using Kontechs Mod2PSG2 (http://mod2psg2.kontechs.de). Mod2PSG2 is a very powerful music tracker for the SN76489 sound chip that is used in the TI-99/4A, Colecovision, SEGA Master System, …
This windows software is NOT included in SPECTRA, but is available for free via the mentioned web link.

After exporting the tune from Mod2PSG2, you can use the included PERL conversion utility for generating the required BYTE source statements you then include in your source code.
PERL itself is a well-known open source script programming language that you can download from http://www.perl.org

Don't forget to start the PSGTCK controller that is responsible for playing the tune (see example for details).

Example:

The below example is taken from Time Pilot, my new homebrew game for the TI-99/4A.
It assumes that the byte data with label SND1 was already exported from the MOD2PSG2 Tracker and converted using the little PERL conversion utility.

```
MAIN            ....
                MOV    @PLS2,@THIGH ; Highest slot in use
                BL     @MKSLOT
                DATA   0,2,KBSCAN   ; Controller 0 - Keyboard
                BL     @MKSLOT
                DATA   1,1,PSGTCK   ; Controller 1 - Play PSG sound
                LI     R0,TUNE
                BL     @EPSGMD      ; Setup memory for playing tune
                B      @TMGR        ; Start Task Scheduler
***************************************************************
* Game introduction tune
********@*****@************************************************
SND1    BYTE >0D,>00,>FF,>00,>08,>80,>00,>08,>80,>00,>08,>80
        BYTE >00,>08,>80,>00,>FF,>80,>FF,>00,>08,>80,>00,>08
        BYTE >FF,>80,>FF,>00,>08,>80,>00,>08,>80,>00,>08,>80
         ...
```

Conversion utility:

You can use the below PERL script for converting the exported tune from the Mod2PSG2 tracker into assembly BYTE statements:

```
open (FH,"<","your_file.epsgmod")  || die("Couldn't open file!\n");
binmode(FH)                        || die("Don't know but something went wrong\n");

my $cnt   = 0;
my $label = '       BYTE ';
my $data  = '';
while (read(FH,$buf,2)) {
   if ($cnt % 6 == 0) {
      $data .= "\n$label";
   } else {
      $data .= ",";
   }
   my $val = sprintf("%04X", unpack("n",$buf));
   $data .= ">" . substr($val,0,2) . ",";
   $data .= ">" . substr($val,2,2);
   #printf ("%X",unpack("n",$buf));
   $cnt++;
};
print "RUNSIZ  EQU ", $cnt, "\n";
print "RUN $data\n";
close(FH);
```

Remarks:

See the license section for the conditions on using the EPSGMOD player in your game.
Also big thanks to Tursi and Kontechs for their permission on including the player.

Note that an optimized version of the EPSGMOD player and a decent conversion utility is planned for the next release of SPECTRA.

Timers

TIMERS

What I need to know

Below you find some information that may be helpful when creating tasks in SPECTRA.

Timer table

The timer table is used by the Timer Manager (TMGR) for executing subroutines at a specified interval. It is the game programmers' responsibility to allocate the amount of required memory and to assign the label TIMER to it.

A timer table consists of 1 to X timer slots. See Timer slot format for further details. Make sure to set the SPECTRA memory location @**THIGH** to the number of the highest slot in use.

Label	TIMER
Size	Depends on amount of slots to use. 1 slot requires 8 bytes.
Remarks	THIGH must contain the number of the highest slot in use

Example:

Allocate 20 timer slots in your main program.

```
TIMER           BSS     160         ; Space for 20 timer slots
MAIN            LI      R0,19
                MOV     R0,@THIGH   ; Highest slot in use
```

Timer slot format

A timer slot consists of 8 bytes (4 words) and the initial setup is normally done by using the MKSLOT subroutine. However manual manipulation is also easily possible and often useful when changing certain slot aspects on-the-fly.

BYTE 0-1	BYTE 2-3	BYTE 4-5	BYTE 6-7
Interval (Target tick count)	Controller (address of routine to call)	Internal tick counter	Private area (available for personal use)

Interval Determines at what interval the slot should be fired. This interval must be specified in ticks per second.

On an American TI-99/4A console there are 60 ticks per second.
On an European TI-99/4A console there are 50 ticks per second.

Controller This is the address of the subroutine that will be called by the Timer Manager when the slot is fired.
In SPECTRA such subroutine is called a “controller”.
The Timer Manager calls the controller using the BL @xxx statement.

Internal counter Is an internal counter used by the timer manager to keep track about when the slot should be fired. Do not modify.

Private Area This area is available for personal use. You would normally use this area to store some flags related to your controller, as a pointer to a memory area you allocate, etc.

Highest slot in use

The timer manager TMGR must know how many slots it needs to handle. Therefore the SPECTRA library has a memory location @**THIGH** that must contain the number of the highest slot in use.
See section “SPECTRA – memory layout” for the exact location of @THIGH.

Example:

Set highest slot in use to 4 (which is the 5th slot)

```
MAIN            LI     R0,4
                MOV    R0,@THIGH   ; Highest slot in use
```

Equates for accessing timer slots

For doing easy slot manipulation SPECTRA delivers a set of equates in **spectra_timers.a99** for accessing 20 timer slots:

TSLOTx Byte 0-1 : Timer interval (byte 0-1) of specified slot
TSUBx Byte 2-3 : Controller address of specified slot
TPRVx Byte 6-7 : Private area of specified slot

* **x**=number between 0 and 19

Example:

Assuming you have a controller ZBLNK that runs in slot 10, the below example will use the private area of that slot to store a game counter, cycling from 0-3.

```
ZBLNK           INC    @TPRV10      ; Increase counter (private variable slot 10)
                C      @TPRV10,@PLS3
                JNE    ZBLNKZ
                CLR   @TPRV10        ; Reset counter (private variable slot 10)
ZBLNKZ          RET
```

TIMERS

MKSLOT

Allocate specified timer slot

Main Category: **TIMERS**

File **spectra_timers.a99**
Keywords RAM, TIMERS, ANIMATIONS

Call format	`MYTEST BL @MKSLOT` `DATA P0,P1,P2`
Input	P0 = Slot number P1 = Run slot every P1 ticks P2 = Controller (subroutine to call via BL @P2 when slot is fired)
Output	-
Stack usage	8 bytes (R0,R1,R2,R11)

Description:

This subroutine is used for allocating the specified timer slot.

P0 is the slot number to use. The amount of available slots is determined by the size of the slot table in RAM memory. See section "Timer layout" and the timer manager subroutine TMGR for details.

P1 determines the interval at which the task scheduler should run the subroutine specified in parameter P2. The value for the interval is defined in ticks per second.

On an American TI-99/4A console there are 60 ticks per second. On an European TI-99/4A console there are 50 ticks per second.

P2 contains the address of the subroutine to call via BL @P2 when the slot is fired.
We call this type of subroutine a "controller".

Note that the MKSLOT subroutine only allocates the timer, it does not process it.
Processing of all timer slots is done by the timer manager, so you have to make sure that it is running. See subroutine TMGR for details.

Example:

See TMGR – Timer Manager

Remarks:

Spectra already provides some basic controllers.
See section "Default Controllers" for further details.

TIMERS

TMGR

Timer Manager – the Spectra task scheduler

Main Category: **TIMERS**

File **spectra_timers.a99**
Keywords RAM, TIMERS, ANIMATIONS

Call format	`MAIN B @TMGR`
Input	-
Output	-
Stack usage	0 bytes

Description:

This subroutine is used for running the timer manager, the task scheduler that is provided with SPECTRA. It should be called after doing the program initialisation (setting up memory, allocating timer slots, etc).

TMGR acts as the main loop of your program and requires that VDP interrupts are disabled, so it first does a **"LIMI 0"**. The subroutine then basically synchronizes with the VDP by continuously checking the interrupt flag (BIT0) of the VDP status register.

If the flag is switched on (VDP memory access window open), the subroutine will first save a copy of the VDP status in the VDP status shadow register @VDPSTA and will then loop over all defined slots.
For each slot it first updates the internal counter TCOUNT, compares with the target interval defined in the slot and if it matches it calls the corresponding subroutine via BL @xxxx.

Make sure that you always properly initialize the TIMER table with >00 value. If you have junk in the table it can happen that the Timer Manager interprets it as a slot that needs to be fired and locks up.

You also need to set @THIGH to the highest slot currently in use.

See the MKSLOT subroutine for details on allocating a new timer slot.
See section "TIMERS – MEMORY SETUP" for details on the timer memory structure.

Be aware that an American TI-99/4A console (9918VDP) there are 60 ticks (interrupts) per second. On an European TI-99/4A console (9928VDP) there are 50 ticks (interrupts) per second.

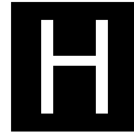
Example:

The below example uses a controller to fill the screen with a character and sets the border colour. It starts with an interval of 1 second and then gets faster until it resets to an interval of 1 second again. Each time the controller is fired, it fills with the next character and next border colour.

```
********@*****@*********************@**************************
        IDT   'TI'
        TITL  'TEST1'
        DEF   SFIRST, SLAST, SLOAD
        AORG  >A000                   ; High-memory. Use >6000 for super-cart
SFIRST  EQU   $
SLOAD   EQU   $
TIMER   EQU   >8340                   ; Allocate timer table in scratchpad memory
****************************************************************
* Main program
********@*****@*********************@**************************
MAIN    LWPI  WSSPC1                  ; Load main workspace
        LIMI  0
        LI    STACK,STKBUF            ; Setup simulated stack
        BL    @FILMEM
        DATA  >8320,>83FF,>00         ; Clear scratchpad memory (except workspace)
*--------------------------------------------------------------
* VDP setup
*--------------------------------------------------------------
        LI    R0,VMODE1               ; "graphics mode 1"
        BL    @LVDPSH                 ; Load shadow registers
        LI    R0,>0700                ; Background "black"
        MOV   R0,@VDPR7               ; Put in shadow register 7
        BL    @VIDOFF                 ; Disable screen updates
        BL    @WVDPSH                 ; Write shadow registers to VDP
        LI    R0,32
        BL    @FILSCR                 ; Clear screen
        BL    @VIDON                  ; Enable screen updates again
        BL    @WVDPSH                 ; Write shadow registers to VDP
*--------------------------------------------------------------
* Setup controller
*--------------------------------------------------------------
        CLR   @THIGH                  ; Highest slot is 0
        BL    @MKSLOT
        DATA  0,60,ZCYCLE
        MOV   @CHAR,@TPRV0            ; Store in private area slot 0
        B     @TMGR                   ; Start task scheduler


*--------------------------------------------------------------
* Controller: Cycle screen background
*--------------------------------------------------------------
ZCYCLE  INCT  STACK
        MOV   R11,*STACK+
        MOV   R0,*STACK+
        MOV   R1,*STACK               ; Just push/pop some registers for testing
*--------------------------------------------------------------
* Fill screen with character
*--------------------------------------------------------------
        INC   @TPRV0                  ; char = char + 1
        MOV   @TPRV0,R0
        CI    R0,143                  ; Character with ASC
        JLE   ZCYCLF
        MOV   @CHAR,@TPRV0            ; Reset character in private area slot 0
ZCYCLF  BL    @FILSCR                 ; Fill screen with character
*--------------------------------------------------------------
* Cycle screen background
*--------------------------------------------------------------
        MOV   @VDPR7,R0
        BL    @VWTR                   ; Write shadow register
        INC   R0                      ; Increase color
        CI    R0,>070F                ; Last color reached ?
        JLE   ZCYCLG
        LI    R0,>0700                ; Reset color
ZCYCLG  MOV   R0,@VDPR7               ; Sync shadow register
        DECT  @TSLT0                  ; interval = interval - 2
        JNE   ZCYCLH
        LI    R0,60                   ; Reset interval
        MOV   R0,@TSLT0               ; Update interval
ZCYCLH  B     @POPRG1
CHAR    DATA  32                      ; White space - ASCII 32
```

Default Controllers

DEFAULT CONTROLLERS



KBSCAN

Scan the virtual TI-99/4A keyboard

Main Category: **TIMERS**

File **spectra_ctrl_keyb.a99**
Keywords RAM, KEYBOARD, CONTROLLER

Call format	`MYTEST BL @MKSLOT` `DATA P0,P1,KBSCAN`
Input	P0 = Slot number P1 = Repeat interval
Output	@VIRTKB
Stack usage	8 bytes (R11,R0,R1,R2)

Description:

This controller implements a simple virtual TI-99/4A game keyboard. It basically scans the keyboard and joystick 1 and maps the result as bit flags on a virtual keyboard mask.

The controller stores the resulting bit-flags in the memory word @VIRTKB.
See section SPECTRA memory layout for details on memory location.

Benefit is that your game controller does not specifically need to check both keyboard and joystick. 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 'KLEFT' in @VIRTKB to 1.

Note that this controller does not support all keys, but does handle enough for supporting an arcade game.
The controller itself also checks for FNCTN-QUIT and exits to the TI-99/4A title screen when pressed.

The controller needs a 6 byte memory buffer for storing the keyboard/joystick column results.
Be default this is @RAMBUF

Example:

The below example starts the keyboard controller and a game controller that checks for up and down.

```
MAIN      ...
          BL      @MKSLOT
          DATA    0,2,KBSCAN          ; Setup virtual-keyboard controller
          BL      @MKSLOT
          DATA    1,2,GAME            ; Setup your game controller
          MOV     @PLS1,@THIGH        ; Highest slot is 1
          B       @TMGR               ; Start task scheduler
*---------------------------------------------------------------
* Controller: Check game moves
*---------------------------------------------------------------
GAME      INCT    STACK
          MOV     R0,*STACK
          MOV     @VIRTKB,R0          ; Get virtual keyboard
          COC     KUP,R0              ; 'E' pressed or joystick 1 moved up ?
          JNE     GAME1               ; No, continue checking
          ...                         ; Yes, your stuff here
GAME1     COC     @KDN,R0             ; 'X' pressed or joystick 1 moved down ?
          ...
          B       @POPRG0             ; Exit
```

Overview @VIRTKB flags

Below are the bit flags currently used by the @VIRTKB memory word (see section "SPECTRA – memory layout" for RAM location of this word.

BIT	Description	Meaning
0	Left	0=no 1=yes
1	Right	0=no 1=yes
2	Up	0=no 1=yes
3	Down	0=no 1=yes
4	Space / Fire / Q	0=no 1=yes
5	ALPHA LOCK down	0=no 1=yes
6	Pause	0=no 1=yes
7	-not used-	0=no 1=yes
8	REDO	0=no 1=yes
9	BACK	0=no 1=yes
10	QUIT	0=no 1=yes
11	-not used-	0=no 1=yes
12	-not used-	0=no 1=yes
13	-not-used-	0=no 1=yes
14	-not-used-	0=no 1=yes
15	-not-used-	0=no 1=yes

Equates for controller

Below is an overview of the equates that are delivered with the KBSCAN controller.
They can be used for doing bit comparison against @VIRTKB.

Equate	Virtual key	Real keyboard	Joystick 1	BIT in @VIRTKB	Equate value
KLFT	left	S	Left	0	>8000
KRGT	Right	D	Right	1	>4000
KUP	Up	E	Up	2	>2000
KDN	Down	X	Down	3	>1000
KUPLFT	Up and left	E+S	Up and left	2 and 0	>A000
KUPRGHT	Up and right	E+D	Up and right	2 and 1	>6000
KDNLFT	Down and left	X+S	Down and left	3 and 0	>9000
KDNRGT	Down and right	X+D	Down and right	3 and 1	>5000
KFIRE	Fire	Q or space	Fire	4	>800
KALPHA	Alpha lock down	Alpha lock down	-	5	>400
KPAUSE	Pause	P	-	6	>200
KREDO	REDO	8	-	8	>80
KBACK	BACK	9	-	9	>40
KQUIT	Quit	FCTN + 0	-	10	>20

TI-99/4A keyboard 8x8 matrix

Below is an overview of the keyboard mapping that can be checked by reading the 6 byte memory buffer @RAMBUF.

```
Keyboard 8x8 matrix: If 0 then key is down

COLUMN      0      1  2  3  4  5    6    7
       +-------------------------------+------+
ROW 7  |    =      .  ,  M  N  /    JS1  JS2 | Fire  |
ROW 6  | SPACE     L  K  J  H  ;    JS1  JS2 | Left  |
ROW 5  | ENTER     O  I  U  Y  P    JS1  JS2 | Right|
ROW 4  |           9  8  7  6  0    JS1  JS2 | Down  |
ROW 3  | FCTN      2  3  4  5  1    JS1  JS2 | Up    |
ROW 2  | SHIFT     S  D  F  G  A            +------|
ROW 1  | CTRL      W  E  R  T  Q                   |
ROW 0  |           X  C  V  B  Z                   |
       +-------------------------------------------+

See MG smart programmer 1986
September/Page 15 and November/Page 6
```

Remarks:

It is advised that the virtual keyboard runs as slot 0. You can then implement your game controller as slot 1 and all game animations as separate controllers starting slot 2.
To pause the game you would just check @VIRTKB for KPAUSE. If it is pressed you then do some preparation (e.g. mute sound chip), set @THIGH to slot 1 and all animations will stop.

See section “TIMERS – Memory setup” for details on @THIGH.

Appendix

Overview video mode tables

Below you find the list of default video mode tables provided in the file **spectra_vdp.a99**
See section "**VDP LOW LEVEL**" for details on shadow register usage

Mode	PNT VDP#2	PCT VDP#3	PDT VDP#4	SAT VDP#5	SPT VDP#6	FG/BG VDP#7	COL	Description
VMODE1	>0000	>0380	>0800	>0300	>0400	-/Black	32	Graphics mode 1
VMODET	>0000	>0380	>0800	>0300	>0400	White/ Black	40	Text mode

PNT = Pattern Name Table
PCT = Pattern Color Table
PDT = Pattern Descriptor Table
SAT = Sprite Attribute Table
SPT = Sprite Pattern Table

```
***************************************************************
* ROM: VDP graphics mode 1 (values for shadow registers)
********@*****@*********************@**************************
* 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=0: M1 \ Graphics 1 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 background color to black
****************************************************************
VMODE1   DATA  >0000,>01E2,>0200,>030E,>0401,>0506,>0680,>0700,32
```

```
****************************************************************
* ROM: VDP text mode (values for shadow registers)
********@*****@*********************@***************************
* 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 color to white/Background color to black
****************************************************************
VMODET   DATA  >0000,>01F2,>0200,>030E,>0401,>0506,>0680,>07F0,40
```