Working with ScratchPad RAMS for MIPS32® Cores Application Note

Document Number: MD00540
Revision 02.01
August 14, 2007

Contents

1 Introduction

The MIPS32® 24K®, 34K®, and 1004K™ processor core families support the addition of high-speed local memory blocks during configuration. These blocks, referred to as ScratchPad RAM (or SPRAM), provide low-latency storage for critical code and/or data. SPRAM access speed is similar to that of locked cache lines, but without impact on cache performance or maintenance.

This application note provides an introduction to working with SPRAM on a 24K-, 34K- or 1004K-based hardware target and includes example code you can use as a starting point for your SPRAM system software design. An archive of the code and scripts used in this application note can be downloaded from the MIPS Technologies website as MD00540-2B-SPRAM-APP.tgz.

The following tools were used during the development of this note:

- Malta™ Development Board with YAMON™ (v2.13) ROM Monitor (see MIPS documents MD00048 and MD00009).
- CoreFPGA3™ core card with "A00076-34Kc-2_3_0-REF0170-XC4VLX200.fl" (see MIPS document MD00481).
- MIPS® SDE Lite software development tools (v6.06.01) (see MIPS document MD00428).
- FS2™ System Navigator™ for MIPS (ver 2.2.0.0) (see <install dir>/SysNav/Doc/).

Information in this note and the accompanying files may require modification when used with other processors[1], boards, or tool environments. Device and tool behavior may also change as new versions add or enhance features.

This application note assumes you are familiar with the listed tools, and that you have a functional working environment where you are able to use the FS2 Console to control your target hardware. For additional information, refer to the documentation for each tool.

2 Detecting SPRAM

Two types of SPRAM blocks may be implemented: Instruction SPRAM (ISPRAM) and Data SPRAM (DSPRAM). To determine if your processor implementation contains SPRAM, check the *ISP* and *DSP* fields in the CP0 *Config* register: ISPRAM is implemented when *ISP* (bit 24) is set, and DSPRAM is implemented when *DSP* (bit 23) is set.

You can use the FS2 Console to read the CP0 *Config* register by name, or by using the `cop` command, specifying the coprocessor, register number, and select value.[2, 3]

1. The scripts and code may require small modifications to work with members of the 4K® or 4KE® processor core families.
2. The `device` command sets the default device (`34K vpe`) where commands are sent. Skip this command if you are not using a 34K. (I have chosen `vpe0`, because this device is enabled from reset. You can use `vpe1`, but additional steps are required to execute non-debug mode code.)
3. Some CP0 (Coprocessor 0) registers are accessible by name from the FS2 Console (case sensitive in the case of *Config*). The `cop` console command can be used to access any coprocessor register.

```
21> device vpe0
vpe0
vpe0:2> Config
0x81840483
vpe0:3> cop 0 16 0
0x81840483
```

If you know some Tcl, you can easily add a procedure to the debugger to extract and format the specific information of interest.[4, 5]

```
vpe0:33> proc spram {} {
puts "Config ISP/DSP: [expr ([Config] >> 24) & 1]/[expr ([Config] >> 23) & 1]"
}
vpe0:34> spram
Config ISP/DSP: 1/1
```

3 Reading SPRAM Configuration

Reading or modifying a SPRAM block’s configuration is not as straightforward. SPRAM configuration information is stored in special “pseudo tags” which are accessed using the “instruction/data index load/store tag” variants of the `cache` instruction while the *SPR* field (bit 28) in the CP0 *ErrCtl* register is set.

The included file, `spram.tcl` (available in the companion archive), can be sourced from the FS2 Console to add SPRAM-specific commands (Tcl procedures) to the FS2 System Navigator Console debugger. [6, 7, 8]

```
vpe0:42> cd c:/temp/MD00540-2B-SPRAM-APP/fs2_scripts
vpe0:43> source spram.tcl
<spram.tcl> ::vpe0
    ispram ?<Enable> ?<BasePAddr>??  (Display/modify ISPRAM configuration.)
    dspram ?<Enable> ?<BasePAddr>??  (Display/modify DSPRAM configuration.)
    ispramfill ?<ByteCount>?  (Fill ISPRAM from memory at same physical address.)
    ispramdump ?<ByteCount>?  (Dump ISPRAM to memory at same physical address.)
    spramtest
```

4. The FS2 System Navigator Console is based on TkCon, a Tcl/Tk-based console. Tcl/Tk documentation will be helpful in understanding command syntax for specific commands. There is a simple Tcl/Tk command reference under the Help Menu. (For a complete reference, I recommend the book “Practical Programming in Tcl/Tk”.)
5. Tcl allows dynamic addition of commands(/procedures) using the `proc` command. Commands added to the Console debugger in this way are lost when you exit the Console application.
6. If you want to access files in a path which includes spaces in the name, you will need to escape the spaces with backslashes.
7. To reuse procedure definitions, save them in a file and source that file with the `source` command.
8. Some CP0 registers are not normally accessible by name from the FS2 Console. Sourcing `spram.tcl` makes a few additional CP0 registers available by name so as to improve readability of the supplied procedures.

The added `ispram` and `dspram` commands greatly simplify SPRAM detection and configuration. Without any parameters, the `ispram` and `dspram` commands read and decode their respective SPRAM configuration tags. [9, 10, 11]

```
vpe0:187> ispram
ISPRAM tag0: 0x00258000 (BasePAddr: 0x00258000p, Enable: 0)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00258000p..0x0025ffffp)
vpe0:188> dspram
DSPRAM tag0: 0x00058000 (BasePAddr: 0x00058000p, Enable: 0)
DSPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00058000p..0x0005ffffp)
```

4 Modifying SPRAM Configuration

The `ispram` and `dspram` commands added to the debugger by sourcing `spram.tcl` can also be used to modify a SPRAM block's configuration. They take one or two optional parameters. The first is a binary enable flag (0 or 1) and the second is a physical base address. [12, 13, 14, 15, 16, 17]

```
vpe0:194> ispram 1
ISPRAM tag0: 0x00258080 (BasePAddr: 0x00258000p, Enable: 1)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00258000p..0x0025ffffp)
vpe0:195> ispram 1 0x00a00000p
ISPRAM tag0: 0x00A00080 (BasePAddr: 0x00a00000p, Enable: 1)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00a00000p..0x00a07fffp)
vpe0:196> ispram 0
ISPRAM tag0: 0x00A00000 (BasePAddr: 0x00a00000p, Enable: 0)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00a00000p..0x00a07fffp)
vpe0:197> dspram 0 0x00b00000
DSPRAM tag0: 0x00B00000 (BasePAddr: 0x00b00000p, Enable: 0)
DSPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00b00000p..0x00b07fffp)
vpe0:198> dspram 1
DSPRAM tag0: 0x00B00080 (BasePAddr: 0x00b00000p, Enable: 1)
DSPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00b00000p..0x00b07fffp)
```

9. The FS2 System Navigator Console debugger use the "p" suffix to denote a physical address.
10. Some example implementations of SPRAM allowed multiple blocks of ISPRAM and/or DSPRAM to be instantiated. The example SPRAM block supplied with the 24K/34K/1004K Processor Core Family implements at most 1 block of each. (If you wish to implement multiple blocks of ISPRAM or DSPRAM, please follow MIPS recommendations for self-identifying block configuration, as this will allow seamless software development tool support.)
11. Look in `spram.tcl` for the full implementation details of these procedures.
12. If a base address is not provided, the existing base address will remain unchanged.
13. You can only specify a new base address if the enable flag is also specified.
14. The physical base address supplied will be rounded down to the nearest 4Kbyte aligned address. This is a restriction of the *BaseAddr* field of the SPRAM configuration tags.
15. It is not recommended to overlap ISPRAM and DSPRAM regions when using tools which auto-detect and manage ISPRAM access.
16. Only the *BaseAddr* (Base Address) and *En* (Enable) fields of the first SPRAM tag are programmable in the 24K/34K/1004K example SPRAM blocks. The *Size* field is fixed at device configure time to an integer multiple of 4Kbytes.
17. You may want to invalidate the D$ just prior to Enabling DSPRAM to make sure nothing from the overlaid address range is in the D$. This is done for you if you use the `dspram` command to enable DSPRAM.

5 Accessing DSPRAM

The FS2 tools also allow the specification of physical addresses in commands. Using this capability, we can access an enabled DSPRAM block located anywhere in the physical memory. [18, 19, 20]

```
vpe0:69> dspram 1 0xf0000000p
DSPRAM tag0: 0xF0000080 (BasePAddr: 0xf0000000p, Enable: 1)
DSPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0xf0000000p..0xf0007fffp)
vpe0:70> word 0xf0000000p
0x00000000
vpe0:71> word 0xf0000000p 0x33227766
vpe0:72> word 0xf0000000p
0x33227766
```

However, typical DSPRAM access is normally made through loads and stores to a virtual address which maps to a physical address covered by the (enabled) DSPRAM. [21, 22]

```
vpe0:261> dspram 0 0x00100000p
DSPRAM tag0: 0x00100000 (BasePAddr: 0x00100000p, Enable: 0)
DSPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00100000p..0x00107fffp)
vpe0:262> word 0x80100000..+16 0x32323232 ;# Write pattern under disabled DSPRAM
vpe0:263> dspram 1
DSPRAM tag0: 0x00100080 (BasePAddr: 0x00100000p, Enable: 1)
DSPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00100000p..0x00107fffp)
vpe0:264> word 0x80100000..+16 0xdeadbeef ;# Write pattern to enabled DSPRAM
vpe0:265> word 0x80100000..+16 ;# Display pattern in DSPRAM
0xDEADBEEF 0xDEADBEEF 0xDEADBEEF 0xDEADBEEF
vpe0:266> dspram 0
DSPRAM tag0: 0x00100000 (BasePAddr: 0x00100000p, Enable: 0)
DSPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00100000p..0x00107fffp)
vpe0:267> word 0x80100000..+16 ;# Display memory under disabled DSPRAM
0x32323232 0x32323232 0x32323232 0x32323232
```

18. If there is no valid mapping to the physical address specified, a TLB entry will be borrowed to create a temporary mapping.
19. DSPRAM does not require any backing memory. Data accesses (loads and stores) to an enabled DSPRAM region are redirected to the DSPRAM. Any memory and/or devices which may exist at the same physical address range are not affected.
20. The processor can not fetch code from DSPRAM. Fetching of code from memory which exists in an enabled DSPRAM is possible but is not recommended, as this can cause debug tools to display misleading information. (Stepping through code executing behind an enabled DSPRAM is a little disturbing, as the "disassembly" displayed by the debugger is actually the disassembly of data in the DSPRAM over the actual code executing from the memory beneath it.)
21. Although the 24K/34K/1004K example SPRAM blocks can be accessed at uncached addresses, restricting SPRAM accesses to cached addresses will keep software portable. (Some SPRAM implementations my require access via cached addresses.)
22. Keeping the physical address range of the DSPRAM in the lowest 512 Mbytes (<0x20000000) allows access to the DSPRAM using the unmapped cacheable kseg0 address range (0x80000000..0x9fffffff.)

6 Accessing ISPRAM

While becoming familiar with ISPRAM, you should disable the FS2 tools automatic handling of debugger accesses to enabled ISPRAM blocks:

```
vpe0:5> config UseSpram off
off
```

Access to ISPRAM is only possible through processor fetching or through execution of “instruction index load/store tag/data” variations of the `cache` instruction.

If you have located an ISPRAM block over valid memory, you can use the added `ispramfill` command to fill ISPRAM from the memory it overlays.[23]

```
vpe0:274> ispram 0 0x1fc00000p ;# Locate/disable ISPRAM over boot code.
ISPRAM tag0: 0x1FC00000 (BasePAddr: 0x1fc00000p, Enable: 0)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x1fc00000p..0x1fc07fffp)
vpe0:275> ispramfill 64 ;# Fill first 64 bytes of ISPRAM from memory under it.
Filling first 64 bytes of ISPRAM from underlay (via kseg0).
(Each '.' is 256 bytes of progress. Press [Esc] key to cancel.)
```

You can also use the added `ispramdump` command to dump ISPRAM contents into the memory it overlays.[24, 25, 26, 27]

```
vpe0:276> ispram 0 0x00b00000p ;# Locate/disable ISPRAM over some system memory.
ISPRAM tag0: 0x00B00000 (BasePAddr: 0x00b00000p, Enable: 0)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00b00000p..0x00b07fffp)
vpe0:277> word 0x80b00000..+64 0xdeadc0de ;# Pattern memory under ISPRAM.
vpe0:278> ispramdump 32 ;# Dump first 32 bytes of ISPRAM into the memory.
Dumping first 32 bytes of ISPRAM into underlay (via kseg0).
(Each '.' is 256 bytes of progress. Press [Esc] key to cancel.)

vpe0:279> dasm 0x80b00000..+40 ;# Look at the contents of memory.
80B00000  0x10000005  b           0x80B00018
80B00004  0x00000000  nop
80B00008  0x00000000  nop
80B0000C  0x00000000  nop
80B00010  0x00012520  add         $a0,$0,$at
80B00014  0xFFFFFFFF  sd          $ra,0xFFFFFFFF($ra)
80B00018  0x00000000  nop
80B0001C  0x40809000  mtc0        $0,WatchLo
80B00020  0xDEADC0DE  ld          $t5,0xFFFFC0DE($s5)
80B00024  0xDEADC0DE  ld          $t5,0xFFFFC0DE($s5)
```

23. This method is slow but you can use the optional “ByteCount” parameter to limit how much of the ISPRAM is filled.
24. If no ByteCount is provided `ispramfill` and `ispramdump` will process the entire ISPRAM block.
25. Using the [Esc] key to cancel a long `ispramfill` or `ispramdump` will leave the system with corrupted CP0 registers.
26. An ISPRAM block does not need to be enabled for the `ispramfill` or `ispramdump` commands to work.
27. ISPRAM access is done two words at a time. The `ispramfill` and `ispramdump` ByteCount parameter is rounded up to a multiple of 8 bytes.

7 Debugging Code Executing from ISPRAM

The low-level FS2 probe driver can be configured to automatically detect enabled ISPRAM blocks, and to use `cacheops` instead of normal loads and stores for all debugger accesses to those regions. [28, 29] This facilitates support of software breakpoints and disassembly of code in ISPRAM. [30]

```
vpe0:342> config UseSpram auto ;# Turn on automatic handling of (enabled) ISPRAM.
auto
vpe0:343> ispram 0 0x00a00000p
ISPRAM tag0: 0x00A00000 (BasePAddr: 0x00a00000p, Enable: 0)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00a00000p..0x00a07fffp)
vpe0:344> word 0x80a00000..+64 0x7000003f ;# fill memory under (disabled) ISPRAM
with sdbbp
vpe0:345> ispram 1 ;# Enable ISPRAM (resynchronize debugger's ISPRAM support.)
ISPRAM tag0: 0x00A00080 (BasePAddr: 0x00a00000p, Enable: 1)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00a00000p..0x00a07fffp)
vpe0:346> copy 0xbfc00000..+64 0x80a00000 ;# fill ispram with copy of some code.
64
vpe0:347> pc 0x80a00000 ;# Point the "PC" to the ISPRAM (via kseg0.)
0x80A00000
vpe0:348> dasm [pc]..+32 ;# Show the code in the ISPRAM.
80A00000  0x10000005  b              0x80A00018
80A00004  0x00000000  nop
80A00008  0x00000000  nop
80A0000C  0x00000000  nop
80A00010  0x00012520  add            $a0,$0,$at
80A00014  0xFFFFFFFF  sd             $ra,0xFFFFFFFF($ra)
80A00018  0x00000000  nop
80A0001C  0x40809000  mtc0           $0,WatchLo
vpe0:349> bkpt setsw 0x80a00018 ;# Set a software breakpoint on code in ISPRAM.
{0x80A00018 sw}
vpe0:350> go ;# Start executing code from ISPRAM. (Should hit breakpoint.)
vpe0: Software breakpoint.
80A00018  0x00000000  nop
```

28. The tools require that ISPRAM and DSPRAM blocks do not overlap.
29. When the "UseSpram" configuration item is set to on/auto, remember that all debugger accesses to memory under an enabled ISPRAM block are redirected to the ISPRAM block, and the memory (if any) under the ISPRAM block is not accessible from the debugger until the ISPRAM is disabled or the UseSpram is configured to "off". The memory under ISPRAM is always accessible (via loads and stores) from code executing on the processor.
30. If you manually modify SPRAM configuration information, you will need to synchronize the tools to the new configuration by configuring UseSpram to off and then to your desired setting. This is done for you if you use the `ispram` and `dspram` commands.

If we disable ISPRAM, we can see that the memory beneath it was not changed by the debugger.

```
vpe0:351> ispram 0 ;# Disable ISPRAM
ISPRAM tag0: 0x00A00000 (BasePAddr: 0x00a00000p, Enable: 0)
ISPRAM tag1: 0x00008000 (Size: 32 Kbytes, 0x00a00000p..0x00a07fffp)
vpe0:352> dasm 0x80a00000..+32 ;# Look at contents of memory under disabled ISPRAM.
80A00000  0x7000003F  sdbbp          0x00000
80A00004  0x7000003F  sdbbp          0x00000
80A00008  0x7000003F  sdbbp          0x00000
80A0000C  0x7000003F  sdbbp          0x00000
80A00010  0x7000003F  sdbbp          0x00000
80A00014  0x7000003F  sdbbp          0x00000
80A00018  0x7000003F  sdbbp          0x00000
80A0001C  0x7000003F  sdbbp          0x00000
```

8 DSPRAM Performance

DSPRAM has a one-cycle load-to-use requirement. If you load something from DSPRAM and try to use it in the next instruction, the pipeline will stall for one cycle. [31]

All DSPRAM requests complete across the DSPRAM interface in order. [32] Because stores to DSPRAM complete at the end of the pipeline, the worst case DSPRAM access sequence is: store, load, use-load-data. [33]

31. This is the same as the data cache's one-cycle load-hit-to-use requirement. (Multithreading can remove these stalls.)
32. There is no bypass from a DSPRAM store to a DSPRAM load of the same data.
33. The addresses of the DSPRAM store and load have no affect on the stall.

If your core was configured with PDTrace^{TM}, you can use it to view a cycle-accurate trace of this worst case DSPRAM access sequence. [34, 35, 36]

```
halt
config UseSpram auto
tracemode +ca +io ;# Cycle accurate, inhibit overflow
ispram 1 0x00100000 ;# enable ISPRAM over (cacheable direct-mapped) kseg0
dspram 1 0x00108000 ;# enable DSPRAM just past ISPRAM
asm 0x80100000 {lui r2 0x8010}
asm {ori r2, r2, 0x8000} ;# Load r2 with base of DSPRAM
asm {lw r4, 0x10(r2)}
asm {or r3,r3,r4} ;# Load to use will cause 1-cycle stall.
asm {sw r3, 0x10(r2)}
asm {lw r4, 0x40(r2)}
asm {or r3,r3,r4} ;# Store to Load to use will cause 6-cycle stall.
asm {sdbbp} ;# Add a software breakpoint to get us back into the debugger.
pc 0x80100000 ; dasm [pc]..+32
go ; pt info ; pt all dasm
```

```
…
0.23: { cpu 0 tc 0 } mode kernel (exl=0, erl=0), isa=MIPS32, asid=0x00
0.23: { cpu 0 tc 0 } pc   0x00:0x80100000 0x3C028010  lui $v0,0x8010
1.14: { cpu 0 tc 0 } pc   0x00:0x80100004 0x34428000  ori $v0,$v0,0x8000
1.61: { cpu 0 tc 0 } idle cycles 1
1.62: { cpu 0 tc 0 } pc   0x00:0x80100008 0x8C440010  lw $a0,0x0010($v0)
2.18: { cpu 0 tc 0 } idle cycles 1
2.19: { cpu 0 tc 0 } pc   0x00:0x8010000C 0x00641825  or $v1,$v1,$a0
2.28: { cpu 0 tc 0 } pc   0x00:0x80100010 0xAC430010  sw $v1,0x0010($v0)
2.37: { cpu 0 tc 0 } pc   0x00:0x80100014 0x8C440040  lw $a0,0x0040($v0)
2.51: { cpu 0 tc 0 } idle cycles 6
2.52: { cpu 0 tc 0 } pc   0x00:0x80100018 0x00641825  or $v1,$v1,$a0
2.62: { cpu 0 tc 0 } mode debug, isa=MIPS32, asid=0x00
2.62: { cpu 0 tc 0 } idle cycles 2
```

34. To save time, you can copy a command sequence and paste it into the FS2 Console to execute the list of commands.
35. Red highlighting was added to help understand dependencies leading to stalls.
36. The first single-cycle stall in the trace output shown has nothing to do with DSPRAM. (It is an ALU to AGEN stall.)

9 Example SPRAM Target Code

The SPRAM manipulations we have done so far have all been carried out by code executing in debug-mode on the target CPU via debugger commands. The same manipulations can be done by executing non-debug-mode code on the target. If you have a very simple task (like reading CP0 *Config*), you can use FS2 Console to assemble some MIPS32 code directly into memory to accomplish this task.

```
vpe0:90> asm 0xa0100000 {lui r1, 0x0180} ;# r1 <- mask for ISP/DSP cp0
Config[24:23]
4
vpe0:91> asm {mfc0 r2, r16, 0} ;# r2 <- CP0 Config (No "ehb/jr.hb" for read.)
4
vpe0:92> asm {and r3, r2, r1} ;# r3 <- r2 | r1 (isolating ISP/DSP).
4
vpe0:93> asm {b 0xa010000c} ;# Branch to self (busy infinite loop.)
4
vpe0:94> asm {nop} ;# Branch (to self) delay slot (.
4
vpe0:95> pc 0xa0100000 ;# Uncached/Unmapped kseg1 addresses beyond those used by
YAMON.
0xA0100000
vpe0:96> dasm [pc]..+20 ;# Display code in memory
A0100000  0x3C010180  lui            $at,0x0180
A0100004  0x40028000  mfc0           $v0,Config
A0100008  0x00411824  and            $v1,$v0,$at
A010000C  0x1000FFFF  b              0xA010000C
A0100010  0x00000000  nop
vpe0:97> go ;# Execute code ending in an infinite loop.
vpe0: Emulation started.
vpe0:98> halt ;# stop execution of the infinite loop.
vpe0: User halt.
A010000C  0x1000FFFF  b              0xA010000C
vpe0:102> puts "r1: [r1]  r2: [r2]  r3: [r3]" ;# Display the value read from CP0
Config.
r1: 0x01800000  r2: 0x81840483  r3: 0x01800000
```

For more complex tasks, you will likely use C-level development tools, like MIPS SDE (Software Development Environment.) The supplied kits, intrinsics, and include files will allow you to quickly build larger and more complex programs. Below (and in the companion source archive) is an SDE example composed of a `Makefile` and a C program, `hasspram.c`. Create a `hasspram` directory under the SDE examples directory and place these files in it. You can then build executables, as shown below, to run on a Malta board using YAMON, without depending on the FS2 tools. [37, 38]

<*your SDE path here* >`/sde/examples/hasspram/Makefile`:

```
PROG            =hasspram
SRCS            =hasspram.c
CFLAGS          =-g -O0
include ../make.mk
```

37. The SDE documentation is in <*your SDE install directory name here*>`/doc/sde-guide.pdf`
38. A local tftp server was pointed to the `examples/spram` directory, and YAMON environment variables were set to cause the load command to default to tftp from that server.

<your SDE path here>`/sde/examples/hasspram/hasspram.c`:

```
#include <stdio.h>
#include <mips/cpu.h>

int
main (int argc, char **argv)
{
    unsigned int Config;
    int hasISPRAM, hasDSPRAM;

    Config = mips_getconfig();
    hasISPRAM = (Config >> 24) & 1;
    hasDSPRAM = (Config >> 23) & 1;

    printf ("Config ISP/DSP: %d/%d\n", hasISPRAM, hasDSPRAM);

    return 0;
}
```

Build using the SDE software Development tools MALTA32L “kit” (I/O will be handled by YAMON on tty0).

```
cygwin$ cd <Your SDE path here>/sde/examples/hasspram
cygwin$ sde-make SBD=MALTA32L
cygwin$ cp hasspramram.s3 <your tftp directory name here>
```

Load and run using YAMON.

```
YAMON> load /hasspramram.s3
About to load tftp://192.168.0.112/hasspramram.s3
Press Ctrl-C to break
...

Start = 0x80100000, range = (0x80100000, 0x8010bcaf), format = SREC
YAMON> go
Config ISP/DSP: 1/1
User application returned with code = 0x00000000
```

A larger set of SPRAM related functions (in the form of an SDE example) can be found in the `MD00540-2B-SPRAM-APP/sde-examples/spram` directory of the accompanying archive. Copy the `spram` directory into `<`*`your sde install dir`*`>/examples`, and follow the instructions outlined above.

10 References

A tarred gzipped archive of the code and scripts used in this note can be downloaded from MIPS Technologies’ web-site: MD00540-2B-SPRAM-APP.tgz

11 Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

Revision	Date	Description
1.00	July 30, 2007	Preliminary version.
2.00	August 14, 2007	Additional example code and links.
2.01	June 30, 2008	Update document title.