www.parallaxsemiconductor.com
sales@parallaxsemiconductor.com
support@parallaxsemiconductor.com
phone: 916-632-4664 • fax:916-624-8003

Application Note AN012

# Interfacing the Propeller to External SRAM with SPI

*This application note describes how to interface external SPI-based SRAM to the Propeller P8X32A multicore microcontroller. The Microchip 23K256 32 KB SPI SRAM is used as an example. A complete API developed in Spin supports reading, writing, block operations, as well as support for an advanced local memory cache.*

## Introduction

If there is one thing embedded programmers can never have enough of it's memory. Code space, data space—more is better, always. That said, microcontroller manufacturers must balance large on-die silicon memory with cost analysis and try to select the amount of memory that makes the most sense for the majority of applications. However, if you find that you need more memory then external memory over a serial or parallel bus might be the way to go.

Microcontrollers, unlike microprocessors, usually do not come with external bus interfaces. This wastes precious I/O pins for limited gains in most applications to support external data, address, and control buses. Nonetheless, some microcontrollers do have external bus interfaces allowing designers to augment their on-chip memory map with external RAM and FLASH memories. The nice thing about this is when you add more memory it integrates right into the controller's memory map, so you don't have to change your programming, just the hardware.

However, microcontrollers that support external bus interfaces are usually 100+ pins, more expensive, and lack other I/O features. That said, the other option to add memory to a microcontroller is external non-integrated memory that is interfaced via a driver and either a serial or parallel bus interface.

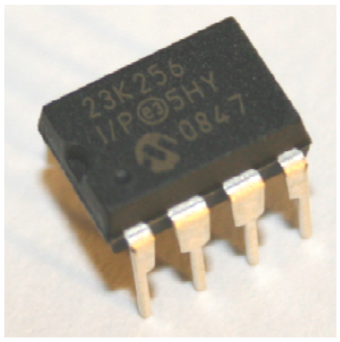
Obviously a parallel bus is going to be faster than a serial configuration, but parallel buses again require a large number of I/O pins (8-12 in most cases) to support a byte-wide bus transfer plus control signals and latches for the address bus. Such a hardware design is non-trivial and expensive, but if the ultimate in speed is the goal then this is the way to go.

A simple option that takes very little hardware skill is to use a serial interfaced external memory. This is the approach used here. Moreover, we are going to focus on adding SRAM rather than more FLASH, but the concept is the same: to use a serial communication scheme.

With that in mind, the SPI protocol was selected for its sheer speed (in excess of 25 MHz with 100 MHz quad SPI devices now available) which, with the right driver, can almost match internal memory with a multi-cog driver design (more on this later).

The next step is the selection of an external SPI SRAM. At this point, a number of companies develop SPI SRAM devices including On Semiconductor, Intersil, Maxim, NXP, and Microchip to name a few. And all of their SPI SRAM devices function more or less the same due to JEDEC[1] compliance. Thus, the pinouts of most are nearly identical and the command protocols overlap quite a bit, so once you write one driver, switching devices takes very little software porting. That said, the Microchip 23K256 device selected here is readily available, economical in low quantities, and comes in both SMT as well as a nice DIP 8 package (Figure 1) which is great for prototyping.

**Figure 1: DIP Version of the Microchip 23K256 32K Serial SPI SRAM**



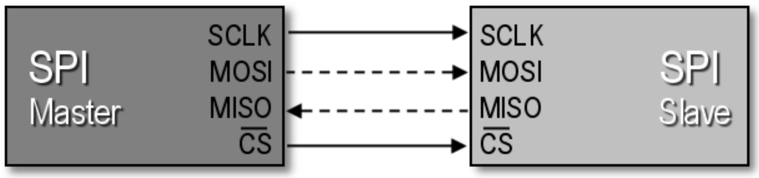
The 23K256 features a 32 K-byte / 256 K-bit storage capacity and 20 MHz operation. The plan is to write a Spin-based driver that allows reading, writing and block operations to be performed with the external SRAM. Additionally, the examples illustrate how to implement a high-speed local memory cache. This cache in essence permits access to the external SPI SRAM a "page" at a time (usually 128, 256, or 512 bytes) and if the page is currently cached in local memory then the driver re-directs to local memory and doesn't need to go out to the slower external SPI SRAM. The only time the SRAM is accessed is when a page fault occurs or the page is "dirty" and needs to be written back to the RAM.

## SPI Protocol Primer

SPI stands for Serial Peripheral Interface, originally developed by Motorola. It's one of most popular modern serial standards including $I^2C$ (Inter Integrated Circuit developed by Phillips). SPI (unlike $I^2C$ which has no separate clock) is a clocked synchronous serial protocol that supports full duplex communication from master to slave. $I^2C$ only requires two wires and a ground; SPI needs 3 wires, ground, and chip select line(s) to enable the slave devices. Due to the synchronous separate clock line SPI is *much* faster, so in cases where speed is desired the extra clock line is warranted and SPI is used.

However, the advantage of $I^2C$ is that you can connect up to 127 $I^2C$ devices on the same 2-bus lines, due to the shared bus protocol and unique addressing. On the other hand, SPI bus protocol requires that every SPI slave has its own chip select line and only a single SPI device at a time can use the bus. So, if you need speed, use SPI; if you need a lot of devices to share a bus, use $I^2C$.

**Figure 2: The SPI Electrical Interface**



## SPI Bus Basics

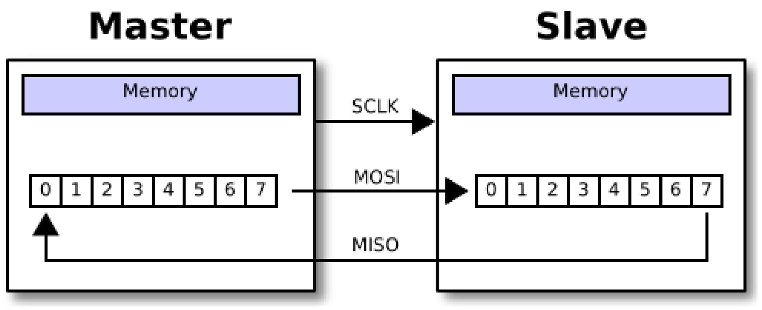
Figure 2 depicts a simple diagram between a master (left) and a slave (right) SPI device and the signals between them which are (along with common aliases):

- SCLK/SCK — Serial Clock (output from master).
- MOSI/SI — Master Output, Slave Input (output from master).
- MISO/SO — Master Input, Slave Output (output from slave).
- CS/SS — Chip/Slave Select (active LOW; output from master).

Note that some devices may have slightly different naming conventions, but there will always be data out, data in, a clock, and a chip select pin.

SPI is very fast since not only is it clocked, but it's a simultaneous full duplex protocol which means that data clocks out of the master into the slave, data also clocks from the slave into the master at the same time. This is facilitated by transmit and receive bit buffers that constantly re-circulate, as shown in Figure 3 below.

**Figure 3: Circular SPI Buffers**



The use of the circular buffers make it possible to send and receive data (2 bytes total data transmitted) in only 8 clocks rather than clocking out 8 bits to send then clocking in 8 bits to receive. Of course, in some cases the data clocked out or in is "dummy" data, meaning when you write data and you are not expecting a result. In other words, the data you clock in is garbage and you can throw it away. Likewise when an SPI read is performed, typically you place a $00 or $FF in the transmit buffer as dummy data since something has to be sent and it might as well be predictable.

Sending bytes with SPI is similar to the serial RS-232 protocol: place a bit of information on the transmit line, then strobe the clock line (of course RS-232 has no clock). Concurrently, read the receive line since data is being transmitted in both directions.

This is simple enough, but SPI protocol has some very specific details regarding when signals should be read and written relating to the rising or falling edge of the clock, as well as the polarity of the clock signal. This way there is no confusion about edge, level, or phase of the signals. These various modes of operation are logically referred to as the "SPI mode" and must be agreed upon by the master and slave before communication can begin.

The SPI modes are numerically listed in Table 1 below for reference.

**Table 1: SPI Clocking Modes**

| Mode# | CPOL (Clock Polarity) | CPHA (Clock Phase) |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |

## Mode Descriptions

- **Mode 0** — The clock is active when HIGH. Data is read on the rising edge of the clock. Data is written on the falling edge of the clock (default mode for most SPI applications and chips).
- **Mode 1** — The clock is active when HIGH. Data is read on the falling edge of the clock. Data is written on the rising edge of the clock.
- **Mode 2** — The clock is active when LOW. Data is read on the rising edge of the clock. Data is written on the falling edge of the clock.
- **Mode 3** — The clock is active when LOW. Data is read on the falling edge of the clock. Data is written on the rising edge of the clock.

Note that most SPI slaves default to mode 0, so typically this mode is what is used to initiate communications with a SPI device.

**Figure 4(a): SPI Timing Diagrams for Clock Phase Polarity (CPHA=0)**

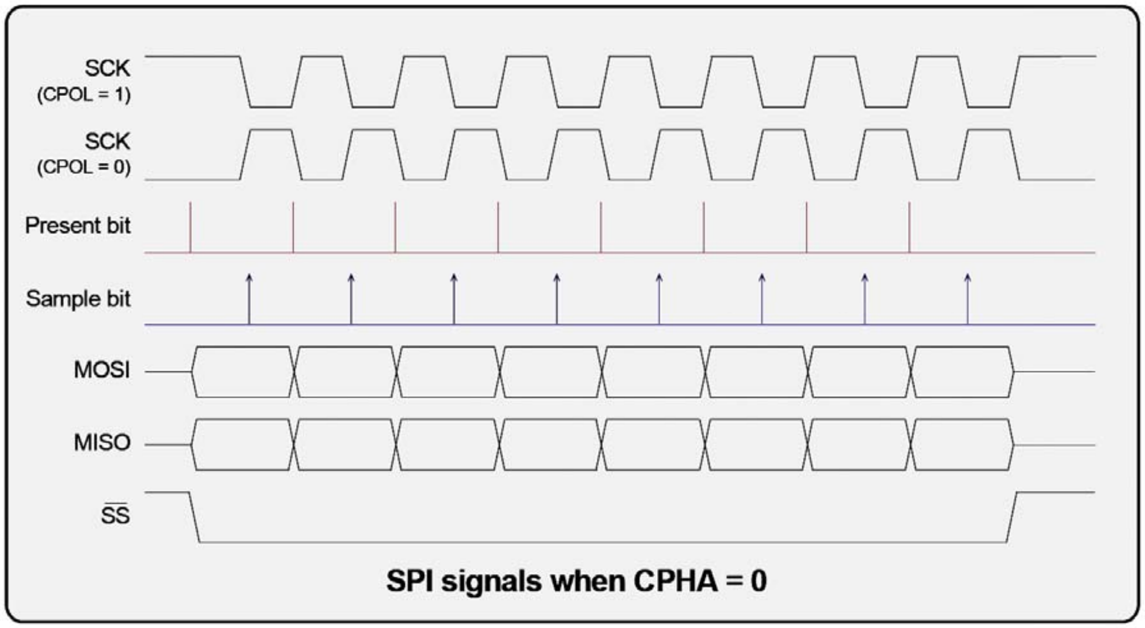
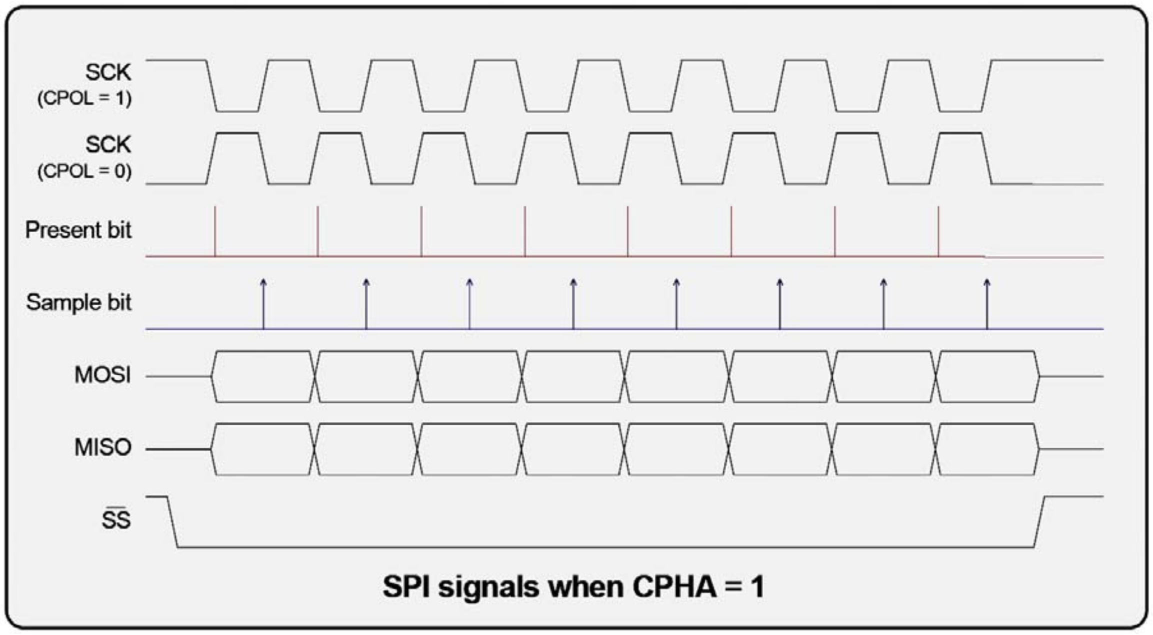


**Figure 4(b): SPI Timing Diagrams for Clock Phase Polarity (CPHA=1)**



## Basic SPI Communications Steps

Figure 4(a) and (b) depict the timing diagrams for all variants of clock polarity (CPOL) and clock phase (CPHA). It's necessary to adhere to these timing constraints during communications. In most cases, use **Mode 0** since it's the default that most SPI devices boot with.

The Propeller uses software to implement peripherals virtually. An SPI driver for a dedicated cog may be written in Spin code or Propeller Assembly to send and receive SPI data. This application notes opts for simple Spin, but later you might want to port to assembly language. No matter how you implement the SPI driver, the process is very straightforward. First, select a set of I/O pins on the Propeller and assign them the function of MOSI (master out slave in), MISO (master in slave out), SCLK (clock signal), and CS (chip select). This is accomplished with the code shown below (from the driver developed in this application note):

```
PUB SPI_Init
{{
DESCRIPTION: Initializes the SPI IO's, counter, and mux, selects channel 0 and returns.

PARMS: none.
RETURNS: nothing.
}}
  ' set up SPI lines
  OUTA[SPI_MOSI]     := 0 ' set to LOW
  OUTA[SPI_SCK]      := 0 ' set to LOW
  OUTA[SPI_CS]       := 1 ' set to HIGH (de-assert)

  DIRA[SPI_MOSI]     := 1 ' set to output
  DIRA[SPI_MISO]     := 0 ' set to input
  DIRA[SPI_SCK]      := 1 ' set to output
  DIRA[SPI_CS]       := 1 ' set to output

' end SPI_Init
```

The SPI pin constants are defined in the driver's CON section and assigned to appropriate pins on your particular Propeller development board. For example, on the Propeller C3 development board[2], something like this would work:

```
  ' SPI pins (convenient free I/Os' on a standard Prop C3 platform)
  SPI_CS    = 3 ' SPI chip select (active low)
  SPI_SCK   = 0 ' SPI clock from master to all slaves
  SPI_MOSI  = 1 ' SPI master out serial in to slave
  SPI_MISO  = 2 ' SPI master in serial out from slave
```

Once the I/O pins are selected and initialized a single method is required to send and receive bytes over the SPI bus. However, since this single method is written in Spin and must be very fast, it's worth the effort to optimize it and unroll the main loop. Take a look below for the code listing.

```
PUB SPI_Write_Read( num_bits, data_out, bit_mask) | data_in, num_bits_minus_1
{{
DESCRIPTION: This method writes and reads SPI data a bit at a time (SPI is a circular buffer
protocol), the data is in MSB to LSB format and up to 32-bits can be  transmitted and
received, the final result is bit masked by bit_mask

PARMS:

num_bits : number of bits to transmit from data_out
data_out : source of data to transmit
bit_mask : final result of SPI transmission is masked with this to grab the
           relevant least significant bits

RETURNS: data retrieved from SPI transmission

}}
  ' clear result
  data_in := 0
  num_bits_minus_1 := num_bits-1 ' optimization pre-compute this since compiler
                                 ' will continually evaluate any constant math

  ' optimize code for 8 bit case by unrolling loop, if other bit lengths occur
  ' frequently unroll as well
  if (num_bits == 8)

    ' begin 8-bit case ----------------------------------------------------
    ' now read the bits in/out

    ' bit 0
    OUTA[SPI_SCK]  := 0                                ' drop clock
    OUTA[SPI_MOSI] := (data_out >> 7)                  ' place next bit on MOSI
    data_in := (data_in << 1) + INA[SPI_MISO]          ' now read next bit from MISO
    OUTA[SPI_SCK]  := 1                                ' raise clock

    ' bit 1
    OUTA[SPI_SCK]  := 0                                ' drop clock
    OUTA[SPI_MOSI] := (data_out >> 6)                  ' place next bit on MOSI
    data_in := (data_in << 1) + INA[SPI_MISO]          ' now read next bit from MISO
    OUTA[SPI_SCK]  := 1                                ' raise clock

    ' bit 2
    OUTA[SPI_SCK]  := 0                                ' drop clock
    OUTA[SPI_MOSI] := (data_out >> 5)                  ' place next bit on MOSI
    data_in := (data_in << 1) + INA[SPI_MISO]          ' now read next bit from MISO
    OUTA[SPI_SCK]  := 1                                ' raise clock
    ' bit 3
    OUTA[SPI_SCK]  := 0                                 ' drop clock
    OUTA[SPI_MOSI] := (data_out >> 4)                   ' place next bit on MOSI
    data_in := (data_in << 1) + INA[SPI_MISO]           ' now read next bit from MISO
    OUTA[SPI_SCK]  := 1                                 ' raise clock

    ' bit 4
    OUTA[SPI_SCK]  := 0                                 ' drop clock
    OUTA[SPI_MOSI] := (data_out >> 3)                   ' place next bit on MOSI
    data_in := (data_in << 1) + INA[SPI_MISO]           ' now read next bit from MISO
    OUTA[SPI_SCK]  := 1                                 ' raise clock

    ' bit 5
    OUTA[SPI_SCK]  := 0                                 ' drop clock
    OUTA[SPI_MOSI] := (data_out >> 2)                   ' place next bit on MOSI
    data_in := (data_in << 1) + INA[SPI_MISO]           ' now read next bit from MISO
    OUTA[SPI_SCK]  := 1                                 ' raise clock

    ' bit 6
    OUTA[SPI_SCK]  := 0                                 ' drop clock
    OUTA[SPI_MOSI] := (data_out >> 1)                   ' place next bit on MOSI
    data_in := (data_in << 1) + INA[SPI_MISO]           ' now read next bit from MISO
    OUTA[SPI_SCK]  := 1                                 ' raise clock

    ' bit 7
    OUTA[SPI_SCK]  := 0                                 ' drop clock
    OUTA[SPI_MOSI] := data_out                          ' place next bit on MOSI
    data_in := (data_in << 1) + INA[SPI_MISO]           ' now read next bit from MISO
    OUTA[SPI_SCK]  := 1                                 ' raise clock

    ' end 8-bit case --------------------------------------------------------

  else ' general n bit case

    ' now read the bits in/out
    repeat num_bits
      ' drop clock
      OUTA[SPI_SCK] := 0

      ' place next bit on MOSI
      ' optimization, no need for "& $01"
      OUTA[SPI_MOSI] := ((data_out >> (num_bits_minus_1--)))
      ' now read next bit from MISO
      data_in := (data_in << 1) + INA[SPI_MISO]

      ' raise clock
      OUTA[SPI_SCK] := 1

  ' set clock and MOSI to LOW on exit
  OUTA[SPI_MOSI]  := 0
  OUTA[SPI_SCK]   := 0

  ' at this point, the data has been written and read, return result
  return ( data_in & bit_mask )

' end SPI_Write_Read
```

`SPI_Write_Read` is a great example of optimization in the Spin language. Here, the main loop is unrolled for the "number of bits equals 8" case. The problem with all loops is that the conditional, increment, and repeat code actually takes up some time. When trying to squeeze every microsecond/millisecond of performance out of Spin, those computations aren't acceptable. Thus, an optimization technique is used here called "loop unrolling" which unrolls a loop and manually performs each iteration.

In this case, a good 20% speed increase was realized. Additionally, math simplification and experimentation with Spin operations for the bit operations were performed to make it even faster. Of course, in Propeller Assembly language, we wouldn't have to be this aggressive, but since the SRAM API including the SPI code above is all written in Spin, its pays to optimize.

Optimization aside, the method is very simple. All it does is set the clock line LOW, then place a bit on the SPI MOSI output line, read the MISO input line, assert the clock, and bit-shift the output and input bytes until the operation is complete. If the caller is requesting an 8-bit write/read operation then the optimized unrolled loop is used, otherwise, a slower `repeat` loop is used for 16-, 24-, and 32-bit operations.

You might wonder, why not optimize the 16-, 24-, and 32-bit variants as well? The reason is they happen so rarely that they account for very little of the transfers (usually address operations and setup), so optimizing them doesn't pay off as much. However, optimizing the 24-bit variant and unrolling it can't hurt, and if you want some more speed then you should perform this further optimization since it does help for single-byte transfers. However, later we will see a cache implementation which really speeds things up.

## Interfacing to the Microchip 23K256 32 KB SRAM

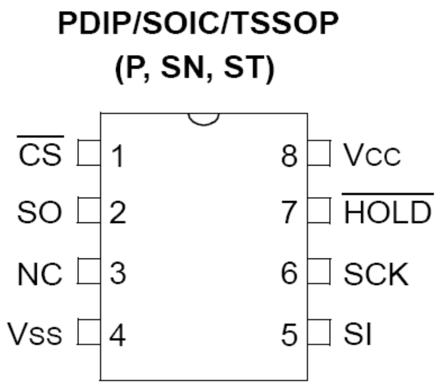
The 23K256 SRAM has a rather robust set of functions which is outlined in its data sheet[3]. The SRAM is non-trivial; to expose all its functionality and speed read the data sheet (23K256_SRAM.pdf) in the included ZIP file.

This application note develops a simple API that allows you to read, write, and fill the SRAM with data. These methods can access the 32 KB of space and immediately put it to use. However, the driver methods are written in Spin, so if you want to speed them up, you will have to recode to Propeller Assembly (along with the SPI driver itself).

### Device and Hardware Setup

The 23K256, like most SPI devices only requires 4 signals; *MOSI*, *MISO*, *CLK*, and *CS*. Figure 5 is an illustration of the pinout of the 23K256.

**Figure 5: Signal Pinout for the 23K256 32K SPI SRAM**



Referring to Figure 5, pin 1 is the chip select line (active LOW), pin 2 is the serial out line which maps to *MISO*, pin 5 is the serial in line which maps to *MOSI*, and finally pin 6 is the clock line. Vcc and Vss are power and ground respectively and the 23K256 can tolerate from 2.7-3.6V with the I/O being compatible with 5V TTL/CMOS with 100 ohm series resistors. Make sure to power the chip and place a 0.01–0.1 µF ceramic capacitor close to the Vcc pin for noise and bypass.

The remaining signal line on pin 3 is a no-connect, but *HOLDn* on pin 7 deserves a moment of discussion. The *HOLDn* line when asserted LOW basically takes the device off the SPI bus and all input signals (clock and serial in) are ignored. Furthermore, serial out is placed in a high impedance state. Thus, *HOLDn* is a great way to take a device off a shared bus for a moment and then place it back on the bus right where it left off. The chip select line *CSn* is a little different; when this is de-asserted, the device is removed from the bus as well, but its state is lost and if you were in the middle of an operation it would be lost. In our case, *HOLDn* is tied HIGH and not used.

To interface one or more 23K256 devices to the Propeller chip is easy: just select 4 I/O signals that are free and not connected or loaded to any other devices, and hook the Propeller I/O pins directly to pins 1, 2, 5, and 6. (Remember SPI is high speed, so the signal lines have to be clear of other devices hanging on them).

If you re-write the driver in Propeller Assembly  or push the SPI bus past the 10 MHz mark, consider putting some series 33 Ω to 50 Ω dampening resistors inline with the SPI bus signals to reduce noise and transmission line effects on the signals themselves. But, for the Spin driver, just hook some wires up from the Propeller to the 23K256 and it should work fine. Keep the wires short and direct; if possible run them flat against your prototype board.

## Software Driver Considerations

The SRAM data sheet gives you all the architectural details of the SRAM itself, but from our perspective all we are interested in is reading, writing, and understanding devices the three modes of operation:

- Byte Mode — A single byte is read or written.
- Page Mode — A page of 32 bytes is read or written.
- Sequential Mode — Any number of bytes is read or written.

Most SPI devices, whether they be SRAM, FLASH or other array-based memory, have a number of optimized “access” modes to decrease the amount of SPI traffic to read and write data. For example, to write a single byte to the SRAM, you first need to address the byte (2 bytes) and then write the byte (1 byte) and let’s not forget the command itself (1 byte). Therefore, to write a single random access byte is 4 bytes of information! That’s a huge waste. But, if you need to randomly access the SRAM, that’s what to expect.

What if you want to write a continuous stream of bytes? Maybe you have a 1024-byte buffer in Propeller hub RAM that you want to copy to the SPI SRAM. Well, that’s where the sequential mode comes in. In this mode, you write the instruction for sequential access, then the address, then you write byte after byte, so the overhead to write 1024 bytes is nearly zero and the real work are continuous 8-bit transfers (this is why the 8-bit transfer operation was optimized in the SPI API).

Therefore, a complete API would support these modes and would either allow user selection via function names or parameters or maybe automatic mode selection based on the data sent, size, etc. Some features to think about for future drivers. For this application note, the

***byte*** and ***sequential*** modes are supported in the API. The ***page*** mode is exotic and not really useful.

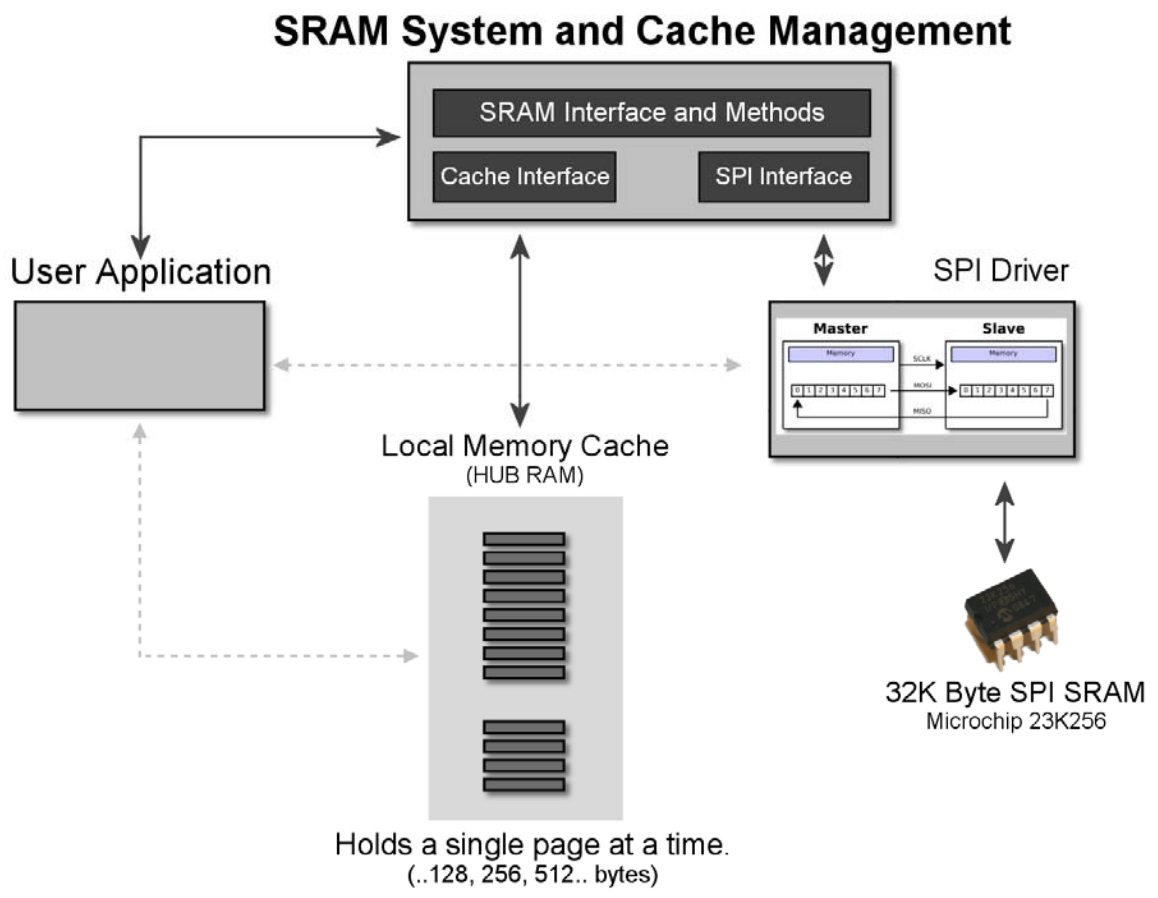
The interesting thing about the Microchip SRAM is their simplicity in commands. They only have 4 commands as shown in Table 2 below.

**Table 2: SRAM Commands**

| Instruction Name | Value | Description |
|---|---|---|
| READ | 0000 0011b | Read data from memory array beginning at selected address |
| WRITE | 0000 0011b | Write data to memory array beginning at selected address |
| RDSR | 0000 0101b | Read STATUS register |
| WRSR | 0000 0001b | Write STATUS register |

Note that there is no mention of the 3 modes of operation (byte, page, sequential). These are set and selected initially by writing the ***status*** register and then the chip remains in that mode until you change it—refer to the data sheet to learn more.

**Figure 6: The SRAM Driver Components**



## Understanding the SRAM Driver Object and API

The SRAM driver object is composed of three major components as shown in Figure 6:

- The SPI Driver
- The SRAM System Interface
- The Local Memory Cache Sub-System

The SPI driver as outlined before uses standard SPI protocol to communicate with any external SPI device. You simply need to connect the 23K256 SPI SRAM to your Propeller board's I/O pins (make changes in the `CON` section of your program to reflect the final I/O assignment) and you are ready to use the SPI function directly if you wish. However, if you are using the SRAM object as intended then the SPI API can be ignored for the most part since the SRAM system talks directly to the SPI methods itself. Nonetheless, if you want to use the SPI methods to talk SPI protocol to some other device you are free to do so and it won't cause any issues with the SRAM methods or state.

The SRAM system itself is composed of a number of methods to read, write, fill and initialize the SRAM. In most cases, you will only need to access the SRAM methods (along with the cache methods) and you won't make calls to the SPI functions directly.

Finally, the cache sub-system implements a local hub memory cache that caches a single page of SRAM in block sizes that are user selectable in the driver. Like any cache, this speeds up writing and reading access times overall for the driver.

The SRAM driver object integrates all these components into a single object named SRAM_driver_23K256_v010.spin (located in the ZIP) Let's briefly review the API methods and then they will all be exercised with a demo that brings it all together.

For more detail about the parameters, side effects and functionality of the API methods, please review the driver source; it's well documented and goes into heavy detail on each method.

### SPI Methods

There are only 3 SPI methods in the SRAM driver object:

`PUB SPI_Init` — This method initializes the SPI I/O pins and sets their direction and state. This is called internally by the SRAM `Init` method, so you shouldn't need to call this.

`PUB SPI_Reset_Pins(pSPI_CS, pSPI_SCK, pSPI_MOSI, pSPI_MISO)` — This method lets you change which pins are connected to your SPI device. Typically, this method is called after a call to the main `Init` method and you want to change I/O pins to talk to another SPI device (maybe another SRAM).

`PUB SPI_Write_Read(num_bits, data_out, bit_mask)` — This method performs the actual bit- banging to send and receive SPI packets. You can send/receive any number of bits, but in most cases you will work with 8-, 16-, 24-, or 32-bit size data. Again, you typically will not need to call this method yourself if you are happy wit the SRAM access methods.

## SRAM Access Methods

This is the primary API interface for communicating with the external Microchip 23K256 32 KB SPI memory. Normally, your application will make an initial call to the `Init` method with a set of I/O pins to assign to chip select, clock, serial out, and serial in. The `Init` method then initializes the SPI driver itself as well as starts up the 23K256 and puts it into sequential access mode. Finally, the method initializes the cache sub-system and puts it into a known state with an empty cache.

`PUB Init(pSPI_CS, pSPI_SCK, pSPI_MOSI, pSPI_MISO)` — This method initializes the whole SRAM interface, SPI driver, and cache.

`PUB SRAM_Init(mode)` — This method is internal and sends commands to the SRAM to place it in the requested access mode.

`PUB SRAM_Chip_Select` — This method asserts the chip select *CSn* line on the SRAM.

`PUB SRAM_Chip_Deselect` — This method de-asserts the chip select *CSn* line on the SRAM.

`PUB SRAM_Write(addr, num_bytes, src_buffer_ptr)` — This method writes any number of bytes from a buffer to the SRAM.

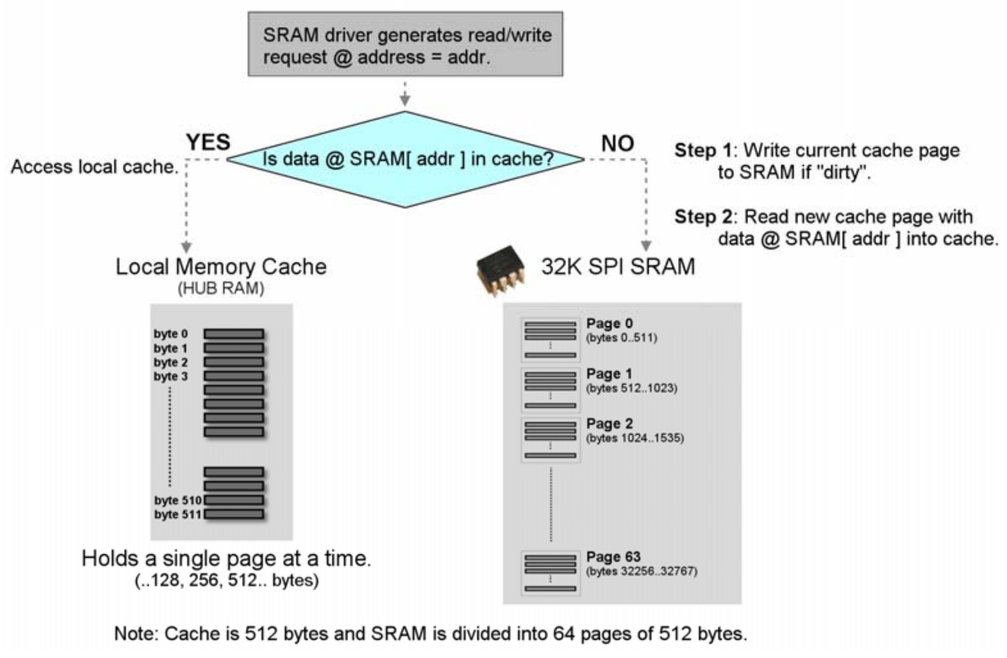
`PUB SRAM_Fill(addr, num_bytes, value)` — This method fills a region of the SRAM with a specific value.

`PUB SRAM_Read(addr, num_bytes, dest_buffer_ptr)` — This method reads any number of bytes from the SRAM and stores them in a buffer.

`PUB SRAMR(addr)` — This method reads a single byte from the SRAM.

`PUB SRAMW(addr, data)` — This method writes a single byte to the SRAM.

**Figure 7: Cache Architecture and Detail**


SRAM driver generates read/write request @ address = addr.
Is data @ SRAM[ addr ] in cache?
YES
Access local cache.
NO
Step 1: Write current cache page to SRAM if "dirty".
Step 2: Read new cache page with data @ SRAM[ addr ] into cache.
Local Memory Cache
(HUB RAM)
byte 0
byte 1
byte 2
byte 3
byte 510
byte 511
Holds a single page at a time.
(..128, 256, 512.. bytes)
32K SPI SRAM
Page 0
(bytes 0..511)
Page 1
(bytes 512..1023)
Page 2
(bytes 1024..1535)
Page 63
(bytes 32256..32767)
Note: Cache is 512 bytes and SRAM is divided into 64 pages of 512 bytes.

## Cache Support Methods

Referring to Figure 7, the SRAM cache is implemented as local Propeller hub memory of any power-of-2-fixed-size page (2…128, 256, 512 bytes, etc.). Then, when a byte is read from the SRAM, instead of reading a single byte, the entire page is read in from the SRAM. If any byte with an address located in the cached page is read or written, then that access is made locally in the cache and the external SRAM is *not* accessed. This is the where the speed gain is realized in the cache; as long as memory accesses remain in the cache page, the external memory isn't accessed.

However, at some point the application will require an address that is not in the cache. In this case a new page must be loaded in and the old page destroyed. This is called "thrashing" the cache. This is where the concept of "dirty" and "clean" cache pages comes into play.

A cache page is said to be "clean" if the page has only been read. However, if writes have occurred to the cached data, the cache becomes "dirty" and those changes must be written out to the external SRAM to synchronize it to the changes made by the application program. Thus, dirty cache pages must be written back out to the SRAM when a new cache page is requested by the user via a memory address that is not currently cached. Now, let's review the cache methods and follow up with an example.

`PUB SRAM_Cache_Init` — This method initializes the cache. This is called internally by the main `Init` method.

`PUB SRAM_Cache_Flush` — This method forces a "flush" of the cache. Flushing the cache refers to writing dirty pages out and synchronizing them.

`PUB CacheR(addr)` — This method performs a read from the external SRAM with cache support. Send the address you want to read from and the cache engine does the rest for you all behind the scenes. Note the shortened name to make "array like" access more natural and save typing.

`PUB CacheW(addr, data)` — This method performs a write to the external SRAM with cache support. Send the address and data you want to write to the external SRAM and the cache engine does the rest for you all behind the scenes. But, be aware that nothing will be written to SRAM until a cache page thrash occurs or you manually flush the cache. Note the shortened name to make "array like" access more natural and save typing. Also, this method returns the value `data` itself, so you can use this write method in expressions.

### Example Cache Operation and State Tracking

Let's step through a hypothetical example of the cache functioning. First, configure the cache setup in the driver object file's `CON` section; here's the code of interest:

```
' cache constants, YOU must set these to adjust the cache size to optimize your
' application usually 128, 256, 512 will be optimal, but you might need a really
' small or really large cache to see performance improvement. You set the CACHE_SIZE
' then compute CACHE_SIZEL2

  CACHE_SIZE   = 512    ' size of cache in bytes, must be power of 2, you set this
  CACHE_SIZEL2 = 9      ' size of cache log2 of cache_size, you set this,
                        ' so log2(512) = 9, log2(256)=8, log2(128)=7, etc.
  CACHE_MASK   = CACHE_SIZE - 1 ' bit mask
```

By default the cache size is set to 512 bytes. If you want to change it, then change the constant `CACHE_SIZE`; additionally you must compute the *log*$_2$ `CACHE_SIZE` and assign it to `CACHE_SIZEL2`. With the cache size of 512 bytes, that means if we use the cache read/write functions, the moment a read operation is performed, the page that contains the byte will be cached and 512 bytes read into the Propellers hub RAM. With that in mind, let's step through the example cache use.

**State 1:** The system just started, cache is empty.

**State 2:** Using the cache methods memory location 756 is read from the external SRAM. The cache is empty; therefore, a page must be brought in. Since each page is 512 bytes, memory location 756 is located on the second page or page 1 if we are counting 0-based (which we are). Page 1 is cached into the local cache memory (which is nothing more than a byte buffer in the Propeller's hub RAM). The cache state is updated as page 1 loaded and clean.

**State 3:** Calls are made using the cache methods to read addresses 600-1000 from the external SRAM. Since the cache holds page 1 (addresses 512 to 1023) all of these reads pass straight through to the local hub RAM and no external SPI traffic is needed, thus these read operations run at near-full Spin speed.

**State 4:** Calls are made to write memory locations 800-900. These locations are once again located in the cache since all addresses 512 to 1023 are cached with page 1. Again, no external SPI traffic is generated since the cache has all the data. However, these write operations dirty the cache and the dirty flag is set.

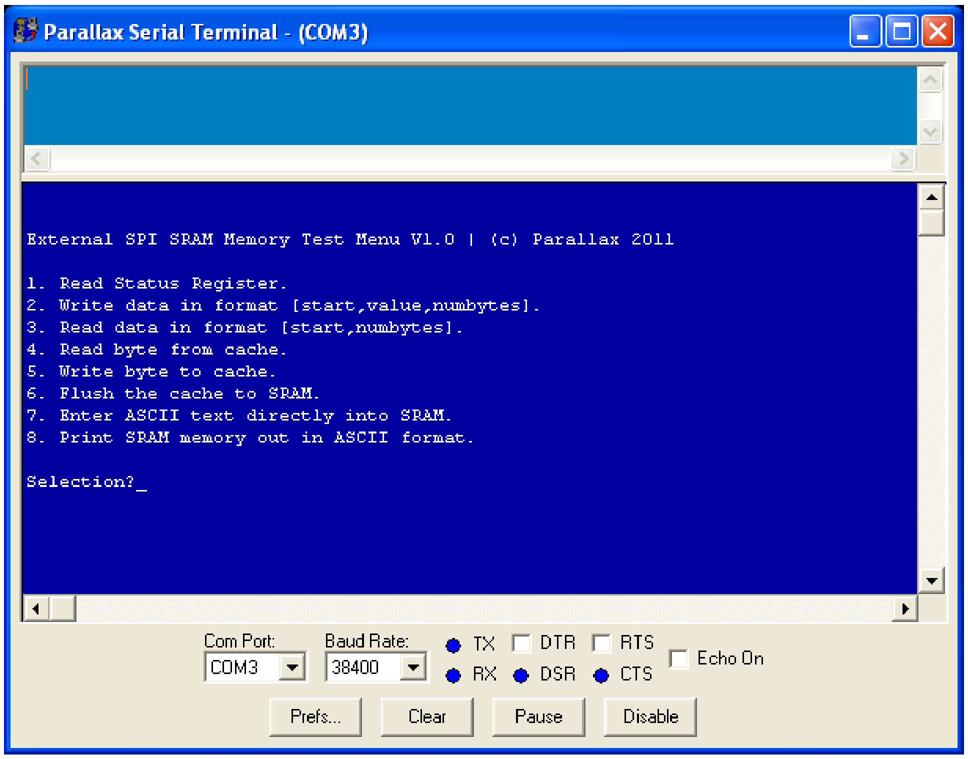
**State 5:** A call is made to read memory location 56. This is in page 0 of the external SRAM. A cache thrash is generated and the cache system needs to load a new cache page to access this memory location. However, before the current page can be discarded and a new page loaded (page 0), the cache manager notes that the current cache page 0 is dirty, and thus writes the current page back out to the cache. This synchronizes the external SRAM once again with what has been going on locally. Now, the cache manager is free to load page 0 into the cache, set the state flag to clean, and return the value of memory location 56.

When writing is performed, we only take a hit when a new cache page is required. So, for an application where the external SRAM is initially written with a set of data and then used as a read-only device, performance is instantaneous. On the other hand, if byte addresses are generated that are beyond cache page boundaries every cycle, coupled with write operations, then the cache actually makes the system slower. It's up to you to benchmark performance and decide of the cache helps, and determine the optimal cache page size for maximum performance. Cache sizes of 64 and 512 are probably optimal for the two extreme cases of far and near address histograms.

## Hands-on Demonstration Application

The goal of the final demo is to show off the entire SRAM API and give you a good working template to develop your own applications. The demo is shown in Figure 8 below running on the Parallax Serial Terminal (PST)[4]. The demo is completely serial UART based, thus you can get it to work with little more than a Propeller chip with a USB to serial programming interface, power, and of course the external SPI RAM hooked up properly. The demo is menu driven and each menu item illustrates a different SRAM API operation, so you can see them all in action and experiment in real-time.

**Figure 8: The SRAM Demo Program in Action**



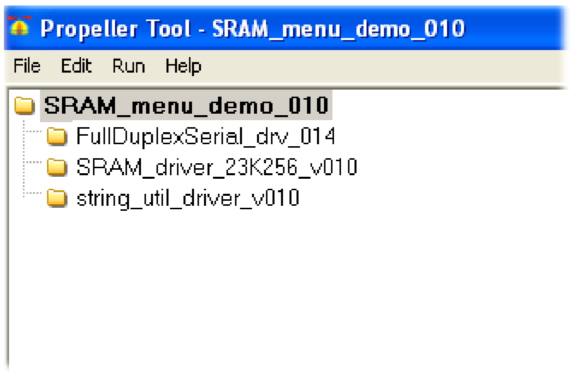
```
External SPI SRAM Memory Test Menu V1.0 | (c) Parallax 2011

1. Read Status Register.
2. Write data in format [start,value,numbytes].
3. Read data in format [start,numbytes].
4. Read byte from cache.
5. Write byte to cache.
6. Flush the cache to SRAM.
7. Enter ASCII text directly into SRAM.
8. Print SRAM memory out in ASCII format.

Selection?_
```

Com Port: COM3
Baud Rate: 38400

NOTE: The demo runs at 38,400 baud with N81 for serial settings.

To compile and run the demo, use the top-level application SRAM_menu_demo_010.spin located in this note's ZIP file. The demo includes the main SRAM driver as well as a VGA driver and utility object with some useful text and parsing functions that might come in handy. Figure 9 shows the file tree.

**Figure 9: File Tree for the SRAM_menu_demo_010**

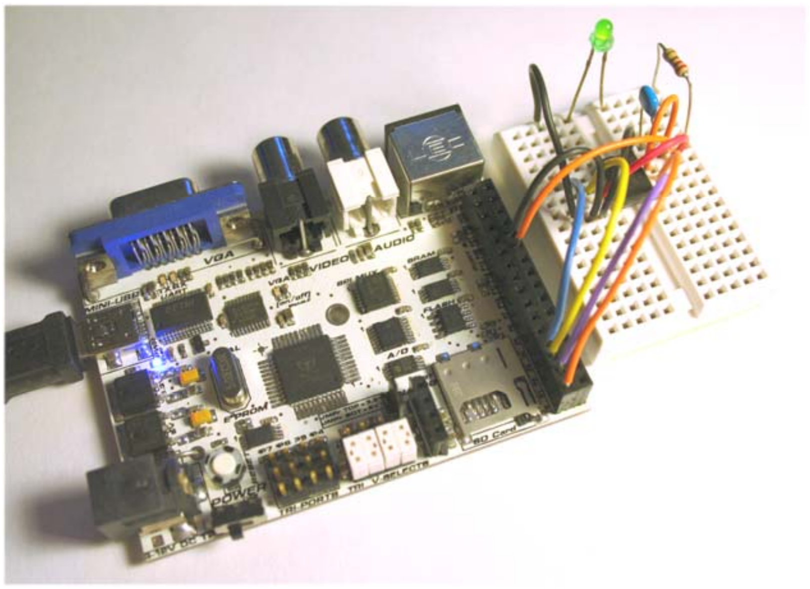


```
SRAM_menu_demo_010
├ FullDuplexSerial_drv_014
├ SRAM_driver_23K256_v010
└ string_util_driver_v010
```

## Connecting the 23K256 SPI SRAM

The first step to getting the demo up and running is to connect a 23K256 SPI SRAM to your Propeller development board. Referring to back to Figure 5, connect the 4 signals from your particular Propeller board's available I/Os to the clock, serial in, serial out, and chip select of the 23K256. Additionally, hook up 3.3 V to Vcc and ground to Vss (along with a 0.1 µF bypass capacitor). Finally, tie *HOLDn* HIGH (3.3 V). Figure 10 below is a photograph of a Propeller C3 platform[2] with a solderless breadboard holding a DIP version of the 23K256.

**Figure 10: The 23K256 Connected to a Propeller C3 for Experimentation**



**NOTE:** The C3 already has a pair of 23K256's onboard, but doesn't mean we can't connect more SPI devices to the external expansion headers.

The code excerpt below from the driver SRAM_driver_23K256_v010.spin shows the circuit diagram as well.

```
{
 Circuit Schematic - To use this driver, simply connect a Microchip 23K256 32K SPI
 SRAM up to your Propeller platform as shown below:

 These signals from the Prop I/O pins map to the SPI interface.

 SPI_CS
 SPI_SCK
 SPI_MOSI
 SPI_MISO                                                    3.3V (VCC)
                                                               ↑
                   Microchip 23K256 32K SRAM                   │
                                                               ┼──────┐
                                    ┌──────────┐               │     ═╪═ 0.1uF (bypass)
        SPI_CS ────────→ CSn│1          8│Vcc ───────┼      ↓
                                    │          │               │     GND
        SPI_MISO ←────── SO │2          7│HOLDn ─────┘
                                    │          │
                         NC │3          6│SCK ←─────── SPI_SCK
                                    │          │
                     ┌── Vss│4          5│SI ←──────── SPI_MOSI
                     │              └──────────┘
                     │
                     │
                     ↓
                    GND

}
```

Once you have the circuit hooked up, set the proper I/O pins in the top-level application SRAM_menu_demo_010.spin to reflect your hardware setup. These constants are located at the top of the `CON` section and set as follows; change these to the respective I/O pins you will be using.

```
  ' SPI pins
  SPI_CS      = 3 ' SPI chip select (active low)
  SPI_SCK     = 0 ' SPI clock from master to all slaves
  SPI_MOSI    = 1 ' SPI master out serial in to slave
  SPI_MISO    = 2 ' SPI master in serial out from slave
```

## Running the Demo

Once you have connected your 23K256 to your Propeller board and updated the I/O pins in SRAM_menu_demo_010.spin, it's time to exercise the SRAM device and see it perform. Compile and download the program to your Propeller board. Then, launch the Propeller Serial Terminal (PST), set it to 38400 baud, N81, set the COM port to the correct value and then press **<Enable>** on the PST.

And nothing should happen. The reason why nothing should happen is that the Propeller already booted a long time ago and printed out the ASCII menu, thus, you need to hit the reset button on your Propeller board to get it to reboot. Now, you should see the menu as depicted in Figure 8.

TIP: Both your Propeller board and the PST use the same serial connection. Remember to first disable the PST, then compile and download to your Propeller board, re-enable the PST and hit reset on your Propeller board to get the menu up, and then you can proceed.

The demo menu program is meant to serve as a template and example platform to see each SRAM function in action. The program relies on the SRAM driver itself, a serial driver, and the text, character parsing object spoke of before. Here's the `OBJ` section for reference:

```
OBJ

 ' drivers and objects required for the demo
 SERIAL      : "FullDuplexserial_drv_014.spin" ' full duplex serial driver
 SRAM        : "SRAM_driver_23K256_v010.spin"  ' 32K Microchip 23K256 SRAM driver
 STR         : "string_util_driver_v010.spin"  ' helpful string and character
                                               ' processing methods
```

The `STR` object is useful—take a look at the source. It has a lot of character and text processing methods based on common C functions, which are very helpful when writing text-based applications.

Moving on, here's the `Main` method where the program begins:

```
PUB Main : status

  ' give system time to initialize...
  STR.Delay_US( 1*1_000_000 )

  ' initialize serial driver (only works if nothing else is on serial port)
  SERIAL.start(31, 30, %0000, 38_400) ' receive pin, transmit pin, baud rate

  ' initialize SPI SRAM driver and save pointer to cache for direct access (optional)
  g_cache_ptr := SRAM.Init( SPI_CS, SPI_SCK, SPI_MOSI, SPI_MISO )
  ' enter infinite test loop
  repeat
    ' clear screen
    SERIAL.tx( $00 )

    ' call the test
    SRAM_Test

' end Main
```

The code starts with a short delay to let the Propeller board settle and the driver(s) start up. Then the call to the SRAM driver's `Init` method is made. This is all you need to do to use the SRAM driver. The call takes the pin numbers assigned to chip select, clock, serial out, and serial and that's it. The method initializes the SRAM driver and SPI I/O interface, then finally starts the cache up. The method returns a pointer to the cache in hub RAM. This is just a convenience in case you need direct access to the byte buffer for some reason. Otherwise, you can ignore the return value.

Next, the screen is cleared by sending a $00 (ASCII VT-100 clear screen) to the Parallax Serial Terminal. Then a call to `SRAM_Test` begins the demo and displays the menu. Since we are going to discuss each menu item and what it does, take a look at the code below. Each menu item is handled by a different case block (highlighted below):

```
PUB SRAM_Test

  '/////////////////////////////////////////////////////////////////////////////
  ' SRAM TEST SUITE /////////////////////////////////////////////////////////////
  ' Each case in the switch statement illustrates an API method(s), use as an
  ' example of how to use the API methods in your own applications to access
  ' the external SPI SRAM.
  '/////////////////////////////////////////////////////////////////////////////

  repeat
    ' draw menu
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )
    SERIAL.txstring( string ("External SPI SRAM Memory Test Menu V1.0 | (c) Parallax 2011") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("1. Read Status Register.") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("2. Write data in format [start,value,numbytes].") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("3. Read data in format [start,numbytes].") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("4. Read byte from cache.") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("5. Write byte to cache.") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("6. Flush the cache to SRAM.") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("7. Enter ASCII text directly into SRAM.") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("8. Print SRAM memory out in ASCII format.") )
    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.tx( ASCII_CR )
    SERIAL.tx( ASCII_LF )

    SERIAL.txstring( string ("Selection?") )

    ' get string from user
    Get_String_Serial(@g_sbuffer, 4 )

    ' convert to integer
    g_key := STR.atoi2(@g_sbuffer, 4)

   ' what is user requesting?
   case g_key
     1: ' READ STATUS REGISTER ----------------------------------------
       SRAM.SRAM_Chip_Select
       g_temp1 := SRAM.SPI_Write_Read( 16, %00000101_00000000, $FF )
       SRAM.SRAM_Chip_Deselect

       SERIAL.tx( ASCII_CR )
       SERIAL.tx( ASCII_LF )
       SERIAL.txstring( string ("Status Register = ") )
       SERIAL.dec( g_temp1 )

     2: ' WRITE DATA ----------------------------------------------------
       SERIAL.tx( ASCII_CR )
       SERIAL.tx( ASCII_LF )

       SERIAL.txstring( string ("Start address?") )

       ' get string from user
       Get_String_Serial(@g_sbuffer, 8 )

       ' convert to integer
       g_temp1 := STR.atoi2(@g_sbuffer, 8)

       SERIAL.tx( ASCII_CR )
       SERIAL.tx( ASCII_LF )

       SERIAL.txstring( string ("Value to write?") )

       ' get string from user
       Get_String_Serial(@g_sbuffer, 8 )

       ' convert to integer
       g_temp2 := STR.atoi2(@g_sbuffer, 8)

       SERIAL.tx( ASCII_CR )
       SERIAL.tx( ASCII_LF )
```

```
      SERIAL.txstring( string ("Number of bytes to write?") )

      ' get string from user
      Get_String_Serial(@g_sbuffer, 8 )

      ' convert to integer
      g_temp3 := STR.atoi2(@g_sbuffer, 8)

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.txstring( string ("Writing Data...") )

      ' write the bytes
      SRAM.SRAM_Fill( g_temp1, g_temp3, g_temp2 )

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.dec ( g_temp3 )
      SERIAL.txstring( string (" Bytes written.") )

    3: ' READ DATA ----------------------------------------------------

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.txstring( string ("Start address?") )

      ' get string from user
      Get_String_Serial(@g_sbuffer, 8 )

      ' convert to integer
      g_temp1 := STR.atoi2(@g_sbuffer, 8)

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )
      SERIAL.txstring( string ("Number of bytes to read?") )

      ' get string from user
      Get_String_Serial(@g_sbuffer, 8 )

      ' convert to integer
      g_temp3 := STR.atoi2(@g_sbuffer, 8)

     ' print buffer to screen
      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )
      SERIAL.txstring( string ("Bytes read:") )

      repeat g_index from g_temp1 to (g_temp1 + g_temp3 - 1)
        ' print starting address to left every 8 bytes
        if ((g_index // 16) == 0)
          SERIAL.tx( ASCII_CR )
          SERIAL.tx( ASCII_LF )

          SERIAL.hex (g_index, 6 )
          SERIAL.txstring( string (":") )

        ' read a single byte at a time
        SRAM.SRAM_Read( g_index, 1, @g_sbuffer[0])

        ' print byte
        SERIAL.hex ( g_sbuffer[0], 2 )
        SERIAL.tx( "," )

    4: ' READ BYTE FROM CACHE ----------------------------------------

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.txstring( string ("Start address?") )

      ' get string from user
      Get_String_Serial(@g_sbuffer, 8 )

      ' convert to integer
      g_temp1 := STR.atoi2(@g_sbuffer, 8)

     ' print buffer to screen
      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.txstring( string ("Data = $") )

      ' read a single byte and print in hex format
      SERIAL.hex( SRAM.CacheR( g_temp1 ), 2 )

    5: ' WRITE BYTE TO CACHE -----------------------------------------
      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.txstring( string ("Start address?") )

      ' get string from user
      Get_String_Serial(@g_sbuffer, 8 )

      ' convert to integer
      g_temp1 := STR.atoi2(@g_sbuffer, 8)

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )
      SERIAL.txstring( string ("Value to write?") )

      ' get string from user
      Get_String_Serial(@g_sbuffer, 8 )

      ' convert to integer
      g_temp2 := STR.atoi2(@g_sbuffer, 8)

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.txstring( string ("Writing Data to Cache...") )

      ' write the byte
      SRAM.CacheW( g_temp1, g_temp2 )

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.txstring( string (" Bytes written.") )

    6: ' FLUSH THE CACHE ---------------------------------------------

      SRAM.SRAM_Cache_Flush
      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )
      SERIAL.txstring( string ("Cache Flushed.") )

    7: ' ENTER ASCII TEXT DIRECTLY INTO SRAM -------------------------

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      SERIAL.txstring( string ("Type Text, CTRL-Z to Save") )

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )
      SERIAL.txstring( string ("TYPE>") )

      ' counts number of bytes written
      g_data2 := 0

      repeat

        ' get next byte from serial port
        if ( (g_data := SERIAL.rx ) == CTRL_Z)
          ' user is done typing
          SERIAL.tx( ASCII_CR )
          SERIAL.tx( ASCII_LF )

          SERIAL.txstring( string ("Number of bytes written to SRAM = "))
          SERIAL.dec( g_data2 )

          ' return to main menu
          quit
        else
          ' print character to serial terminal
          SERIAL.tx( g_data )

          ' store in memory
          SRAM.SRAMW( g_data2++, g_data )

    8: ' PRINT ASCII TEXT FROM SRAM TO SCREEN ------------------------

      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )
      SERIAL.txstring( string ("Number of bytes to ASCII dump from SRAM?") )

      ' get string from user
      Get_String_Serial(@g_sbuffer, 8 )

      ' convert to integer
      g_temp1 := STR.atoi2(@g_sbuffer, 8)

     ' print buffer to screen
      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )
      SERIAL.txstring( string ("ASCII SRAM Dump:") )
      SERIAL.tx( ASCII_CR )
      SERIAL.tx( ASCII_LF )

      ' iterate thru and read and print each value as an ASCII printable
      repeat g_index from 0 to g_temp1-1
        ' read and print byte
        SERIAL.tx ( SRAM.SRAMR( g_index ) )

    ' return to caller
    Other: return
  ' end SRAM_Test
```

Take some time to review each functional block in the case statement. 90% of the code is for display and character entry; the actual SRAM API calls are 1-2 lines usually. But, since this is a user text-based application, a lot of code is needed to get user input, parse, print, etc.

text entry parser method `Get_String_Serial` is local to the demo, located at the end. This method has a little built-in single line editor that can handle backspace and delete, a very useful feature to add to your library of Spin methods.

The menu program works as follows: the menu is printed to the serial terminal, then the `Get_String_Serial` method is used to get the user's selection. This string is converted to an integer and then the appropriate **`case`** is executed based on the value 1..8. In each case block the relevant SRAM method is called after getting the needed parameters from the user. Let's look at the complete list and detail what each does.

**Menu Items**

1. Read Status Register
2. Write data in format [start,value,numbytes]
3. Read data in format [start,numbytes]
4. Read byte from cache
5. Write byte to cache
6. Flush the cache to SRAM
7. Enter ASCII text directly into SRAM
8. Print SRAM memory out in ASCII format

Now, depending on which menu item is called, you will need to enter one or more numeric parameters for the function request. The parser can actually understand decimal, hex ($), and binary (%) formats. So, if you want to enter a number such as 255, you can enter "255", or "$FF", or "%11111111". The parser translates all formats for you with the **`atoi2`** method. Of course, this has nothing to do with SRAM access, but if you are going to write any text applications then parsing and processing are always welcome. Additionally, if you make a mistake while entering values, the **<**Del**>** and **<**Backspace**>** keys work on the line editor. Now, let's review each menu command.

## Menu Commands Explained

**Read Status Register** — This prints out the status of the 23K256 device. It simply sends the get status command to the device and returns it with this single line of code:

```
g_temp1 := SRAM.SPI_Write_Read( 16, %00000101_00000000, $FF )
```

**Write data in format [start,value,numbytes]** — This command requires 3 parameters: the starting address to write at in the SRAM (0..32767), the value to write (0...255), and finally the number of bytes to write. Of course, you can use decimal, hex, or binary numbers as you see fit. The function performs the write operation with the `SRAM_Fill` method:

```
' write the bytes
SRAM.SRAM_Fill( g_temp1, g_temp3, g_temp2 )
```

**Read data in format [start,numbytes]** — Similar to the write command, this one requires a starting address as well as the number of bytes to read. The `SRAM_Read` method is used:

```
' read a single byte at a time
SRAM.SRAM_Read( g_index, 1, @g_sbuffer[0])
```

**Read byte from cache** — This function uses the cache management system to read from the SRAM with the `SRAM_CacheR` method. The function will ask for the starting address to read from and then return the data either from the cache or read it from the SRAM if there is a cache miss.

```
SRAM.CacheR( g_temp1 )
```

**Write byte to cache** — Similar to the read cache function above, you must give this one a starting address as well as the byte to write. The cache is written to, or written through to the SRAM if there is a cache miss.

```
' write the byte
SRAM.CacheW( g_temp1, g_temp2 )
```

**Flush the cache to SRAM** — This function flushes the cache. In other words, it writes the currently cached SRAM page (if there is one) back to the SRAM.

```
' flush the cache
SRAM.SRAM_Cache_Flush
```

These final two menu items allow you to enter text directly into the SRAM via the PST, much like a screen editor. Simply type whatever you want—you can make edits, add carriage returns etc. Whatever characters you type (up to 32,3278 of them) are recorded and written to the SRAM starting from address 0. Then when you print the ASCII text back, the SRAM is simply read from address 0 and printed as ASCII data to the screen. A poor-man's MS Word.

**Enter ASCII text directly into SRAM** — Type anything you wish, up to 32,768 characters. When you are finished press **<**CTRL-Z**>** to save the text in SRAM. Once saved, the demo will print out how many bytes you type; remember this number so you can enter it on the next menu item. There is no particular method call for this; it just uses the standard `SRAM_Write` method.

**Print out SRAM memory in ASCII format** — This prints out the requested number of bytes from the SRAM in ASCII format. Typically, you will use this menu item function after the previous function, so you recall what you typed from the SRAM. The function asks you only how many bytes you want to print out. Again, this relies on no special API function; it just reads the SRAM with `SRAM_Read`.

That's all for the demo. Simply copy and past the setup code into your application, or use this demo as a template for your own experiments to help integrate external SRAM into your next product.

## Optimization Strategies

Before concluding, let's take a few moments to discuss optimization strategies to speed up the external SRAM access. The first obvious optimization is to convert the driver to 100% Propeller Assembly language. This is surely going to make things faster, but it's a lot of work and might not be necessary. A less-aggressive approach would be to try running the Spin version of the SPI SRAM driver on its own cog. Then with a caching system request memory—if it's in the cache then the memory is immediately available; if not, then the primary execution thread can request the page be brought into the cache and return later. This allows the primary thread to continue performing work rather than blocking while a page is brought into main memory.

However, if constant throughput is required then a Propeller Assembly driver is required. The best architecture is to integrate the SPI driver and all SRAM functions (including the cache) into the assembly language driver and then, via shared global memory, use a message-passing scheme to read and write bytes from and to the SPI SRAM.

If you are coding your application in Spin in the first place, Spin's maximum execution velocity will set the pace of your application and dictate how fast you need external memory access. However, if your application is written in assembly language in the first place and speed is of utmost concern, then an assembly language version of the SRAM driver with SPI integrated and a memory mapped shared address space for inter-cog communication is the way to go.

## Summary

This application note has described how to interface an SPI-based external SRAM device to the Propeller chip. A complete driver and API was developed that supported basic read and write operations as well as a caching system for high performance—all in Spin. Using the driver is as easy as including it in the **OBJ** section of your programs and connecting an 8-pin DIP IC up to your Propeller board.

## Resources

The example code below is available in a zip archive for download from this application note's web page: www.parallaxsemiconductor.com/an012

string_util_driver_v010.spin
SRAM_menu_demo_010.spin
SRAM_driver_23K256_v010.spin
FullDuplexSerial_drv_014.spin

## References

1. Joint Electron Device Engineering Council
2. Propeller C3, Parallax #32209; www.parallax.com
3. 23K256 SPI SRAM: http://ww1.microchip.com/downloads/en/DeviceDoc/22100E.pdf
4. Parallax Serial Terminal software is available alone or bundled with the Propeller Tool IDE software: www.parallaxsemiconductor.com/software

## Revision History

Version 1.0: original document.