Version 1.0

HYDRA XTREME 512K MEMORY CARD

PROGRAMMING AND USER MANUAL

Andre' LaMothe

Nurve Networks LLC

HYDRA™ XTREME 512K Memory Card User Manual v1.0

Author
Andre' LaMothe

Editor/Technical Reviewer
The "Collective"

Printing
0001

ISBN
Pending

Licensing, Terms & Conditions

NURVE NETWORKS LLC, . END-USER LICENSE AGREEMENT FOR HYDRA HARDWARE, SOFTWARE , EBOOKS, AND USER MANUALS

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS PRODUCT. IT CONTAINS SOFTWARE, THE USE OF WHICH IS LICENSED BY NURVE NETWORKS LLC, INC., TO ITS CUSTOMERS FOR THEIR USE ONLY AS SET FORTH BELOW. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT USE THE SOFTWARE OR HARDWARE. USING ANY PART OF THE SOFTWARE OR HARDWARE INDICATES THAT YOU ACCEPT THESE TERMS.

GRANT OF LICENSE: NURVE NETWORKS LLC (the "Licensor") grants to you this personal, limited, non-exclusive, non-transferable, non-assignable license solely to use in a single copy of the Licensed Works on a single computer for use by a single concurrent user only, and solely provided that you adhere to all of the terms and conditions of this Agreement. The foregoing is an express limited use license and not an assignment, sale, or other transfer of the Licensed Works or any Intellectual Property Rights of Licensor.

ASSENT: By opening the files and or packaging containing this software and or hardware, you agree that this Agreement is a legally binding and valid contract, agree to abide by the intellectual property laws and all of the terms and conditions of this Agreement, and further agree to take all necessary steps to ensure that the terms and conditions of this Agreement are not violated by any person or entity under your control or in your service.

OWNERSHIP OF SOFTWARE AND HARDWARE: The Licensor and/or its affiliates or subsidiaries own certain rights that may exist from time to time in this or any other jurisdiction, whether foreign or domestic, under patent law, copyright law, publicity rights law, moral rights law, trade secret law, trademark law, unfair competition law or other similar protections, regardless of whether or not such rights or protections are registered or perfected (the "Intellectual Property Rights"), in the computer software and hardware, together with any related documentation (including design, systems and user) and other materials for use in connection with such computer software and hardware in this package (collectively, the "Licensed Works"). ALL INTELLECTUAL PROPERTY RIGHTS IN AND TO THE LICENSED WORKS ARE AND SHALL REMAIN IN LICENSOR.

RESTRICTIONS:
(a) You are expressly prohibited from copying, modifying, merging, selling, leasing, redistributing, assigning, or transferring in any matter, Licensed Works or any portion thereof.
(b) You may make a single copy of software materials within the package or otherwise related to Licensed Works only as required for backup purposes.
(c) You are also expressly prohibited from reverse engineering, decompiling, translating, disassembling, deciphering, decrypting, or otherwise attempting to discover the source code of the Licensed Works as the Licensed Works contain proprietary material of Licensor. You may not otherwise modify, alter, adapt, port, or merge the Licensed Works.
(d) You may not remove, alter, deface, overprint or otherwise obscure Licensor patent, trademark, service mark or copyright notices.
(e) You agree that the Licensed Works will not be shipped, transferred or exported into any other country, or used in any manner prohibited by any government agency or any export laws, restrictions or regulations.
(f) You may not publish or distribute in any form of electronic or printed communication the materials within or otherwise related to Licensed Works, including but not limited to the object code, documentation, help files, examples, and benchmarks.

TERM: This Agreement is effective until terminated. You may terminate this Agreement at any time by uninstalling the Licensed Works and destroying all copies of the Licensed Works both HARDWARE and SOFTWARE. Upon any termination, you agree to uninstall the Licensed Works and return or destroy all copies of the Licensed Works, any accompanying documentation, and all other associated materials.

WARRANTIES AND DISCLAIMER: EXCEPT AS EXPRESSLY PROVIDED OTHERWISE IN A WRITTEN AGREEMENT BETWEEN LICENSOR AND YOU, THE LICENSED WORKS ARE NOW PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THE WARRANTY OF NON-INFRINGEMENT. WITHOUT LIMITING THE FOREGOING, LICENSOR MAKES NO WARRANTY THAT (i) THE LICENSED WORKS WILL MEET YOUR REQUIREMENTS, (ii) THE USE OF THE LICENSED WORKS WILL BE UNINTERRUPTED, TIMELY, SECURE, OR ERROR-FREE, (iii) THE RESULTS THAT MAY BE OBTAINED FROM THE USE OF THE LICENSED WORKS WILL BE ACCURATE OR RELIABLE, (iv) THE QUALITY OF THE LICENSED WORKS WILL MEET YOUR EXPECTATIONS, (v) ANY ERRORS IN THE LICENSED WORKS WILL BE CORRECTED, AND/OR (vi) YOU MAY USE, PRACTICE, EXECUTE, OR ACCESS THE LICENSED WORKS WITHOUT VIOLATING THE INTELLECTUAL PROPERTY RIGHTS OF OTHERS. SOME STATES OR JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY MAY LAST, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IF CALIFORNIA LAW IS NOT HELD TO APPLY TO THIS AGREEMENT FOR ANY REASON, THEN IN JURISDICTIONS WHERE WARRANTIES, GUARANTEES, REPRESENTATIONS, AND/OR CONDITIONS OF ANY TYPE MAY NOT BE DISCLAIMED, ANY SUCH WARRANTY, GUARANTEE, REPRESENATION AND/OR WARRANTY IS: (1) HEREBY LIMITED TO THE PERIOD OF EITHER (A) Five (5) DAYS FROM THE DATE OF OPENING THE PACKAGE CONTAINING THE LICENSED WORKS OR (B) THE SHORTEST PERIOD ALLOWED BY LAW IN THE APPLICABLE JURISDICTION IF A FIVE (5) DAY LIMITATION WOULD BE UNENFORCEABLE; AND (2) LICENSOR'S SOLE LIABILITY FOR ANY BREACH OF ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION, AND/OR CONDITION SHALL BE TO PROVIDE YOU WITH A NEW COPY OF THE LICENSED WORKS. IN NO EVENT SHALL LICENSOR OR ITS SUPPLIERS BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, THOSE RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT LICENSOR HAD BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE LICENSED WORKS. SOME JURISDICTIONS PROHIBIT THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

SEVERABILITY: In the event any provision of this License Agreement is found to be invalid, illegal or unenforceable, the validity, legality and enforceability of any of the remaining provisions shall not in any way be affected or impaired and a valid, legal and enforceable provision of similar intent and economic impact shall be substituted therefore.

ENTIRE AGREEMENT: This License Agreement sets forth the entire understanding and agreement between you and NURVE NETWORKS LLC, supersedes all prior agreements, whether written or oral, with respect to the Software, and may be amended only in a writing signed by both parties.

NURVE NETWORKS LLC
1112 Nancy Lane
San Ramon, CA 94582

Version & Support/Web Site

This document is valid with the following hardware, software and firmware versions:

- HYDRA Game Console Revision A. or greater.
- Propeller Tool 1.0 or greater.

The information herein will usually apply to newer versions but may not apply to older versions. Please contact Nurve Networks LLC for any questions you may have.

Visit **www.xgamestation.com** for downloads, support and access to the XGameStation/HYDRA user community and more!

For technical support, sales, general questions, share feedback, please contact Nurve Networks LLC at:

support@nurve.net / **nurve_help@yahoo.com**

HYDRA XTREME 512K SRAM Card
(HX512)
User Manual

1.0 HYDRA XTREME 512K Card User Manual Overview

Welcome to the user manual for the **HYDRA XTREME 512K SRAM Card** (HX512). This manual covers the design, operation, and programming of the memory card for use with your HYDRA™ Game Console. Please read the entire manual carefully. The following outlines the various topics in the manual for your convenience:

Table of Contents

Figure 1.0 – Product Contents.

1.1 Product Contents

The HYDRA XTREME 512K Card kit consists of the following items as shown in Figure 1.0

1. The HYDRA XTREME 512K card itself.
2. A PC compatible CD-ROM with this document on it.
3. Printed Quick Start Sheet (not shown).

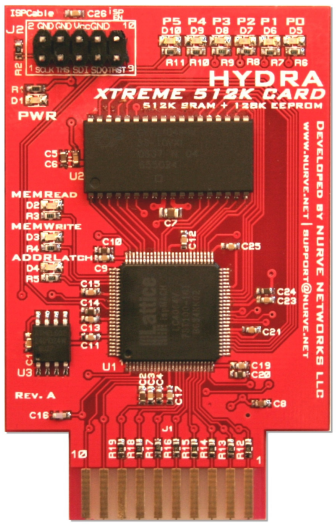
1.1.1 CD-ROM Contents

The CD-ROM contains the documentation, drivers, demos, and all source code for the HX512. Additionally, there is a bonus sub-directory with the latest HYDRA demos and software that are user submitted. The CD-ROM layout is as follows:

```
CD_ROOT:\       README.TXT
                AUTORUN.SYS
                LICENSE.TXT
                SOURCES\
                SCHEMATICS\
                DOCS\
                        \DATASHEETS
                TOOLS\
                GOODIES\
```

Where “CD_ROOT” is your CD-ROM drive letter; “D:”, “E:”, etc.

Figure 2.0 – The HYDRA XTREME 512K Card up Close and Personal..



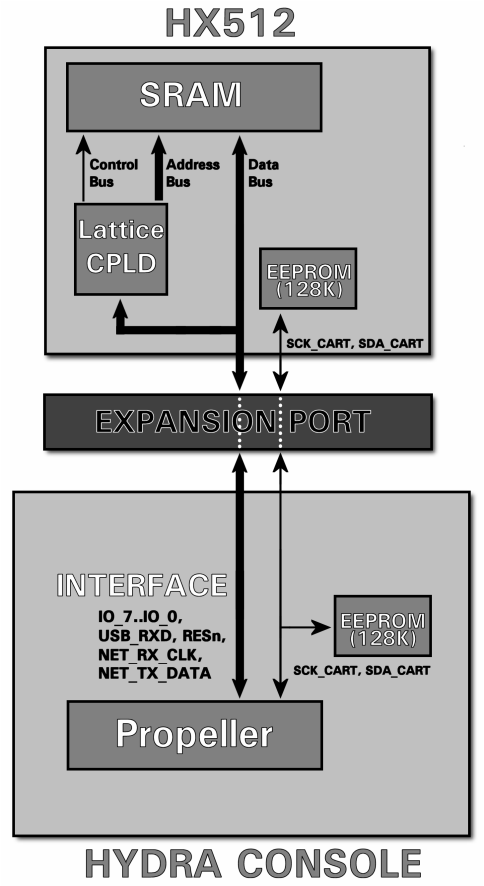
1.2 Introduction and Quick Start

The HYDRA XTREME 512K Card or **"HX512"** for short completes the HYDRA system giving it a full 512K of Static RAM (SRAM) and 128K BYTE EEPROM for program storage (shown in Figure 2.0). Additionally, there is a **Lattice ispMach 4064 C**omplex **P**rogrammable **L**ogic **D**evice (CPLD) onboard which acts as the memory controller and "glue" logic interfacing the HYDRA and the HX512K. The CPLD is used to address the large 512K memory as well as act as a simple memory controller that is capable of auto increment and decrement functionality to help accelerate your code. The HX512 has the following features:

- Supports 512K of SRAM thru a single BYTE wide bus and a handful of control lines.
- Directly addressable lower 64K of SRAM.
- Upper 64-512K accessible thru block reads/writes.
- Programmable post increment/decrement after reads/writes allowing for high speed block access.
- Most operations can be performed in a few ASM instructions.
- SRAM can be accessed as fast as global (HUB) memory in many cases from within a COG.
- Built in 128K EEPROM on board, so firmware can be loaded onto EEPROM rather than HYDRA main board.
- Lots of LEDs for status and debugging uses!

With this exciting addition to your HYDRA, it literally transforms the HYDRA into a full featured 32-bit computer that can host large programs, operating systems, interpreters, compilers, and more advanced games and graphics applications. Additionally, the Lattice CPLD can be re-programmed with an 3rd party programmer or one you build (instructions included in **Appendix C**) . By re-programming the CPLD, you can literally change the ***"personality"*** and behavior of the HX512 altering it to suit your needs as well as using the HX512 as ***"poor mans"*** CPLD development kit. The HX512 has a number of LED indicators on that can be used as indicators to help with debugging your experiments as well. Figure 3.0 shows a block diagram of the HX512 and its relationship to the HYDRA expansions port interface.

Figure 3.0 – HX512 System Diagram.



Referring to Figure 3.0, once the HX512 card is inserted into the HYDRA, the HYDRA will boot off the 128K EEPROM on the card and load the first 32K image into the Propeller. You do not need to use the memory functions if you don't want to; however, the HX512 does multiplex control and clock information on the following interface lines shown in Table 1.0.

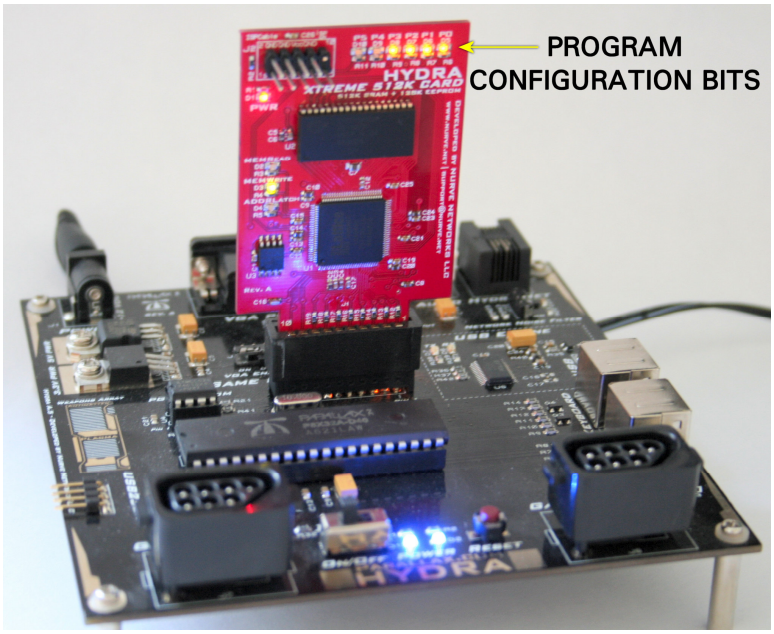
Table 1.0 – Important HX512 Control Lines.

HYDRA Function	*HX512 Function*	*Expansion Interface Pin*	*Propeller Pin*
NET_RX_CLK	SRAM_C0 (control)	10	1
NET_TX_DATA	SRAM_C1 (control)	9	2
USB_RXD	SRAM_CLK (clock strobe)	19	30
RESn	SRAM_RESn (reset)	11	11
IO_0..IO_7	SRAM_D0..SRAM_D7 (data bus)	1..8	16..23

WARNING!	From the HX512's point of view the three control lines above are all inputs, so the HX512 never drives them, thus you won't have a contingency. Nonetheless, once the HYDRA boots and the USB_RXD line is quiet then if its clocked by any COG other than the one communicating with the HX512 then you run the risk of instructing the HX512 to load addresses, read or write which would be unintentional. This is a side-effect since the USB_RXD line doubles for the SRAM_CLK line on the HX512 which is more or less the "execute command" strobe line. Thus, the HX512 when inserted disallows the serial communications from the PC for all intent purposes. This has nothing to do with programming of course, this issue only comes into play ***after*** the programming of the EEPROM or Propeller is complete by the Propeller tool.

Thus, you can still use the HYDRA networking port, but the outgoing serial communications will clock the SRAM if you're not careful. These topics will be discussed in more detail in later sections of the manual, now let's plug the HX512 card in and see if it works!

Figure 4.0(a) – Inserting and Powering up the HX512 Card.



QUICKSTART GUIDE

With the first release of the HX512 card there isn't much software or demos to go along with the card -- only drivers. Nonetheless, you can verify everything is working by inserting the HX512 card into your HYDRA and watching the pre-programmed self-test suite run to confirm the board is working. Make sure your HYDRA is connected to the TV and follow these steps:

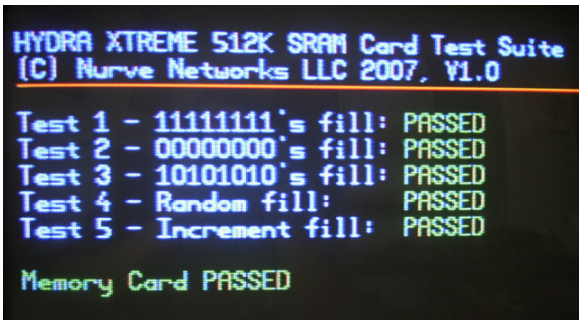
- ✓ **Step 1.** With the HYDRA hooked up to power, TV and the PC, simply insert the HX512 into the expansion slot facing the front of HYDRA, be careful not to force it. The HYDRA can be on or off.

- ✓ **Step 2.** Turn the HYDRA on and/or reset it and the card should reset and boot up the test suite. You will see the top 4 LEDs on the top-right of the card light up as shown in Figure 4.0(a). If the HX512 doesn't boot, leave the power on and simply remove and re-insert the card to get a better insertion connection.

- ✓ **Step 3.** As the tests run, they will display a PASSED/FAIL on the screen. If everything goes well, you should see something like that shown in Figure 4.0(b).

- ✓ **(Optional) Step 4.** Launch the Propeller Tool and load in the test suite program as the top level source file from the CD located in **CD_ROOT:\SOURCES\HX512KSRAM_TEST_010.spin**. Compile and download to the HYDRA by pressing **<F11>** in the Propeller Tool (to program the EEPROM as well), the results should be the same as the pre-loaded test demo on the EEPROM.

Figure 4.0(b) – HX512 Running the Memory Test Suite.

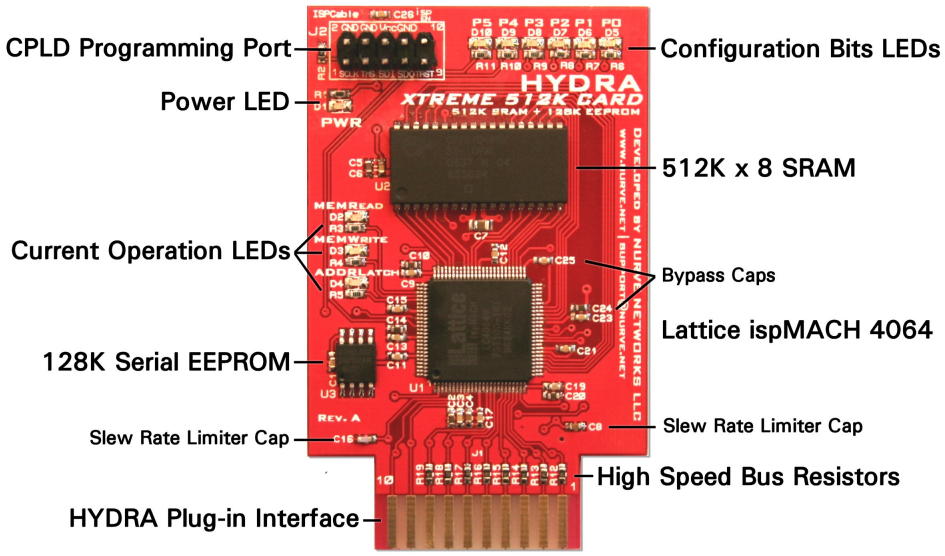


TIP	If you do not see the 4 LEDs at the top right of the HX512 turn on then the card simply didn't boot. Try hitting reset, and/or turn the HYDRA on/off and that should do it. Also, pull the card and re-insert it, you might have a bad insert. All 20 pins needs to make a good connection. Also, sometimes the USB connection to the Propeller gets hung, try removing the USB cable from the HYDRA power the HYDRA down, re-insert the USB, power up.

Once you see the test suite running and you see **"PASSED"** on all tests (which you should) you are in business and everything is working perfectly. You are now ready to start using and programming your HYDRA XTREME 512K Card!

WARNING!	All the chips on the HX512 have static discharge protection; however, when handling the card, make sure to hold it by its edges and please don't press on the large CPLD chip. Additionally, try ***not*** to touch the edge connector. Not only will you potentially static discharge into the connector, but you want to minimize the amount of dirt, oil, and grime you get on the contact fingers.

Figure 5.0 – HYDRA XTREME 512K Card with Detailed Annotation.



1.3 Printed Circuit Board Annotation and I/O Interface Description

Referring to Figure 5.0, the important functional blocks on the PCB are annotated for your reference. Starting at the top left corner of the card is the programming interface (and right under it the RED power LED). This interfaces with a Lattice ISP (In System Programmer) standard 2x5 header which you can build (based on the schematics later in the manual) or buy separately from Lattice Semiconductor at these links:

ISP Download Cables for PCs

http://www.latticesemi.com/products/developmenthardware/programmingcables.cfm

The specific part #'s for the parallel port version and USB version are as follows:

- ispDOWNLOAD Cable Parallel Port (HW-DLN-3C) - $65.00
- ispDOWNLOAD Cable USB (HW-USBN-2A) - $149.00 (recommended if you can afford it)

And here's a link to the online store where you can buy either:

http://www.latticesemi.com/store/hardware.cfm

With the ISP interface on the HX512 you can re-program the CPLD if you wish and change the behavior of the CPLD (of course you need a programmer to do this).

Continuing with the annotation, the top right has (6) LEDs labeled P5..P0. These are the **"configuration or program"** LEDs and indicate the control bits loaded into the HX512K on boot. In this revision of the board only (4) of the control bits are used P3..P0. They indicate the up/down post increment/decrement behavior of the HX512K. This will be discussed in more detail later. For now, just note that P3..P0 all ON mean ***"after each read or write operation automatically increment the address latch by 1".***

The large chip at the top of the board is the asynchronous 512Kx8 static RAM otherwise known as a ***SRAM***. It is a 36-pin SOJ (Small Outline J-leaded) package manufactured by Cypress Semiconductor, Alliance, or ISSI depending on the build batch. Under the SRAM is the Lattice ispMach 4064 CPLD. It's a 100 pin PQFP (Plastic Quad Flat Pack). To its immediate upper left hand corner you should see (3) LEDs, these are "current operation" indicators and indicate memory READ, WRITE, and LATCH operations respectively in real-time (of course if you re-program the CPLD then they are just lights).

Finally, to the immediate bottom left of the PCB is a 128Kx8 serial (I^2C) EEPROM. This EEPROM overrides the HYDRA's onboard EEPROM when the HX512K card is inserted. Thus, you can deploy your SRAM based applications with firmware on the same card, rather than first inserting the card and then loading the software into the HYDRA's EEPROM. This way you can potentially write games, languages, or other applications and then resell the HX512K card with your application on it. You can contact Nurve Networks LLC at support@nurve.net for bulk pricing of the HX512K cards if you're interested.

The remaining components on the board are mostly bypass/filter capacitors and resistors for dampening high frequency reflections and current limiting allowing the HX512 to run as fast as humanly possible without a single error.

1.3.1 I/O Interface Description

The electrical interface between the HX512K and the HYDRA is facilitated thru the HYDRA's 20-pin expansion interface. The detailed pinouts and relationships are shown back in Table 1.0, the HX512K's interface consists of the following signals in the following detailed groups:

Control

Signal(s): SRAM_C1, SRAM_C0 (Propeller Pin 2,1)

These two lines form a 2-bit control word which selects the operation of the CPLD when clocked. The CPLD then executes the command by communicating with the SRAM; reading, writing, and updating its internal state. The bit encoding are shown in Table 2.0.

Table 2.0 – SRAM Control Line Bit Encodings.

SRAM_C1	***SRAM_C0***	***Operation***
0	0	Write SRAM
0	1	Read SRAM
1	0	Latch lower 8-bits into 19-bit address counter
1	1	Latch upper 8-bits into 19-bit address counter and zero upper 3-bits.

The SRAM data bus is 8-bits wide, so when you issue a read or write, 8-bits or a single BYTE is transferred at once. However, there isn't enough room on the expansion interface for an address bus, thus the data bus is multiplexed and doubles as the address bus via writing to the memory controller. The 512K SRAM needs 19 address bits to address all 512K. Thus the CPLD has an internal 19-bit address latch/counter, but this address must be latched a BYTE at a time since the expansion interface only has a BYTE wide bus. The controller gives you the ability to directly latch address bits 15..8, and 7..0 at any time. However, the upper 3-bits (18..16) can't be accessed directly. The workaround is that when you write the upper address latch it zeros these bits in its internal buffer. Then through auto-increment or decrement operations you can sequence thru the memory. More on this later in the functional description.

Clock

Signal(s): SRAM_CLK (Propeller Pin 30)

This signal is an active HIGH edge triggered clock that controls the execution of all commands on the HX512K. The HX512K uses a dual edged clocking scheme meaning that things happen on both the rising and falling edges of the clock. Thus, under normal circumstances you hold the clock line low, then when you want to execute a command, you strobe the clock line by bringing it high then low again. The HX512K is much faster than the Propeller chip, so as fast as you can clock it, the HX512K will respond.

TRICK	Although, none of the demos use the following trick, it's possible to use one of the Propeller's counters to toggle the SRAM_CLK line. If the SRAM_C1, SRAM_C0 lines are in read/write mode and the HX512K is set for auto increment/decrement after read/write then this trick can be used to sequence thru memory very quickly with dummy reads for example simply to update the 19-bit internal address counter in the CPLD. Even at 128 MHz, the CPLD will be able to keep up. The SRAM is rated at 10-12 ns, so safely you can access it at 50-80 MHz. However, the fastest the Propeller can ever execute instructions is at about 20 MIPs, so the SRAM is well within operating limits of the Propeller even if you overclock it by a factor of 2.

Data

Signal(s): SRAM_D0..SRAM_D7 (Propeller Pin 16..23)

This is the parallel BYTE wide data bus used by the HX512K. The data lines from the expansion interface run to the SRAM itself as well as into the CPLD. They are bi-directional and care must be taken when reading and writing to the SRAM and the HX512K. In general, the Propeller I/O lines must be set to the correct direction (input or output) before executing the respective read/write commands. But, you can't damage the lines if you drive the buses the wrong way.

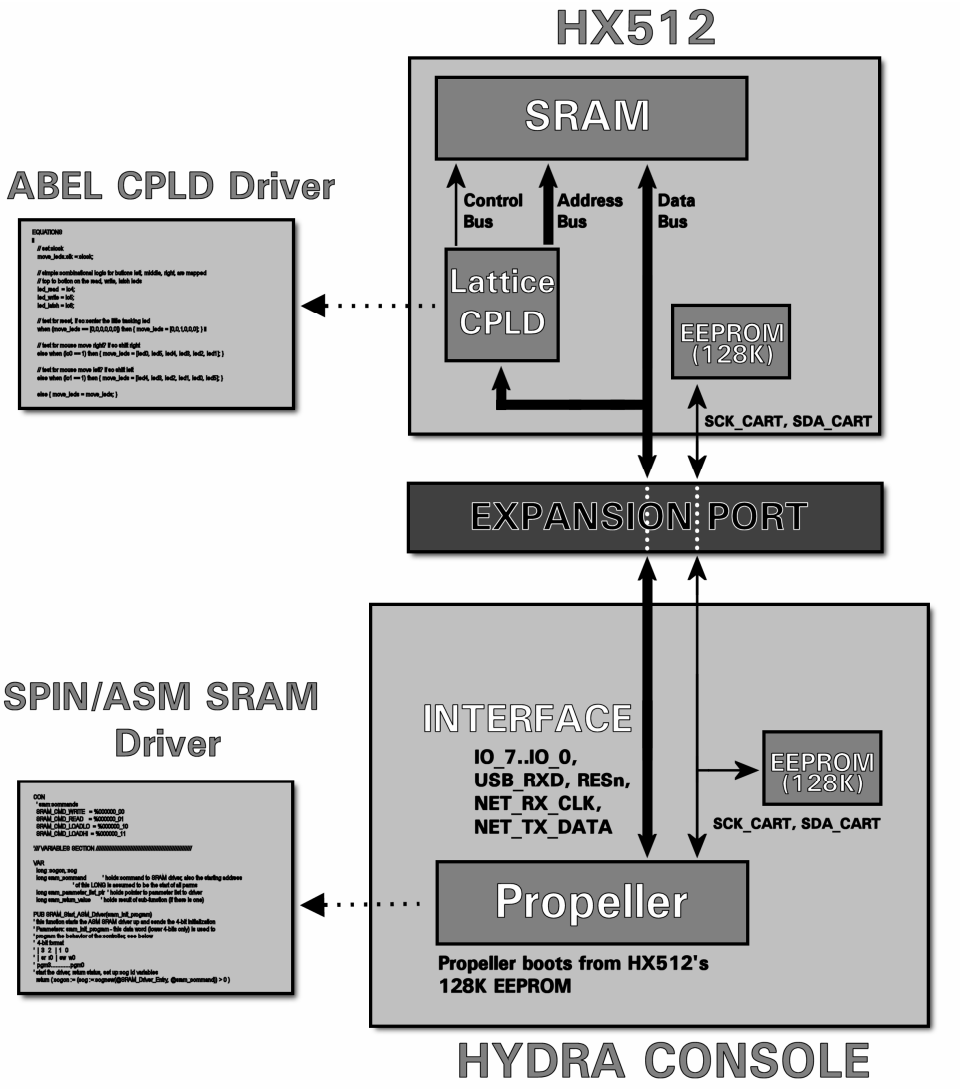
Reset

Signal(s): SRAM_RESn (Propeller Pin 11)

This signal is the standard reset signal that is generated from the USB serial interface as well as the hard reset via the RESET button on the front of the HYDRA. When ***SRAM_RESn*** goes low it resets the HX512K and the CPLD more importantly. Then the CPLD resets to "program" mode and waits for a single BYTE to be placed on the data bus and the

SRAM_CLK to be strobed. Once this is done, this BYTE (lower 4-bits only) becomes the "program" or behavioral control for the CPLD. In this case, all it does is indicate whether or not the address counter in the CPLD should be incremented, decremented or left alone after each read and write operation.

Figure 6.0 – HYDRA XTREME 512K Card Detailed Functional Block Diagram.



1.4 Architectural Description and SRAM Operation

The Propeller chip doesn't have a large onboard RAM memory nor does it have the address space to support one since the Propeller's address space is limited to 64K internally (32K RAM + 32K ROM) and that's that. So the goal of the HX512 was to add a large amount of SRAM to the HYDRA and interface to the Propeller thru the external expansion interface and then come up with a sane method to communicate with the memory, so it integrates well with the Propeller's memory and is as fast as possible for graphics applications. Figure 6.0 shows the final architecture of the HX512 at a block diagram level.

During the design process there were a number of options to accomplish the goals set out for the HX512. An extremely slow serial interface with very few I/O lines could be implemented or an extremely fast interface by using a complete external address, data, and control bus. For a moment, let's analyze the latter option, since serial memory isn't even worth talking about considering the Propeller only runs at 20 MIPs.

Let's assume that you want to interface a 512K x 8 SRAM to the Propeller chip via an external bus. Setting aside how the communication protocol will work on a software level, let's just look at the electrical interface:

Address Bus Requirements:	19 address lines for 512K addresses.
Data Bus Requirements:	8 data lines (bi-directional).
Control Bus Requirements:	3 (Read/Write, Chip Select, Output Enable)
Total:	**30 I/O lines are required**

Of course, you could shave maybe one line off the control bus, with a little logic, so you might be able to pull off a totally parallel 512Kx8 memory interface with 29 lines. If you were just running the Propeller for flat out speed, this is what you could do if you have the I/O capacity, but in our case all we have is the HYDRA expansion port to work with, so another strategy must be derived.

1.4.1 The Memory Controller

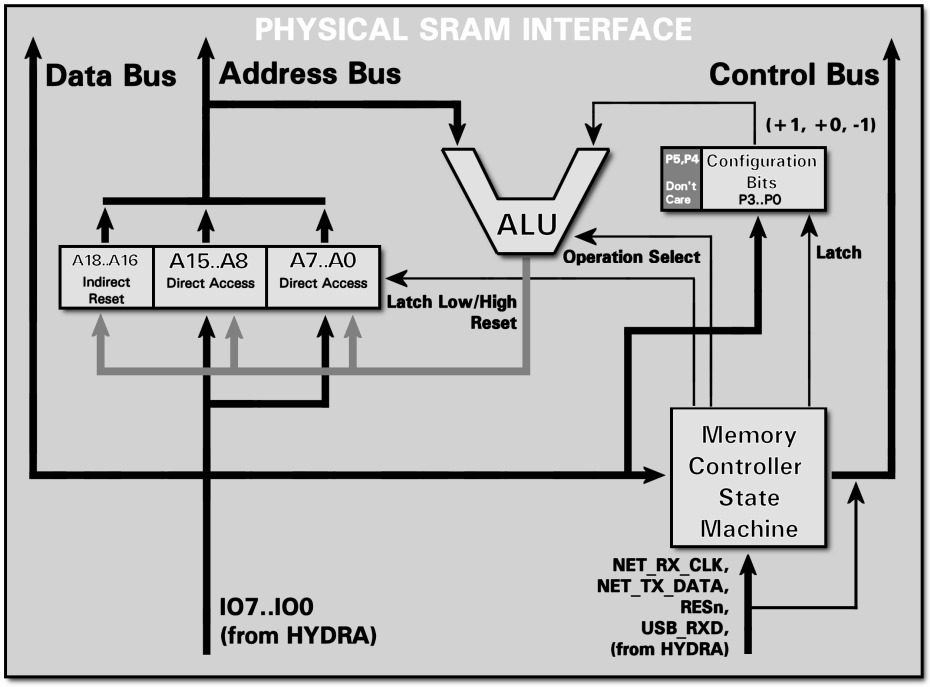
The strategy is to use a **"memory controller"** to interface the Propeller via the expansion port to the actually SRAM memory. In the case of the HX512 this is of course the Lattice ispMACH 4064 which has been programmed to serve this function. The memory controller has a full parallel interface to the SRAM, but the Propeller only has a "command/data/clock" interface with the memory controller. The trick is that inside the memory controller is a full 19-bit address counter that acts as the address bus to the SRAM electrically and logically. Then the Propeller can load the 19-bit address in chunks a BYTE at a time rather than all 19-bits at a time. Of course this isn't as fast as directly controlling all 19-address lines. But, it's not bad if you add some automatic addressing features such as auto increment/decrement after reads and writes anticipating what common memory operations are; write contiguous bytes, read contiguous BYTEs, etc.

Additionally, if you let the memory controller completely control the SRAMs control interface then you can abstract the memory controller's interface from the HYDRA's side to a set of very simple commands (as shown in Table 2.0):

- Read memory
- Write memory
- Latch lower 8-bits (7..0)
- Latch upper 8-bits (15..8) and zero bits (18..16)

So the idea is we need a black box (the Lattice CPLD) that is powerful enough to support the interface from the HYDRA as well as to the SRAM, as well as be able to handle the internal address counter and any kind of automatic behaviors such as post increment/decrement. Figure 7.0 shows a block diagram of the memory controller itself, we will refer to this later in this section, but take a look now for reference.

Figure 7.0 – Block Diagram of the Memory Controller Internally.



With all that in mind then the next decision is how will the interface work? I think you can already tell from the commands listed previously in Table 2.0, we are going to take the approach where we have an address pointer in the memory controller that is used to read or write from. When we want to manually write to the pointer, we must somehow update all 19-bits of it. There are a number of strategies to do this, but the method that gives the most flexibility and the most speed is the one chosen for the HX512. Of course, if you don't like it, then you can always change it by re-programming the CPLD.

As shown in Figure 7.0, the memory controller is rather simple from a block diagram perspective. It has a 19-bit address latch which is accessed via two 8-bit ports addressing bits 15..8 and 7..0 respectively. Also, when the upper 8-bits 15...8 are written the controller always resets the upper 3-bits 18..16 to 0. That said, there is some extra interesting logic on the address latch/counter that is shown in the block diagram and this is of course the post increment/decrement logic.

When the memory controller boots initially via a reset from the HYDRA, the address pointer is loaded with $0_0000, if you request a read or write operation, the address bus into the SRAM will have $0_0000 on it. Now, however, if you want to address another memory location then the address pointer latch must be updated manually (or via automation). This is done by writing to the lower and upper address bits. However, this only gives you access to the first 64K of memory directly, that is, when you update the lower and upper 8-bits of the address latch the upper 3-bits are inaccessible directly, thus, some kind of automation or logic must be used to gain access to them which is outlined next.

1.4.2 Memory Latching Side Effects

Referring to Figure 7.0, if you write to the ***lower*** 8-bits of the address, that is, the lower 8-bits 7..0 then those bits are updated and nothing else happens. This way you can very quickly modify any BYTE in any 256 BYTE page that is currently addressed by the upper address bits with a single latch.

If however, you update the ***upper*** 8-bit latch, it will update address bits 15..8 with the data you latch then it will automatically zero out bits 18..16 (which you can think of as the 64K segment if you will that is currently addressed).

Then you might ask the question, ***"How can I ever access memory above 64K?"***. This is where the post increment/decrement logic comes into play…

1.4.3 Post Increment / Decrement Logic

The address latch is really a counter as well and can count up or down depending on the initial "program behavior" that is loaded into the memory controller on boot from the HYDRA. The program behavior or control bits/flags consist of a single BYTE that is placed on the data bus immediately after reset. The lower 4-bits of the BYTE are used to indicate the behavior that the memory controller should take after each read and write operation. The encoding of the bits are shown in Figure 8.0.

Figure 8.0 - Memory Controller Program Bits

Bit Number							
7	6	5	4	3	2	1	0
0	0	0	0	sr	r0	sw	w0

Referring to the figure, the program bits are referred to PGM3..PGM0, the upper 4 bits of the program word are unused at this time, and always set to 0. The program bits PGM3..PRM0 tell the memory controller how you want to handle the post operation for read and write. Moreover, both operations are independent. For example, you can set both to do nothing, or you can set increment after read, but decrement after write and so forth. However, in most cases, programming the controller with 00001111_2 which means post increment on both read and write is most commonly used. Each 2-bit pair represents the sign and magnitude of the post operation for read and write respectively. This is shown below:

sw - sign bit for write post increment/decrement (1=add, 0=subtract).
w0 - 1 bit magnitude for write post increment/decrement.

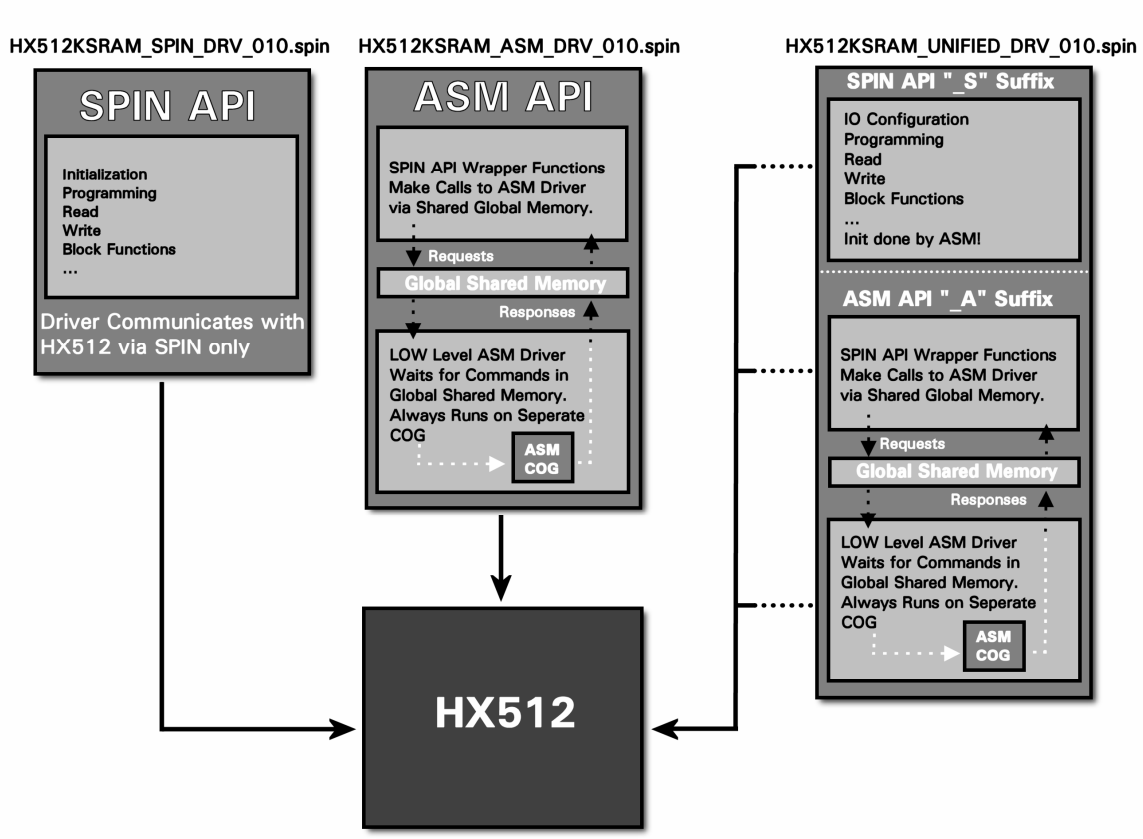
sr - sign bit for read post increment/decrement (1=add, 0=subtract).
r0 - 1 bit magnitude for read post increment/decrement.

Note: that the "sign" bits use "1" to represent addition, and "0" to represent subtraction.

Thus, if you wanted to program the controller to do nothing 0000_0000_2 would work, as would 0000_1010_2, since a "1" in the sign bit will have no effect for a "0" in the magnitude bit. If you wanted the controller to perform a post increment on write, but a post decrement on read then the program bits would be 0000_1101_2. Finally, as the counter increments or decrements it simply wraps around, thus incrementing past 0x3_FFFF (512K-1) results in 0x0_0000, decrementing 0x0_0000 results in 0x3_FFFF (512K-1).

More details about programming later in the API section of the manual.

Figure 9.0 – Graphical Relationship of HX512 Drivers.



1.5 Programming Techniques and Driver API Listing

In this section, the programming of the HX512 will be discussed as well as the API interface explained in detail. Let's begin with outlining the drivers and test programs and what they do, then discuss the entire flow chart of the HX512 booting and accessing memory, and then finally review examples of reading, writing, latching, and finish up with the formal API listing for the HX512 drivers (both SPIN and ASM will be covered). To begin, take a look at Figure 9.0, it depicts the three drivers we have to choose from; SPIN, ASM, and the unified driver (SPASM^{TM}) that has both SPIN and ASM drivers within it. All drivers are located in the **SOURCE** directory on the CD:

- HX512KSRAM_ASM_DRV_010.spin
- HX512KSRAM_SPIN_DRV_010.spin
- HX512KSRAM_UNIFIED_DRV_010.spin

The reason for all three drivers is to give you the maximum amount of flexibility with your code. For example, if the ASM driver does everything you need and speed is your first and foremost concern then you are all set. If on the other hand, you need to be able to make modifications to some of the functions, but aren't an ASM guru then you can use the SPIN only driver if speed isn't a concern. Finally, if you want to modify some SPIN functions, but still want to be able to call the ASM drivers and don't mind the bloat of having both drivers source then the unified driver is the way to go.

Namespace collisions are avoided by appending the suffix "_S" to all the SPIN driver API calls and "_A" to all the ASM driver API calls. Thus, you can have both functions used in the same program, but not name collide at the function name level. If you were to import both drivers at an object level then this wouldn't matter since each object's API functions are resolved with ***"ObjectName.FunctionName"***, thus the ***"ObjectName"*** creates and artificial namespace, but this further suffix syntax gives another level of security, so you don't have to worry about mixing functions in the same source if you like to cut and paste.

The best approach when you are starting to experiment is to use the SPIN or unified driver, since the SPIN code is relatively easy to understand and follow while the ASM driver code is rather complex and highly optimized making it hard to follow and hard to modify for newbie's to ASM.

WARNING!	The one thing you have to keep in mind is that when you use the SPIN driver functions you SPIN code must set up the I/O initially on the Propeller for proper communications since the SPIN code is running on an interpreter loaded into the current COG. Additionally; however, if you start the ASM driver it runs on another COG and thus it also needs to initialize the I/O hardware, so it can access the HX512 as well. Thus, if both drivers run then you have to make sure they can both access the HX512 without bus conflict. Moreover, there is an additional stipulation that only ***one*** driver the SPIN or ASM can initially program the HX512's memory controller program bits. That is, if you use the SPIN driver's call to program the 4-bit configuration program of the memory controller, then you can't use the ASM driver since the ASM driver ***always*** needs to initialize the memory controller itself. Thus, if you want to use SPIN only, you have nothing to worry about, but if you want to use the unified driver, you will always start the ASM driver up and let it initialize the memory controller, you do not want to make another superfluous call to initialize the controller with SPIN code. Not only will it not have any affect, but it might write erroneous data or latch an address to the memory controller unintentionally.

Lastly, the goals of the drivers are simply to be able to access the external memory as if it were internal. Since we can't modify the SPIN interpreter itself and add features to it to transparently access the memory, the interface must be done thru API calls. However, the goal was to make the API calls as painless and easy as possible. And to create an API of the most commonly used functions such as a read BYTE, write BYTE, memory copy, etc. Also, the drivers take advantage of the memory controller's built in ability to auto-increment (or decrement) the current address pointer, so that you can initially set the address once with lengthy latch operations and then as long as you are working with contiguous memory simply read or write without updating the address latch/counter. This saves a huge amount of bus traffic with time sensitive algorithms such as graphics and DSP.

1.5.1 Driver Familiarization: Running the Test Suite

The test program suite runs a series of tests on the HX512 to verify the SRAM, the memory controller and in general to exercise everything on the card to make sure things are in order (even the LEDs). The test program is also a very good tutorial on how to initialize the HX512 as well as make many of the API calls. The test program is located on the CD in the **\SOURCE** directory with the following name:

HX512KSRAM_TEST_010.spin

Simply load it into the Propeller Tool, compile and download to the HYDRA (with the HX512 inserted of course) and after downloading, the Propeller will reset and the program will begin testing the HX512. You should see the exact same results as you did with the pre-loaded test program.

Take a few moments to peruse the code, you will notice in the ***OBJ*** section of the code, the following objects are included:

```
OBJ
  tv    : "tv_drv_010.spin"                ' instantiate a tv object
  gr    : "graphics_drv_010.spin"          ' instantiate a graphics object
  sr    : "HX512KSRAM_ASM_DRV_010.spin"    ' instantiate ASM SRAM driver
```

The only entry of interest is the HX512 driver itself (highlighted) which in this case is the ASM only driver. The ASM driver is very fast and necessary for the test suite since using the SPIN driver would take minutes to hours for each of the tests to run!

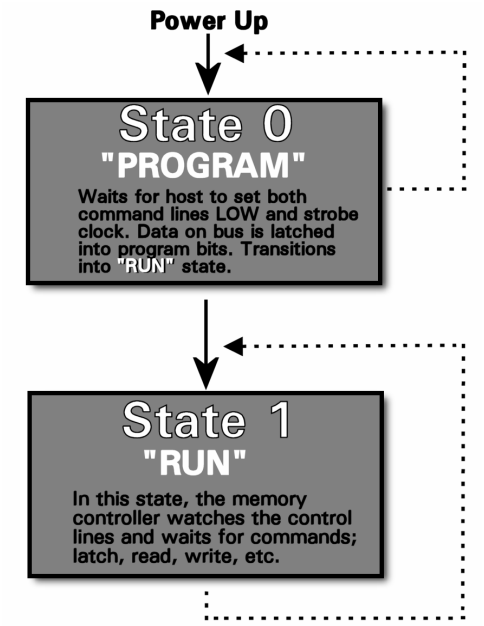
Immediately following the object declaration the main program starts, graphics, and various other variables are initialized and finally on line 160 roughly the call to start the ASM driver up is made with this line:

```
  sr.SRAM_Start_ASM_Driver(%000_11_11)
```

This calls a SPIN function which passes the 4-bit program code to the ASM driver among other things (which we will get into shortly). Once the call is made, another COG is started with the ASM driver and you are free to make calls to it via a simple SPIN API client interface. The API interface more or less passes variables into a global shared memory that the ASM driver is always watching for commands (more on this later) and the ASM driver responds to these commands, thus the ASM driver waits for data in a certain memory location and when something is there, the driver acts. You do not need to interface with the ASM driver with SPIN, you can use another ASM program as well, since the interface is all thru global shared memory, but to make things easier a SPIN interface was elected.

After the HX512 is initialized and ready to go the test suite begins and each test runs in sequence, when the tests are complete they repeat, and that's it.

Figure 10.0 – HX512's Memory Controller State Machine.



1.5.2 HX512 Start-Up Behavior and Program Configuration Bits Programming

Referring to Figure 10.0, when the HX512 is first reset by the HYDRA the CPLD is in the **"PROGRAM"** state. In this mode, the memory controller is waiting to be programmed/configured and sits in a loop. The memory controller accepts the first piece of data written to storing it into the internal program/configuration bits then transitions into the normal **"RUN"** state. Once in the **RUN** state you can issue the normal commands like read, write, load low latch and load high latch.

The trick is to make sure that the memory controller doesn't accidentally get programmed during reset since many of the I/O lines are potentially pulsed with spurious data, thus the memory controller's logic and CPLD program will only respond to a binary "00" on ***(SRAM_C1, SRAM_C0)***, thus you must place both of these lines into output mode, set the data bus I/O to outputs as well as the ***SRAM_CLK*** to an output. During all this I/O juggling, the HX512 shouldn't be erroneously clocked and will remain in the **"PROGRAM"** state until you actually ***want*** to program it. Thus drivers aside, to initialize the HX512 after a ***reset*** (hard or soft) if you were going to code from scratch you would need to following these steps:

Step 1: Set the I/O lines for (SRAM_C1, SRAM_C0), SRAM_CLK, and all the data lines (msb) (SRAM_D7..SRAM_D0) (lsb) all to ***outputs*** while making sure that the output values were set to 0.

Step 2: Write binary 00 to (SRAM_C1, SRAM_C0).

Step 3: Place the 8-bit data on the data bus lines (SRAM_D7..SRAM_D0), only the lower 4-bits matter which map to (PGM3..PGM0); however, always set the upper nibble to binary 0000 just to be safe.

Step 4: Strobe the clock line as follows; SRAM_CLK = 0, SRAM_CLK = 1, SRAM_CLK = 0.

Step 5: ***(Optional)*** Place the data bus back into read or input mode. This is necessary since if another processor wants to read from the data bus, if the interpreter running on the current COG sets the I/O to output then no other COG can set it to input! Thus, always exit functions by setting the data bus to an input. The control lines SRAM_C1, SRAM_C0, SRAM_CLK are ***always*** outputs, so they don't matter as much, leave them as outputs. Of course, this optional step only matters if more than one COG will try to access the HX512.

After performing all 5 steps, the memory controller will transition from the **PROGRAM** state to the **RUN** state and you will see the 4-bit configuration word you just loaded into the program bits reflected on the program LEDs to the top right of the card (in most cases you should write binary 0000_1111_2 which means post increment after read and write).

TIP	Do not get ***"programming the CPLD"*** with ***"programming/configuring the memory controller"*** confused. Programming the CPLD means using a Lattice compatible programmer, writing Verilog or ABEL code, compiling it with a silicon compiler, and flashing it to the CPLD (which we will discuss later). Programming the memory controller or setting the configuration bits on the other hand is completely abstract and simply the action of writing a single BYTE to the controller where the lower 4-bits represent the "behavior" to be performed after reads and writes.

That completes the start up behavior of the HX512. The most important thing to remember is that when the HYDRA re-boots (caused by a reset, power cycling, or the PC downloading code), the HX512 starts in the **PROGRAM** state waiting for a single BYTE to be written. The BYTE must be placed on the data bus, the ***SRAM_CLK*** line strobed LOW-HIGH-LOW and that completes the initialization process, at this point, the HX512 memory controller transitions into the **RUN** state.

After the initial programming/configuration operation, the 19-bit address latch/counter will be set to all zero's ($0_0000) and the memory controller is ready to go and awaits your command(s).

1.5.3 SPIN Driver API Overview

The SPIN driver API is a good segue into the ASM driver since we can focus on functionality rather than how the code works since the ASM is much harder to follow. However, as you will see as the drivers are discussed the functionality of the drivers are nearly identical, and literally only differ in a suffix to differentiate the function calls from an API perspective. Other than that internally they more or less do the same, the ASM drivers just do it hundreds of times faster! So, we will start with the SPIN driver then finish with the ASM driver.

The SPIN driver consists of a little over a dozen function calls. The calls give you the maximum flexibility in communicating with the HX512 to facilitate basic SRAM operations. Of course, the drivers aren't as fast as the ASM drivers, so you won't be doing crazy bitmapped graphics with them since SPIN runs in the thousands or tens of thousands of lines of code a second, not millions of instructions per second needed to do graphics. For higher speed access you will use the ASM drivers (or write your own). But, starting with the SPIN drivers is better since we can look inside functions and see what's going on. The driver source is listed in **Appendix E.**, so we won't cover it here in detail, just the API itself from a functional view; however, some of the constants and data structures are worth reviewing in the header section of the driver to give you a frame of reference for the API listing.

The code begins with a ***CON*** section with the following constants:

```
CON

  ' SRAM bus interface pin constants
  SRAM_CTRL_0 = 1 ' NET_RX_CLK       (expansion pin 10)
  SRAM_CTRL_1 = 2 ' NET_TX_DATA      (expansion pin 9)

  SRAM_STROBE = 30 ' USB_RXD (Prop TX ----> USB_RXD Host) (expansion pin 19)

  SRAM_IO_7 = 23 ' IO_7 (pin 28)
  SRAM_IO_6 = 22
  SRAM_IO_5 = 21
  SRAM_IO_4 = 20
  SRAM_IO_3 = 19
  SRAM_IO_2 = 18
  SRAM_IO_1 = 17
  SRAM_IO_0 = 16  ' IO_0 (pin 21)

    ' sram commands
  SRAM_CMD_WRITE   = %000000_00
  SRAM_CMD_READ    = %000000_01
  SRAM_CMD_LOADLO  = %000000_10
  SRAM_CMD_LOADHI  = %000000_11

  ' size of spin based local memory cache in bytes used for sram copying etc. to speed up process
  LOCAL_MEM_CACHE_SIZE = 256
```

These constants identify the I/O port pins and the SRAM command codes for ease of programming. If you were to interface the HX512 to non-HYDRA hardware, then here's where you would want to make changes to the I/O interface pins.

Also, notice the constant highlighted at the end of the code ***LOCAL_MEM_CACHE_SIZE*** this is the size of the local buffer for memory copies with both source and destination inside the SRAM itself. Since the SRAM is a BYTE device, it becomes very inefficient to move large amounts of memory around within the SRAM itself a BYTE at a time, a better strategy would be to copy blocks or pages of memory back to the Propeller then move the entire block at a time. With a larger CPLD it would possible to actually perform memory copies with the CPLD itself acting as a DMA controller, but this CPLD just doesn't have enough logic to pull that off.

Moving on, the next section contains a few global working variables for general SRAM state tracking and debugging:

```
VAR

  ' SRAM Spin driver working variables
  long sram_addr
  byte sram_ctrl
  byte sram_data
```

Finally, at the end of the source listing is the memory cache storage itself. The cache didn't need to be defined as data statements, but could have easily been modeled with an array up top in the ***VAR*** section; however, with this format, you can initialize the data and values can be placed into the cache for special purposes and testing.

```
DAT

' local memory cache used for spin SRAM functions when local buffer is needed (256 bytes for now)
local_mem_cache         byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                        byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                        byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                        byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                        byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                        byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                        byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                        byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

1.5.5 SPIN Driver API Listing

The SPIN API has five major classes of function as shown in Table 3.0. Along with each class of function are the associated functions listed to the right. After the table, the API functions will be listed one by one along with a short example.

Table 3.0 – SPIN Driver API Function Classes at a Glance.

Functional Class	*Function(s)*
Initialization	SRAM_InitalizeIO_S
	SRAM_WriteControl_S(_data8)
Latching	SRAM_LoadAddr64K_S (_addr16)
	SRAM_LoadAddr512K_S (_addr19)
	SRAM_LoadAddrLow_S (_addr8)
	SRAM_LoadAddrHi_S (_addr8)
Reading	SRAM_Read64K_S(_addr16)
	SRAM_ReadAuto_S
Writing	SRAM_Write64K_S (_addr16, _data8)
	SRAM_WriteAuto_S (_data8)
Block Operations	SRAM_MemSet_S(_dest_addr, _data8, _num_bytes)
	SRAM_MemCopy_S(_dest_addr, _src_addr, _num_bytes)
	MM_Copyto_SRAM_S(_dest_addr19, _src_addr16, _num_bytes)
	SRAM_Copyto_MM_S(_dest_addr16, _src_addr19, _num_bytes)

Note: All functions have the suffix "_S" to help separate them from the ASM versions in the case that source modules are mixed.

Function Prototype: SRAM_InitalizeIO_S

Description: Initializes the COG's I/O in preparation for interfacing to the HX512. Must be called first before making any other SPIN API calls.

Parameters: None.

Example(s): Initialize I/O for COG running interpreter to interface with HX512.

```
OBJ
  sr    : "HX512KSRAM_SPIN_DRV_010.spin"     ' instantiate spin SRAM driver

pub start
  ' initialize the IO for this COG, so its setup for the SPIN driver<-> HX512 interface
  SRAM_InitializeIO_S
```

Function Prototype: SRAM_WriteControl_S(_data8)

Description: Writes the configuration bits to the HX512's memory controller. This function must be called before any API functions are called. Additionally, if the ASM driver is used in conjunction with the SPIN driver either at the source level or with the unified driver then do not call this function, rather, let the ASM driver initialization call handle the configuration of the HX512.

Parameters: **_data8** : 8-bit data that configures the memory controller's post read/write behavior. Only bits 3..0 are used, bits 7..4 are ignored and should be set to 0's. The format of the bits are as follows:

Bit Number							
7	6	5	4	3	2	1	0
x	x	x	x	sr	r0	sw	w0

sw - sign bit for write post increment/decrement (1=add, 0=subtract).
w0 - 1 bit magnitude for write post increment/decrement.

sr - sign bit for read post increment/decrement (1=add, 0=subtract).
r0 - 1 bit magnitude for read post increment/decrement.

Example(s): Configure HX512 to increment after read and increment after write (most common configuration).

```
OBJ
  sr    : "HX512KSRAM_SPIN_DRV_010.spin"     ' instantiate spin SRAM driver

pub start
  ' initialize the IO for this COG, so its setup for the SPIN driver<-> HX512 interface
  sr.SRAM_InitializeIO_S

  ' now configure HX512
  sr.SRAM_WriteControl_S(%0000_1111)
```

Function Prototype: ***SRAM_LoadAddr64K_S (_addr16)***

Description: Loads the HX512's address latch/counter's lower 16-bits with the sent address (zero's the upper 3-bits). This function is the fastest way to set an arbitrary address when accessing only the first 64K of the SRAM.

Parameters: **_addr16** : 16-bit address to set the HX512's internal address counter to.

Example(s): Load the address of $1FF0 into the HX512 (assumes HX512 has been initialized etc.)

```
' set HX512's address to $1FF0
sr.SRAM_LoadAddr64K_S($1FF0)
```

Function Prototype: ***SRAM_LoadAddr512K_S (_addr19)***

Description: Loads the HX512's address latch/counter's full 19-bit address with the lower 19-bits of the sent address. This function assumes that the memory is in post read increment mode since the function achieves the final target address by "walking" to it via dummy reads. Thus, it's very slow the farther out in memory you go. The ASM version is much faster of course. The bottom line is random access beyond 64K is very slow. But, random access in the first 64K is very fast. Additionally, the function is smart enough to use fast direct access to the lower 64K, but there is conditional logic to test for this and since SPIN is interpreted any extra cycles accessing memory that can be avoided should. Hence, if you only need access to the lower 64K make sure to use the 64K version of the function even though this one ***will*** work.

Parameters: **_addr19** : 19-bit address to set the HX512's internal address counter to.

Example(s): Load the address of $10000 into the HX512 (assumes HX512 has been initialized etc.)

```
' set HX512's address to $10000 (128K)
sr.SRAM_LoadAddr512K_S($10000)
```

Function Prototype: SRAM_LoadAddrLow_S (_addr8)

Description: Loads the lower 8-bits of the HX512's address latch/counter's the lower 8-bits of the sent address. This function is good if you want to jump around in a 256 BYTE "page" that is already defined by the upper address bits, but you don't need/want to update the entire 16-bit address (since it costs two latch operations). Bits [a7..a0] are updated while bits [a18..a8] are left untouched.

Parameters: **_addr8** : 8-bit address to set the HX512's lower 8-bits of the internal address counter to.

Example(s): Walk thru addresses 0-255 of the currently address "page" defined by the upper 10-bits of the address latch by only changing the lower 8-bits (assumes HX512 has been initialized etc.)

```
' walk thru the 256 bytes on the current page defined by address latch [a18..a8 | xxxxxxxx ]
repeat index from 0 to 255
  sr.SRAM_LoadAddrLow_S(index)

' doesn't do much since there is no read or write operation!
```

Function Prototype: SRAM_LoadAddrHi_S (_addr8)

Description: Loads the upper 8-bits of the HX512's address latch/counter (bits 15..8) with the lower 8-bits of the sent address. This function is good if you want to jump around accessing the same BYTE in multiple "pages", but you don't need to update the entire 16-bit address (since it costs two latch operations).

Parameters: **_addr8** : 8-bit address to set the HX512's upper 8-bits of the internal address counter to, bits [a15..a8] are updated, bits [a7..a0] are left untouched while bits [a18..a16] are zeroed.

Example(s): Access the same BYTE in 256 pages by only changing the upper 8-bits (assumes HX512 has been initialized etc.)

```
' walk thru the same byte on 256 pages, updates latch [000 | xxxxxxxx | a7..a0 ]
repeat page from 0 to 255
  sr.SRAM_LoadAddrHi_S(page)

' doesn't do much since there is no read or write operation!
```

Function Prototype: SRAM_Read64K_S(_addr16)

Description: Reads a single BYTE from the lower 64K address space of the HX512. Is very fast for random access. After the BYTE is read and returned to caller the HX512 post increments or decrements the internal address pointer (or does nothing) depending on the configuration bits.

Parameters: **_addr16** : 16-bit address of BYTE to read from HX512.

Example(s): Find the integer average of the first 1000 BYTEs in the HX512.

```
sum := 0

' sum data up
repeat index from 0 to 999
  sum += sr.SRAM_Read64K_S(index)

' compute average
sum /= 1000
```

Function Prototype: SRAM_ReadAuto_S

Description: Reads a single BYTE from the HX512's current address. The idea of this function is to first set the address latch of the HX512 with a call to another function then very quickly iterate thru memory without the need for sending the latch address each time. However, for this function to be useful the HX512's configuration bits should be programmed for post increment or decrement after read operations.

Parameters: None (implied as the current address).

Example(s): Find the integer average of the 1024 bytes from address $2000 to $23FF in the HX512.

```
sum := 0

' first set address latch to starting address
sr.SRAM_LoadAddr64K_S($2000)

' sum data up
repeat index from 0 to 1023
  sum += sr.SRAM_ReadAuto_S

' compute average
sum /= 1024
```

Function Prototype: SRAM_Write64K_S(_addr16, _data8)

Description: Writes a single BYTE from to the lower 64K address space of the HX512. Is very fast for random access write. After the BYTE is written the HX512 post increments or decrements the internal address pointer (or does nothing) depending on the configuration bits.

Parameters: **_addr16** : 16-bit address to write BYTE to.
_data8 : 8-bit data BYTE to write (simply uses the lower 8-bits of 32-bit parameter).

Example(s): Write the integers 0..255 starting at address $8000.

```
value := 0

' write bytes
repeat addr from $8000 to $80FF
  sr.SRAM_Write64K_S(addr, value++)
```

Function Prototype: SRAM_WriteAuto_S(_data8)

Description: Writes a single BYTE to the HX512's current address. The idea of this function is to first set the address latch of the HX512 with a call to another function then very quickly iterate thru memory without the need for sending the latch address each time. However, for this function to be useful the HX512's configuration bits should be programmed for post increment or decrement after write operations.

Parameters: **_data8** : 8-bit data BYTE to write (simply uses the lower 8-bits of 32-bit parameter).

Example(s): Write the integers 255..0 starting at address $8000.

```
value := 0

' first set address latch to starting address
sr.SRAM_LoadAddr64K_S($8000)

' write bytes with auto increment after write
repeat value from 255 to 0
  sr.SRAM_WriteAuto_S(value)
```

Function Prototype: ***SRAM_MemSet_S(_dest_addr, _data8, _num_bytes)***

Description: Sets a number of BYTEs in the SRAM to a specific value. Similar to *memset()* in C/C++. Assumes configuration bits set for auto increment.

Parameters:
_dest_addr : 19-bit destination address in SRAM to start BYTE fill.
_data8 : 8-bit data BYTE to write (simply uses the lower 8-bits of 32-bit parameter).
_num_bytes : Number of BYTEs to set/fill, 0..512K.

Example(s): Fill the first 20K BYTEs of memory with $57.

```
sr.SRAM_MemSet_S($0_0000, $57, 20*1024)
```

Function Prototype: ***SRAM_MemCopy_S(_dest_addr, _src_addr, _num_bytes)***

Description: Copies a number of BYTEs internally in SRAM from one location to another. This function uses a local memory buffer within the Propeller to "cache" blocks of memory to speed up the process. The cache is defined in the ***CON*** section and the ***DAT*** section and is recommended to be at least 128 BYTEs in size. Similar to ***memcpy()*** in C/C++. Doesn't handle overlapping copies properly unless the number of BYTEs copied is smaller than the cache size then an overlapping copy will work since the data is cached all at once and not destroyed during the copy process. Assumes configuration bits set for auto increment.

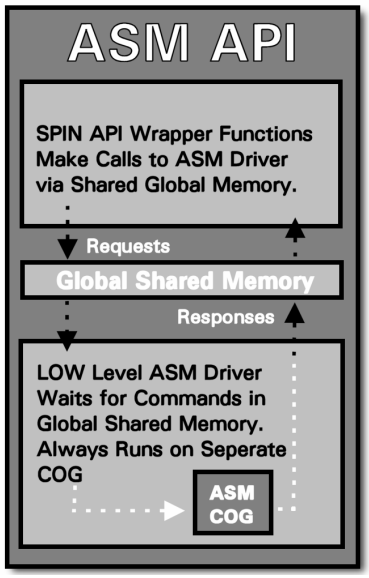
Parameters:
_dest_addr : 19-bit destination address in SRAM to copy BYTEs to.
_src_addr : 19-bit source address in SRAM to copy BYTEs from.
_num_bytes : Number of BYTEs to copy 0..512K.

Example(s): Copy the first 64K in SRAM to destination address $2_0000 (128K) in SRAM.

```
sr.SRAM_MemCopy_S($2_0000, $0_0000, 64*1024)
```

This concludes the SPIN API overview, next the ASM API will be discussed. In most cases, its nearly identical, only the start up and the addition of a couple more functions are where it differs. Also, the data flow in the ASM driver from SPIN client to the driver is interesting and will be discussed.

Figure 11.0 – ASM Driver Communication Architecture.



1.5.6 ASM Driver API Overview

The ASM driver is much faster than the SPIN driver; hundreds of times faster in fact. The driver's architecture and communications interplay is shown in Figure 11.0. When writing code for the Propeller where you want SPIN to interact with ASM there is a problem of communicating with the ASM driver once its running. The problem is once you launch the ASM driver on another COG there is no direct way to send "messages" to the COG. Thus, more creative shared memory strategies much be employed. This is the common technique used in multiprocessor architectures that do not have message passing mechanisms. You will find that all the drivers written in ASM that need interplay from SPIN (or ASM for that matter) use this architecture which is from the 1950's! In any event, the HX512's ASM driver is no exception. The driver is located on the CD in the **\SOURCES** directory with the following filename:

CD_ROOT:\SOURCES\HX512KSRAM_ASM_DRV_010.spin

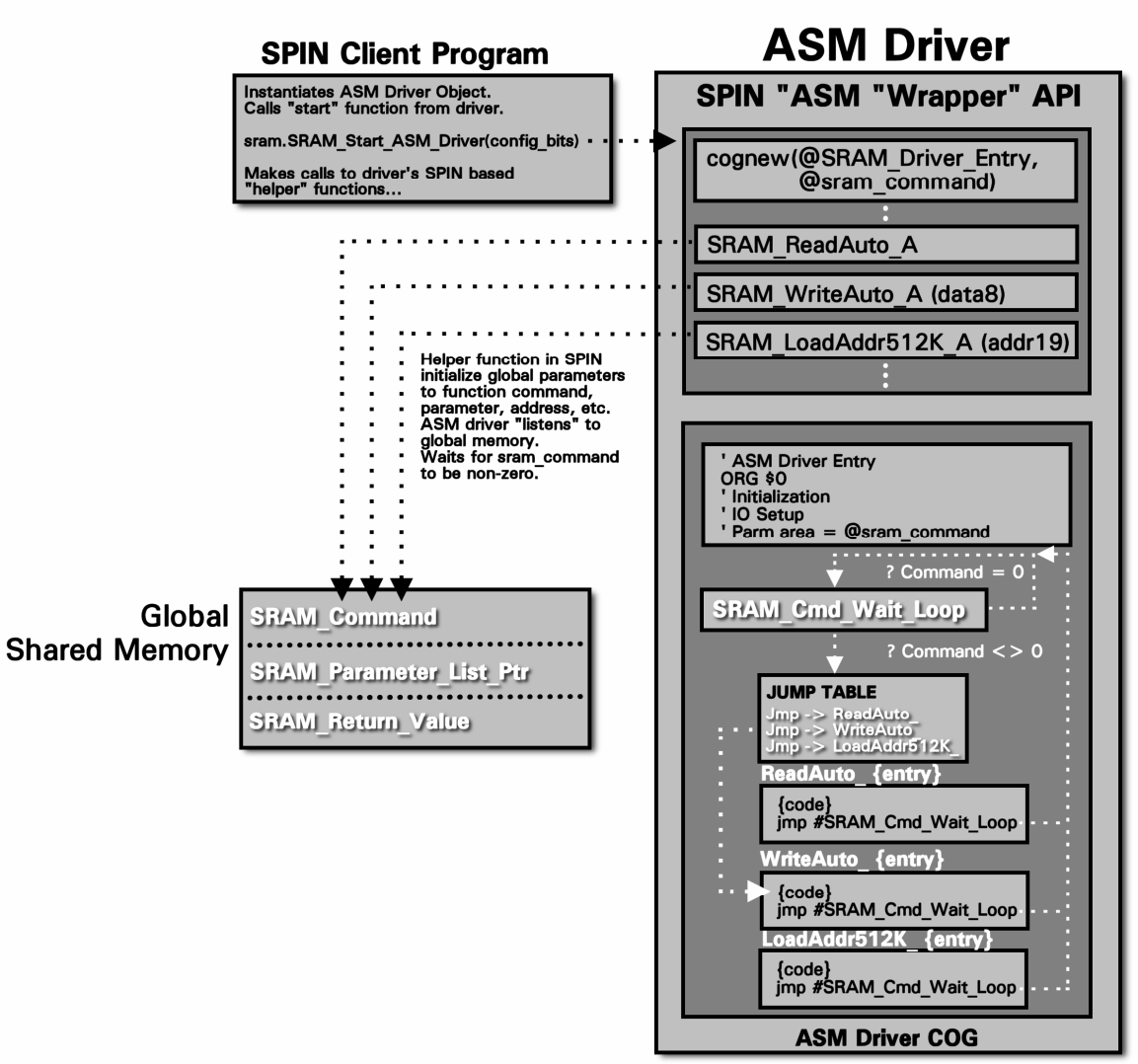
Referring to Figure 11.0, the client's SPIN code running on COG(n) simply makes a call to the ASM driver's start method which is listed below:

```
PUB SRAM_Start_ASM_Driver(sram_init_program)
' this function starts the ASM SRAM driver up and sends the 4-bit initialization program code to it as well as initializes the IO pins for
' proper operation the control word or "program" instructs the SRAM controller to either post inc/dec on reads/write or neither

' Parameters: sram_init_program - this data word (lower 4-bits only) is used to program the behavior of the controller, see below
'
'
'  4-bit format
'
'  |  3   2   |  1   0
'  |  sr  r0  |  sw  w0
'  pgm3...........pgm0
'
'  sw   - sign bit for write post increment/decrement (1=add, 0=subtract).
'  w0   - 1 bit magnitude for write post increment/decrement (w1 ignored in this version).
'
'  sr   - sign bit for read post increment/decrement  (1=add, 0=subtract).
'  r0   - 1 bit magnitude for read post increment/decrement (r1 ignored in this version).
'
'
'  In most cases, its recommended that controller is initialized with both post increment on read/write
'  which is the value %0000_1111, these program bits will show up on the LEDs to the top right of the SRAM card

  ' if the driver is running kill it, however, there is no way to reset the controller, so the program loaded into the program
  ' from RESET will remain there until another reset
  SRAM_Stop_ASM_Driver

  ' set command in global shared variable
  sram_command            := sram_init_program
  ' set starting address of parameters passed to sub-functions, this is a pointer to pointer, in this case NULL since this operation
  ' has no parms
  sram_parameter_list_ptr := 0

  ' start the driver, return status, set up cog id variables
  return ( cogon := (cog := cognew(@SRAM_Driver_Entry, @sram_command)) > 0 )
```

The start method makes a couple global memory assignments (which we will get to shortly) and simply makes a call to ***cognew(…)*** and starts the ASM driver running. The only parameter sent to ***cognew(…)*** is the starting address of the

global parameter passing area. This is the key to making the communication scheme work. The SPIN interpreter running on the master COG needs a way to get to the ASM driver in real-time. Thus, a set of variables are defined in the global memory space, so the SPIN helper functions can access the memory while at the same time the starting address of this region is sent to the ASM driver, thus both parties the client and server know where data will be passed.

Figure 12.0 – The SPIN / ASM Driver Functional Call Interaction Diagram.



1.5.7 The ASM Driver's Global Memory Passing Area

Conceptually how SPIN communicates with the ASM driver is straight forward. A number of parameters are defined in SPIN at the top of the driver program in the VAR section:

```
VAR

  long  cogon, cog

  ' sram parameter passing area for ASM driver, this starting address always holds, the command, parameter ptr, and return value, in that order
  long sram_command             ' holds command to SRAM driver, also the starting address of this LONG is assumed to be the start of all parms
  long sram_parameter_list_ptr  ' holds pointer to parameter list to driver
  long sram_return_value        ' holds result of sub-function (if there is one)
```

The parameters consist of 3 LONGs; the first is always the command, the second a pointer to the parameters, and the third is used as a place holder for return values. Thus, as long as the ASM driver knows where these are, the ASM driver can "watch" the command variable. When it detects a command loaded into this variable, then it jumps to the requested command, queries the parameters by inspecting the pointer and finally returning any data via the return value.

The entire process is graphically depicted in Figure 12.0. Once the ASM driver is running, it sits in a waiting loop inspecting the global memory location which is supposed to hold the next command for the driver to execute. As long as this command is NULL, the driver does nothing. The ASM code that does this is very short:

```
SRAM_Cmd_Wait_Loop
              ' enter into command loop waiting for command
              rdlong sram_cmd, sram_parms_base_ptr wz ' read command from MM in global shared variable
        if_z  jmp #SRAM_Cmd_Wait_Loop                 ' if non-zero then execute command, else continue to loop
```

Once a non-zero value is detected by the waiting loop, the command is used as an index into a command lookup table and the function requested is vectored to. This code looks like this:

```
             ' ok now we basically need to do case (sram_cmd) and for each value execute the code body
              mov r0, #SRAM_Jump_Table          ' r0 = base address of jump table
              add r0, sram_cmd                  ' r0 = r0 + cmd
              movs :Read_Jumpvec, r0            ' access vector address in jump table at [r0 + cmd] -> destination of jmp (self modify code)
              nop       ' wait a second for pre-fetch, let self modifying code complete downstream

:Read_Jumpvec mov r1, 0 ' dummy 0 value has been overwritten with jump vector above
              jmp r1    ' jump to sub-function starting address

' this is an inline jump table, more or less implements an assembly language "case" statement
SRAM_Jump_Table  ' to save memory convert to words or bytes later, but means more code above to perform select logic
                 ' table holds starting address of each sub-function

              long Wait_           '= 0 , do nothing command (DONE)

              long Write64K_       '= 1 , write byte to lower 64K fast mode (DONE)
              long Read64K_        '= 2 , read byte from lower 64K fast mode (DONE)

              long Write512K_      '= 3 , write byte anywhere in 512K memory, slower (not implemented),
                                   ' instead load the address with LoadAddr512K_ then use ReadAuto/WriteAuto
              long Read512K_       '= 4 , read byte from anywhere in memory, slower  (not implemented)

              long WriteAuto_      '= 5 , writes to the current address sram controller is set to, then auto inc/dec executes if programmed (DONE)
              long ReadAuto_       '= 6 , writes to the current address sram controller is set to, then auto inc/dec executes if programmed (DONE)

              long LoadAddr64K_    '= 7 , loads a 16-bit address (0..64K-1) directly into the low and high address latches,
                                   ' also clears the upper 3-bits of address (DONE)
              long LoadAddr512K_   '= 8 , loads a 19-bit address (0..512K-1) into address buffer, by advancing if necessary using dummy reads (DONE)

              long LoadAddrLow_    '= 9 , loads only the lower 8-bits of address into address buffer (DONE)
              long LoadAddrHi_     '= 10 , load only the uppper 8-bits of address into address buffer,
                                   ' also clears upper 3-bits, good to select 256 bytes "pages" (DONE)

              long MemSet_         '= 11 , fills memory anywhere in the 512K region with a byte value (DONE)
              long MemCopy_        '= 12 , copies a number of bytes in the SRAM from source to destination, doesn't support overlapping copies (DONE)

              long MM_Copyto_SRAM_ '= 13 , copies bytes from the Propeller's main memory to the SRAM's 512K space (DONE)
              long SRAM_Copyto_MM_ '= 14 , copies bytes from the SRAMs 512K to the Propeller's main memory (DONE)

              long ReadAddr_       '= 15 , returns the current value of the 19-bit address buffer in the SRAM controller (not implemented)

              long MemSum_         ' = 16 ' sum a region of memory and returns the 32-bit result, helps with diagnostics and DSP stuff
```

The appropriate sub-function is called and the code executes, upon completion the sub-function clears the global command out and the driver is ready to process another command. As an example, here's the code that loads a 64K address into the HX512:

```
LoadAddr64K_  ' = 7 ' loads a 16-bit address (0..64K-1) directly into the low and high address latches, clears the upper 3-bits of address

              ' this sub-function sets the SRAMs' 16-bit latch, low and high address latches are written to with the sent 64K
              ' parameters: one long, starting address: sram_cmd_parms_ptr
              ' parm 0 (32-bit): address to set latches to (lower 16-bit used)

              ' retrieve long holding 16-bit address
              rdlong sram_parm0, sram_cmd_parms_ptr

              ' call set address routing, exprext sram_parm0 = 16-bit address
              mov r7, sram_parm0
              call #SetAddr64K_Proc
```

```
              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done
```

Now, the final part of the system is the SPIN "wrapper" that sets the call up for you. These are all contained within the source file for the ASM driver as well for your convenience. Moreover, there is a wrapper for every one of the ASM functions to set them up. Continuing with our example, here's the wrapper for the above ASM function:

```
PUB SRAM_LoadAddr64K_A(addr16)
' this function sets the SRAM controllers address latch to the sent 16-bit address, clears the upper 3-bits of the address as well
'
' Parameters: addr16 - 16-bit address to set the SRAM address latch to, upper 3-bits is zero'ed

  sram_parameter_list_ptr := @addr16
  sram_command            := _LoadAddr64K ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

  return sram_return_value
```

The wrapper is trivial, more or less just automates the setup of the command and parameter pointer for you. Of course, you do not need to use the wrappers. You are free to call the driver yourself from you own code directly just by assigning the command global to the appropriate function number, but before this ***always*** make sure the parameters are set ***first*** since the instant you set the command the ASM driver will execute thus the parameters need to be set up before the command is issued. The commands are listed in the jump vector table code above and in the driver source itself of course (which is listed for convenience in the appendices).

1.5.8 ASM Driver API Listing

The ASM API has six major classes of function as shown in Table 4.0. Along with each class of function are the associated functions listed to the right. After the table, the API functions will be listed one by one along with a short example.

Table 4.0 – ASM Driver API Function Classes at a Glance.

Functional Class	*Function(s)*
Initialization	SRAM_Start_ASM_Driver(sram_init_program)
	SRAM_Stop_ASM_Driver
Latching	SRAM_LoadAddr64K_A (addr16)
	SRAM_LoadAddr512K_A (addr19)
	SRAM_LoadAddrLow_A (addr8)
	SRAM_LoadAddrHi_A (addr8)
Reading	SRAM_Read64K_A (addr16)
	SRAM_ReadAuto_A
Writing	SRAM_Write64K_A (addr16, data8)
	SRAM_WriteAuto_A (data8)
Block Operations	SRAM_MemSet_A (dest_addr, _data8, num_bytes)
	SRAM_MemCopy_A (dest_addr, _src_addr, num_bytes)
	MM_Copyto_SRAM_A (dest_addr19, src_addr16, num_bytes)
	SRAM_Copyto_MM_A (dest_addr16, src_addr19, num_bytes)
DSP Operations	SRAM_MemSum_A (addr19, num_bytes)

Note: All functions have the suffix "_A" to help separate them from the SPIN versions in the case that source modules are mixed.

Function Prototype: ***SRAM_Start_ASM_Driver(sram_init_program)***

Description: Starts another COG with the ASM driver and initializes I/O on that COG for interoperability with the HX512 and finally writes the configuration bits to the HX512's memory controller. This function must be called before any API functions are called. Additionally, if the ASM driver is used in conjunction with the SPIN driver either at the source level or with the unified driver then do not call the SPIN function ***SRAM_WriteControl_S*****(..)**, rather, let the ASM driver initialization call handle the configuration of the HX512.

Parameters: sram_init_program : 8-bit data that configures the memory controller's post read/write behavior. Only bits 3..0 are used, bits 7..4 are ignored and should be set to 0's. The format of the bits are as follows:

Bit Number							
7	6	5	4	3	2	1	0
x	x	x	x	sr	r0	sw	w0

sw - sign bit for write post increment/decrement (1=add, 0=subtract).
w0 - 1 bit magnitude for write post increment/decrement.

sr - sign bit for read post increment/decrement (1=add, 0=subtract).
r0 - 1 bit magnitude for read post increment/decrement.

Example(s): Start the ASM driver and Configure HX512 to increment after read and increment after write (most common configuration).

```
OBJ
  sr    : "HX512KSRAM_ASM_DRV_010.spin"      ' instantiate ASM SRAM driver

pub start

  ‘ start ASM drier, configure I/O and configure HX512 all in one shot
  sr.SRAM_Start_ASM_Driver(%0000_1111)
```

Function Prototype: ***SRAM_LoadAddr64K_A (_addr16)***

Description: Loads the HX512's address latch/counter's lower 16-bits with the sent address (zero's the upper 3-bits). This function is the fastest way to set an arbitrary address when accessing only the first 64K of the SRAM.

Parameters: **addr16** : 16-bit address to set the HX512's internal address counter to.

Example(s): Load the address of $1FF0 into the HX512 (assumes HX512 has been initialized etc.)

```
‘ set HX512’s address to $1FF0
sr.SRAM_LoadAddr64K_A($1FF0)
```

Function Prototype: ***SRAM_LoadAddr512K_A (_addr19)***

Description: Loads the HX512's address latch/counter's full 19-bit address with the lower 19-bits of the sent address. This function assumes that the memory is in post read increment mode since the function achieves the final target address by "walking" to it via dummy reads. Thus, this is slower that accessing the first 64K. However, the ASM version is much faster of course. The bottom line is random access beyond 64K is very slow. But, random access in the first 64K is very fast.

Parameters: **addr19** : 19-bit address to set the HX512's internal address counter to.

Example(s): Load the address of $10000 into the HX512 (assumes HX512 has been initialized etc.)

```
‘ set HX512’s address to $10000 (128K)
sr.SRAM_LoadAddr512K_A($10000)
```

Function Prototype: ***SRAM_LoadAddrLow_A (addr8)***

Description: Loads the lower 8-bits of the HX512's address latch/counter with the lower 8-bits of the sent address. This function is good if you want to jump around in a 256 BYTE "page" that is already defined by the upper address bits, but you don't need to update the entire 16-bit address (since it costs two latch operations).

Parameters: **addr8** : 8-bit address to set the HX512's lower 8-bits of the internal address counter to, bits [a7..a0] are updated, bits [a18..a8] are left untouched.

Example(s): Walk through addresses 0-255 of the currently address "page" defined by the upper 10-bits of the address latch by only changing the lower 8-bits (assumes HX512 has been initialized etc.)

```
‘ walk thru the 256 bytes on the current page defined by address latch [a18..a8 | xxxxxxxx ]
repeat index from 0 to 255
  sr.SRAM_LoadAddrLow_A(index)

‘ doesn’t do much since there is no read or write operation!
```

Function Prototype: ***SRAM_LoadAddrHi_A (addr8)***

Description: Loads the upper 8-bits of the HX512's address latch/counter (bits 15..8) with the lower 8-bits of the sent address. This function is good if you want to jump around accessing the same BYTE in multiple "pages", but you don't need to update the entire 16-bit address (since it costs two latch operations).

Parameters: **addr8** : 8-bit address to set the HX512's upper 8-bits of the internal address counter to, bits [a15..a8] are updated, bits [a7..a0] are left untouched while bits [a18..a16] are zero'ed.

Example(s): Access the same byte in 256 pages by only changing the upper 8-bits (assumes HX512 has been initialized etc.)

```
‘ walk thru the same byte on 256 pages, updates latch [000 | xxxxxxxx | a7..a0 ]
repeat page from 0 to 255
  sr.SRAM_LoadAddrHi_A(page)

‘ doesn’t do much since there is no read or write operation!
```

Function Prototype: ***SRAM_Read64K_A (addr16)***

Description: Reads a single BYTE from the lower 64K address space of the HX512. Is very fast for random access. After the BYTE is read and returned to caller the HX512 post increments or decrements the internal address pointer (or does nothing) depending on the configuration bits.

Parameters: **addr16** : 16-bit address of BYTE to read from HX512.

Example(s): Find the integer average of the first 1000 BYTEs in the HX512.

```
sum := 0

‘ sum data up
repeat index from 0 to 999
  sum += sr.SRAM_Read64K_A(index)
```

```
' compute average
sum /= 1000
```

Function Prototype: ***SRAM_ReadAuto_A***

Description: Reads a single BYTE from the HX512's current address. The idea of this function is to first set the address latch of the HX512 with a call to another function then very quickly iterate thru memory without the need for sending the latch address each time. However, for this function to be useful the HX512's configuration bits should have programmed it for post increment or decrement after read operations.

Parameters: None (implied as the current address).

Example(s): Find the integer average of the 1024 bytes from address $2000 to $23FF in the HX512.

```
sum := 0

' first set address latch to starting address
sr.SRAM_LoadAddr64K_A($2000)

' sum data up
repeat index from 0 to 1023
  sum += sr.SRAM_ReadAuto_A

' compute average
sum /= 1024
```

Function Prototype: ***SRAM_Write64K_A (addr16, data8)***

Description: Writes a single BYTE from to the lower 64K address space of the HX512. Is very fast for random access write. After the BYTE is written the HX512 post increments or decrements the internal address pointer (or does nothing) depending on the configuration bits.

Parameters: **addr16** : 16-bit address to write BYTE to.
data8 : 8-bit data BYTE to write (simply uses the lower 8-bits of 32-bit parameter).

Example(s): Write the integers 0..255 starting at address $8000.

```
value := 0

' write bytes
repeat addr from $8000 to $80FF
  sr.SRAM_Write64K_A(addr, value++)
```

Function Prototype: ***SRAM_WriteAuto_A (data8)***

Description: Writes a single BYTE to the HX512's current address. The idea of this function is to first set the address latch of the HX512 with a call to another function then very quickly iterate thru memory without the need for sending the latch address each time. However, for this function to be useful the HX512's configuration bits should have programmed it for post increment or decrement after write operations.

Parameters: **data8** : 8-bit data BYTE to write (simply uses the lower 8-bits of 32-bit parameter).

Example(s): Write the integers 255..0 starting at address $8000.

```
value := 0

' first set address latch to starting address
sr.SRAM_LoadAddr64K_A($8000)

' write bytes with auto increment after write
```

```
repeat value from 255 to 0
  sr.SRAM_WriteAuto_A(value)
```

Function Prototype: ***SRAM_MemSet_A (dest_addr, data8, num_bytes)***

Description: Sets a number of BYTEs in the SRAM to a specific value. Similar to *memset()* in C/C++.

Parameters: **_dest_addr** : 19-bit destination address in SRAM to start BYTE fill.
_data8 : 8-bit data BYTE to write (simply uses the lower 8-bits of 32-bit parameter).
_num_bytes : Number of BYTEs to set/fill, 0..512K.

Example(s): Fill the first 20K BYTEs of memory with $57.

```
sr.SRAM_MemSet_A($0_0000, $57, 20*1024)
```

Function Prototype: ***SRAM_MemCopy_A (dest_addr, src_addr, num_bytes)***

Description: Copies a number of BYTEs internally in SRAM from one location to another. This function uses a local memory buffer within COG's 512 LONG memory to "cache" blocks of memory to speed up the process. The cache is defined in the CON section and the DAT section and is recommended to be at least 128 BYTEs in size. Similar to *memcpy()* in C/C++. Doesn't handle overlapping copies properly unless the number of BYTEs copied is smaller than the cache size then an overlapping copy will work since the data is cached all at once and not destroyed during the copy process. Note: The cache right now is nearly as large as it can be since it eats into program space and there is only 512 LONGs for the program. If you find you need more cache size then comment out functions in the driver you don't need and increase the size of the cache to suit your needs.

Parameters: **dest_addr** : 19-bit destination address in SRAM to copy BYTEs to.
src_addr : 19-bit source address in SRAM to copy BYTEs from.
num_bytes : Number of BYTEs to set/fill, 0..512K.

Example(s): Copy the first 64K in SRAM to destination address $2_0000 (128K) in SRAM.

```
sr.SRAM_MemCopy_A($2_0000, $0_0000, 64*1024)
```

Function Prototype: ***SRAM_MemSum_A (addr19, num_bytes)***

Description: Performs a summation on a contiguous block of positive 8-bit integers and returns the 32-bit positive sum. Good for fast summation of memory blocks for DSP and graphics algorithms. Shows what can be done with the HX512 on the driver side, other useful functions would be a MAC (Multiply and Accumulate) function and a large vector dot product etc. This function is a model of how to implement them in ASM.

Parameters: **addr19** : 19-bit source address in SRAM of BYTEs to sum.
num_bytes : Number of BYTEs to sum.

Returns: 32-bit positive sum.

Example(s): Sum the entire 512K memory up.

```
sr.SRAM_MemSum_A($0_0000, 512*1024)
```

1.5.9 Working with the Unified Driver API

The unified driver contains both the ASM and SPIN drivers at the source level, thus the ASM helper functions can name collide with the SPIN function, hence, the use of the suffixes "_S" and "_A". The unified driver is in the **\SOURCES** directory with the name:

CD_ROOT:\SOURCES\HX512KSRAM_UNIFIED_DRV_010.spin

As noted previously the driver is more or less a source file level merge and is identical in functionality to the SPIN and ASM drivers, thus refer to the ASM and SPIN driver APIs for the interfaces. The only thing about using the driver is that you must ***not*** call both of the HX512 configuration calls from the unified driver. The proper way to initialize the unified driver is as follows:

First, you need to import the object and create an instance of it:

```
OBJ
  sr    : "HX512KSRAM_UNIFIED_DRV_010.spin" ' instantiate unified SRAM driver
```

Next, you need to initialize the I/O for the COG that the interpreter is running on ***and*** at start up the ASM driver which in-turn configures the HX512 for both drivers:

```
  ' initialize current COG's IO as not to conflict with ASM SRAM driver's (if its started)
  sr.SRAM_InitalizeIO_S

  ' initialize ASM driver version of SRAM controller
  ‘ (will work with calls to SPIN version as long as above call is made to SPIN driver)
  sr.SRAM_Start_ASM_Driver(%000_11_11)
```

Then you are ready to make calls to either the SPIN or ASM API.

WARNING!	There is the potential for conflict if you make a call to the ASM driver with your own code that doesn't wait for the command to clear. For example, you may be tempted to write you own ASM wrapper functions, but if you forget to ***wait*** for the command global to clear then the following series of events can wreak disaster; you call the ASM driver to do something that takes a long time, like sum 512K, then immediately after you make a call to the SPIN API, but the ASM function hasn't finished! Thus, always make sure when you set a command in the global for the ASM driver you ***block*** on the global until the driver has consumed the global and cleared the variable out. This way you know the ASM function is complete.

1.6 Advanced Programming Concepts and Graphics

The HX512 gives the HYDRA enormous capabilities, but to take full advantage of them one must use assembly language to access the SRAM at rates that are comparable with the Propeller's internal 32K shared memory. The first thing to accept is no external memory will ever run as fast as in the internal memory since the sheer act of communicating with any external device via the I/O lines takes extra steps as does the extraction of the data itself from the I/O read. Thus, the best one can hope for is to make each memory access only a few instructions, thus considering that any COG can only access main memory every 16 clocks due to the HUB rotation then its reasonable to think that with very tight loops and control using 3-5 instructions to access an external memory is almost as fast as accessing global memory from a COG in assembly language.

For example, when the HX512 is in post increment read, reading BYTEs of memory consists of nothing more than strobing the clock line to the HX512 (assuming that the address pointer is already set to the starting location of the memory read). However, after every pulse, the I/O lines still must be read, so the general steps to read a BYTE is are:

Step 1: Set the clock strobe line HIGH.

Step 2: Read the data on the data bus.

Step 3: Set the clock strobe line LOW.

Step 4: (Optional) Read another BYTE; GOTO Step 1.

This is about as fast as is possible. The setting of clock bit can be done by logically OR'ing the I/O, the reading of the data bus can be done with a simple read, and finally the resetting of the clock strobe can be done with a logical AND'ing of a mask with I/O. So 3 ASM instructions can in theory achieve all this with the right set up. However, the challenge is extracting the data from the 32-bit I/O read which requires shifting and masking (2-3 instructions), and so forth. Thus, the actual reading of data is very fast, but the data has to be extracted and positioned to work with.

On the other hand, with proper algorithms and data structures, one can design the code, so that the least amount of post processing to massage the data into position is needed. This way, a short code fragment of 3-4 instructions can be used to read data from the HX512. Considering this, in some cases with proper planning, the HX512 can be as fast as the internal 32K global memory as accessed from a COG.

Considering this, all kinds of possibilities are available. Even if the SRAM was 2x as slow as the internal 32K to access, it doesn't matter since more than one COG can be used to process the data and read the data if need be.

One of the most exciting possibilities (which will be included in the next software release of the HX512) is full bitmapped graphics drivers using the HX512 as a frame buffer. Frame buffers are linear regions of memory used to hold bitmapped graphics, the HX512 has more than enough room to hold large high resolution NTSC or even VGA frame buffers. The question is can the HX512 be accessed fast enough to render the graphics in real-time? The answer is absolutely! Here's an example.

Let's say we want to implement the standard 2-bit per pixel graphics mode that the Propeller chip is used to working with, thus one 32-bit value represents 16 pixels on the screen. With a mode like this and a resolution of 256x192, the memory requirements are:

256x192 * 2 bits per pixel = 12,288 BYTEs per frame buffer.

Assuming, two frame buffer's; one for active display buffer and one for the inactive back buffer that is being rendered into. Then the memory requirements are 2x12,288 = 24,576 BYTEs which easily fits within the first 64K of the HX512 (which is the fastest to access randomly).

Now, the next question is can we access the HX512 fast enough to feed the Propeller's VSU in real-time? Let's generalize a bit and assume that we want 256 pixels during the active scan of the screen which is roughly 52.5uS (the total NTSC is 64.5uS roughly), hence, that means that for every pixel we need to access that pixel at the following rate:

52.5uS / 256 = 205 nS.

NOTE	Astute readers will realize that the color burst bandwidth is 3.58 MHz; therefore, driving the NTSC at anything higher than that which is 279 nS is superfluous. The color transitions will take 1-2 color clocks no matter what, thus the highest number of unique colored pixels during the active scan is only 160 roughly. Nonetheless, we can still drive the NTSC at 256 pixels if we wish, it just ignores many of them and there will be "color artifacting". The point is that this example is an extreme case and in reality the memory requirements and bandwidth will be less.

And we already decided that we need 3-4 instructions to read a BYTE, so at system clock 80 MHZ that translates to, 20 MIPs for COG (at 4 clocks per instruction), or each instruction takes 50 ns. Thus, assuming a 4 instruction average to read a BYTE from the external HX512 it will take the following amount of time:

(4 instructions per read) * (50 ns per instruction) = 200 nS per BYTE read from external memory.

However, each BYTE read doesn't have one, but 4 pixels, since we are using 2-bits per pixel which is 4 pixels per packed BYTE. Alas, we can read data at 400% the required bandwidth to successfully pull off the external frame buffer.

Of course, this is ideal, so we might cut our estimate in half and say that we can read data at 2x the rate. Thus, we can expand our bitmap driver to 4-bits per pixel and still sustain the 256x192, but in 16-colors if we wished! These rough calculations should show you that's it's possible indeed to use the HX512 as a frame buffer. Moreover, if we loosen the resolution requirements up to 160-224 pixels per line and stick with 2-4 bits per pixel, there will be no problem with writing a single COG driver without jumping thru many optimization hoops.

1.7 Re-programming the HX512's CPLD (Complex Programmable Logic Device)

One of the features of the HX512 is the ability to re-program the host CPLD to give the HX512 a completely different ***"personality"*** or to enhance the current functionality. The Lattice ispMach 4064 is based on FLASH memory technology, so you can re-program the chip as long as you have a programmer to do so. The "program" you download to the chip describes a fuse map of sorts that controls the interplay and interconnection of all the logic elements and I/O within the chip. The chip itself is nothing more than a 2D array of similar "logic blocks". The logic blocks are generic computational elements that when wired together can create very complex logical structures. Typically these "blocks" consist of one or more flip flops, feedback networks, and a lookup table. The bigger brother of the CPLD is the **FPGA** or **F**ield **P**rogrammable **G**ate **A**rray. FPGAs typically need external program EEPROMs and are many times larger than CPLDs. But, the idea is the same; use a generic array of logic blocks to implement any complex logical functions in the hardware.

If you have never used a CPLD or FPGA then you might be asking, "How do you program them?". Well, they are programmed in what's called a ***Hardware Description Language*** or HDL. HDLs allow a programmer to describe the high level functionality of a digital circuit (as well as its structure if needed), thus one can code something like:

output_1 = (input1 AND input2)

Now, looking at this, we have no idea what the final structural implementation of this will be nor do we care. As the hardware engineer all we care about is the functionality of the description. Additionally, there are many HDLs just like with software programming there are "camps" and "factions" that think one is better than the other. The two most popular HDLs are **Verilog** and **VHDL**. Verilog is similar to C, VHDL is similar to Ada (if you have ever heard of it, probably not if you're under 25 years old). Also, there are offshoots of Verilog called **SystemC** and **SystemVerilog** which are much more advanced and feature rich.

Here's an example of some Verilog code that tests two push buttons and if one is on activates an LED:

Verilog Code to Test Buttons and Activate an LED:

```
Module Let_There_Be_Light( button1, button2, led1)

input button1, button2; // each is a 1-bit input
output led1;            // 1-bit output

// assign the ouput a logical combination of the inputs
assign led1 = button1 || button2;

endmodule
```

VHDL Code to Test Buttons and Activate an LED:

```
LIBRARY IEEE;                          -- use this library
USE IEEE.STD_LOGIC_1164.all;

-- define the I/O structure of the design; inputs, outputs, etc.

ENTITY Let_There_Be_Light IS
  PORT (button1, button2 : IN STD_LOGIC;
        led1             : OUT STD_LOGIC );
END Let_Their_Be_Light

-- now define the architecture, that is, the actual implementation of the design
ARCHITECTURE design OF Let_There_Be_Light
BEGIN

led1 <= (button1 OR button2)
```

```
END design;
```

As you can see the VHDL is definitely a little more long winded and its harder to understand. While the Verilog version looks like a C program.

In any case, I have programmed in both and hands down I like Verilog better, its simpler, easier to learn and very C like, so most programmers can pick it up immediately. On the other hand, VHDL is a more high level system and allows for more modular design (in some cases) and libraries. However, the problem is it takes a long time to master VHDL. So I suggest you learn both and make your own decisions.

There are also all kinds of other HDLs that translate popular programming languages into Verilog or VHDL, for example, there is a Python HDL that translates into Verilog when you are ready to implement, its called **"MyHDL"**, here's a link if you're interested:

http://myhdl.jandecaluwe.com/doku.php

Verilog and VHDL are great, but they are designed for really large projects. CPLDs on the other hand are usually for simple tasks like glue logic or to combine a few logic chips, counters, etc. into one chip. Thus, there are some other simpler HDL languages that have no where near the capabilities of Verilog or VHDL, but are more than enough to get the job done for simple tasks. Two of these languages that you will see commonly used are CUPL (**C**ornell **U**niversal **P**rogramming **L**anguage) and ABEL (**A**dvanced **B**oolean **E**xpression **L**anguage). CUPL is very old and was initially a proprietary language for logic equation descriptions for PAL (**P**rogrammable **A**rray **L**ogic) which are the forerunners of CPLDs, basically PALs are AND/OR arrays that you could program the interconnects. ABEL is a more recent language that is much more capable than CUPL and what was used to develop the software for the HX512's CPLD. We are going to cover ABEL links and programming later in the manual, but if you are interested, here's a good tutorial and overview on ABEL:

ABEL Tutorial

http://www.ese.upenn.edu/rca/software/abel/abel.primer.html

And similarly, here's one for CUPL:

CUPL Tutorial

http://www.rexfisher.com/Downloads/CUPL%20Tutorial.htm

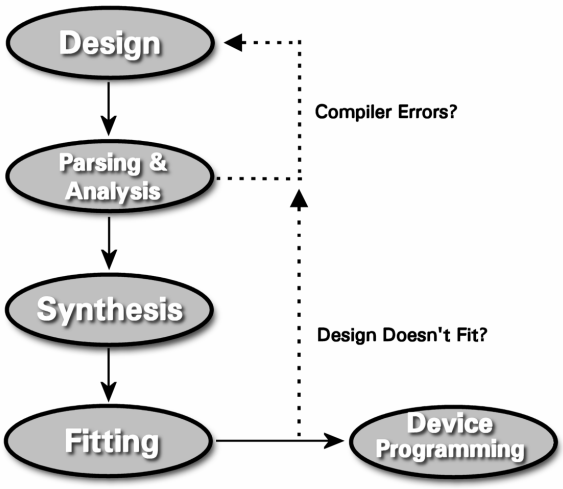
Lastly, its very important to understand that with any HDL, they are intrinsically different from normal programming since when designing hardware the description of hardware is totally parallel, that is the lines of code in a HDL program are ***concurrent***, this is one of the biggest hurdles for software engineers learning HDLs and hardware to grasp since they are used to sequential execution. But, with HDLs, everything happens at the same time, or at very least on a clock edge.

So returning back to our simple example of a generic HDL line of code:

output_1 = (input1 AND input2)

How do you get from the line of code above to the CPLD? This is where the magic comes in a lot of really complex compiler design. Take a look at Figure 13.0 below for a flow chart.

Figure 13.0 – Flowchart of HDL to Final CPLD.



Referring to Figure 13.0, the first step after design is the parsing and analysis of the program and its meaning. Once the logical meaning of the program is understood then the program is converted into various intermediate forms and the circuit synthesis phase begins. In this phase, the circuit design is synthesized from the description into an actual gate level description. But, this isn't the final step, once the synthesis is complete, now its time for what's called **"targeting"** or **"fitting"** the design into the target device you have selected based on what it offers in the form of physical hardware. That is, if the chip has only NAND gates then ***everything*** needs to be converted to NAND gates!

For example, your program might need a simple AND gate. In a CPLD or PAL there are AND gates available since the chips are fairly simple, but in a FPGA, AND gates are buried within larger logic blocks, so an entire logic block or macro cell will be used for the single AND gate, thus potentially wasting space. Either way, the design is fit into the target technology and finally a program file is generated for the device which is usually of the file format JEDEC which stands for **J**oint **E**lectron **D**evice **E**ngineering **C**ouncil. This council has developed program formats for all these devices, thus the final program is in a standard format of sorts ready for the programming stage.

This final JEDEC program is then downloaded into you PAL, GAL, CPLD, or FPGA and you are off and running! Of course, in the real world after the program has been synthesized, a designer may verify, or optimize the design iteratively with special tools to see if it actually does what its supposed to do. This phase is called ***verification*** and ***optimization*** and is very important. Once the designer is happy with the design then he/she can go ahead and target, program it, and try the real part out.

1.7.1 Pseudo Code for CPLD

The CPLD used for the HX512 is a 100-pin 64 I/O device with 64 logic blocks. It's a bit difficult to put this into terms of a gate equivalent since not all gates can be used individually, rather I prefer to think in terms of how many bits can the device store and manipulate? In this case, there are 64 logic blocks and each block has a single flip flop, thus you can work with a single 64-bit number or with four 16-bit numbers and so forth. But, in reality the combinational logic needed to perform logic and state control will eat up your logic blocks just as well and the flip flops may be orphaned. Thus, if you can get 70-90% utilization of the chip you are doing really well.

In the case of the HX512's memory controller implementation, the entire chip was used, even though there was only a 19-bit counter in it. However, the various states, conditional logic, math operations for addition and subtraction quickly

consumed the chip's silicon. So even though the chip has 64 logic blocks and 64 flip flops, the memory controller behavior barely fit into the chip. Moreover, the final fitting process is also constrained by the final location on the package pins where you want your inputs and outputs. The more you constrain the design, the worse it will fit, or it won't fit at all. However, you must constrain the design's package I/O because if you don't then your final inputs and outputs are at the will of the fitter and can move all over the place. Thus, once you have a PCB designed and you make a single change to your HDL code then the chip changes! And this is not good. Thus, a technique that I use is to constrain as much of the design's I/O to certain pins as I can and then let other I/O free to fit where it wants. Then after a few synthesis runs, I see where I/Os are ending up. Then I take those values and constrain the I/Os to where they end up in most cases. Thus, I let the fitter tell me where it wants things most of the time, then I turn off auto fit and tell it to always fit that way (so it likes what I tell it). Then once my package I/Os are stable I can design the PCB and not worry about making changes to the CPLD code in the future since I am confident the design will still synthesize and fit into the same I/O pins. Bottom line, lots to worry about.

However, right now, let's just talk about the general functionality of the CPLD and then we will look at the actual ABEL code that implements it. Assuming you have read the previous documentation and API interface then you know what the memory controller does (implemented in the CPLD), transcribing this to a pseudo-code/algorithmic description we have the following tasks:

CPLD Algorithmic Description for Memory Controller Behavior

Task 1: Assign all input and output pins to the appropriate package pins on the 100-pin package.

Task 2: Generate all combinatorial logic outputs and set up flip flop clocks, resets, etc.

Task 3: Start memory controller in configuration **PROGRAM** state and wait for programming, store 4-bit program into configuration bits.

Task 4: Once programming is complete transition memory controller into **RUN** state.

Task 5: While in RUN state test the following conditions and execute the appropriate code:

Condition 1: Latch operation being requested.

Sub-Condition 1: Latch low 8-bits of address, continue back to RUN state.
Sub-Condition 2: Latch upper 8-bits of address, clear high 3-bits, continue back to RUN state.

Condition 2: Read operation being requested.

Sub-Condition 1: Read byte from memory, request read from SRAM via bus interface, auto increment address counter based on previously programmed configuration bits.

Sub-Condition 2: Read byte from memory, request read from SRAM via bus interface, auto decrement address counter based on previously programmed configuration bits.

Condition 3: Write operation being requested.

Sub-Condition 1: Write byte to memory on data bus, request write from SRAM via bus interface, auto increment address counter based on previously programmed configuration bits.

Sub-Condition 2: Write byte to memory on data bus, request write from SRAM via bus interface, auto decrement address counter based on previously programmed configuration bits.

Task 6: Continue in RUN state.

Now, let's briefly describe these tasks one by one.

Task 1 is very specific to HDLs and the hardware. When writing HDL code, if you only want to simulate then you don't care about any physical chip or package, but in the case of the HX512, it uses a real part, the ispMach 4064 100-pin package specifically, so we must assign pins to signal names, this is the idea behind this task in the code. This doesn't really happen at real-time, but still part of the program.

Task 2 is a concurrent task or process for the memory controller. The idea here is that we need to generate signals that are simply combinatorial combinations of various inputs. For example, the SRAM has a **"control interface"** consisting of read/write line, chip select, etc. and many of these lines are simple ANDs and Ors of the control lines coming from the HYDRA, thus we can assign them in a couple lines of code each. Lastly, the design also has clocked logic or sequential logic specifically the internal state machine and the counter that is used as the address latch, thus, we need to tell the synthesizer exactly which signals are going to clock these components.

Task 3 is the state machine starting point for the memory controller, it waits for a single clock strobe on the clock line, when it receives this clock strobe whatever the data is on the data bus, the first 4-bits are used as the configuration program for the memory controller.

Task 4 is really a finishing step for Task 3, when Task 3 is complete, the controller needs to transition to the **RUN** state.

Task 5 is more or less where all the action takes place. In this task, the control inputs are resolved and used to determine what logical path and what operation (latch, read, write) is being requested. Furthermore, within each operation the configuration bits play a role since they control the auto increment/decrement operation after each read and write.

Finally, **Task 6** just makes sure the system loops, but again the concept of loop infers sequential operation and HDLs are more concurrent programming languages which reflects how the hardware works. Thus, Task 6 just forces the **RUN** state always.

1.7.2 ABEL Driver Code

The ABEL driver code for the HX512's memory controller behavior is listed below. Although, some of the syntax will look alien, much of it is easy to follow at least in the abstract. The devil is in the details though when it comes to HDLs and you need to know exactly what every line does since side affects are disastrous if they aren't intended. The latest driver source code is located on the CD in the **\SOURCES** directory in the following location:

CD_ROOT:\SOURCES\hydra_sram_controller_03.abl

Listing 1.0 below is the source for reference; however, some of the extraneous commented out code etc. has been omitted to save space.

Listing 1.0 – Memory Controller ABEL Code.

```
MODULE hydra_sram_controller_03

TITLE 'hydra_sram_controller_03'

// Version 3.0 - Last modified 2.3.07
// Comments:
// Support up to 512K SRAMs, directly addressable up to 64K, 128K, 256K, 512K must be accessed via autoincrementing.
// Needs 4064 or better to fit.


// this is the controller register that generates the control signals for the low and high address latches
// Command codes are 2-bit c1 and c0
//
// c1 c0 | Function
//  0  0 | write byte to memory
//  0  1 | read byte from memory
//  ------------------
//  1  0 | load low address into address latches
//  1  1 | load high address into address latches


// Command format program(sent low to high)

// 4-bit format

// |  3   2   |  1   0
// |  sr  r0  |  sw  w0
```

```
// pgm3.....pgm0

// sw   - sign bit for write post increment/decrement (1=add, 0=subtract).
// w0   - 1 bit magnitude for write post increment/decrement (w1 ignored in this version).

// sr   - sign bit for read post increment/decrement  (1=add, 0=subtract).
// r0   - 1 bit magnitude for read post increment/decrement (r1 ignored in this version).

DECLARATIONS

    // inputs
    !sram_clk   pin 38 ;  // general system clock
    !sram_resn  pin 8 ;   // hydra reset line (active low)
    sram_c0     pin 9 ;   // hydra control line 0
    sram_c1     pin 10 ;  // hydra control line 1

    d7..d0      pin 15..17,19..23  ; // data bus
    sram_data   = [d7..d0];

    //  outputs
    sram_c0n    pin 3 istype 'com';    // control line 0 out inverted
    sram_c1n    pin 4 istype 'com';    // control line 1 out inverted
    !sram_csn    pin 5 istype 'com';   // sram cs output

    sram_read_led pin  78 istype 'com';
    sram_write_led pin 79 istype 'com';
    sram_latch_led pin 80 istype 'com';

    // registers
    a18..a0      pin 28..31, 41..44, 53..56, 64..67, 70, 71, 72   istype 'reg'; // 19-bit address register 0..512K address space
    sram_addr   = [a18..a0];          // full sram address

    // memory controller "firmware", consists of 6-bit program that describes post read/write behavior
    pgm3..pgm0  pin 14,35, 36, 37  istype 'reg';               // holds sram controller program bits

    sram_pgm    = [pgm3..pgm0];

    // state register used to control flow from programing mode and final "run" mode
    sram_state          state_register;
    s0, s1              state;

EQUATIONS

    // sram chip select needed for srams with only /CE, rather than CE1, /CE2, that is duel high/low enables
    sram_csn  = (!sram_c1 & !sram_clk);

    // command bit(s) inverted
    sram_c0n  = !sram_c0;
    sram_c1n  = !sram_c1;

    // assign clock register and resets
    sram_addr.clk = sram_clk;
    sram_addr.ar  = sram_resn;

    sram_state.clk = sram_clk;

    sram_pgm.clk   = sram_clk;
    sram_pgm.ar    = sram_resn;

    // begin state machine, after reset, s0, begin programming mode, bits d5..d0 are used as program and latched
    state_diagram sram_state

      state s0: // load program bits into controller, looks for C1, C0 = {0,0}, then reads data on next clock into controller

         if (!sram_c1 & !sram_c0) then s1 with  {sram_pgm  :=  [d3..d0];}
         else s0

      state s1:    // run mode, latch low, latch high, read, write, etc.

         sram_pgm := sram_pgm;

         // latch operations
         when      (sram_c1 & !sram_c0) then  { sram_addr := [a18..a8, d7..d0]; sram_latch_led = 1; }
         else when (sram_c1 & sram_c0) then   { sram_addr := [0,0,0, d7..d0, a7..a0];  sram_latch_led = 1;}

         // read operation with post add/sub
         else when (!sram_c1 & sram_c0 & pgm3) then
                   { sram_addr := sram_addr +  [0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0, pgm2];  sram_read_led = 1;}

         else when (!sram_c1 & sram_c0 & !pgm3) then
                   { sram_addr := sram_addr - [0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0, pgm2];  sram_read_led = 1;}

         // write operation with post add/sub
         else when (!sram_c1 & !sram_c0 & pgm1) then
                   { sram_addr := sram_addr +  [0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0, 0, pgm0]; sram_write_led = 1;}

         else when (!sram_c1 & !sram_c0 & !pgm1) then
                   { sram_addr := sram_addr - [0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0, 0, pgm0]; sram_write_led = 1;}

         else      { sram_addr := sram_addr; }

      goto s1;

    async_reset s0 : (sram_resn);

end
```

Teaching ABEL is beyond the scope of this user manual, if you're interested please make sure to read the Lattice ABEL manuals and the various web links provided. A good introduction to ABEL programming (the Lattice ABEL Design Manual) is located on the CD here:

CD_ROOT:\DOCS\ABEL_DESIGN.PDF

Nevertheless, let's take a brief look at the code and see if we can map the tasks we outlined in the pseudo-code with the actual program. The program begins with the ***MODULE*** name and optional ***TITLE*** string. The title string is for commenting purposes, while the module name is the actual name of the module and typically used as the same name of the file when saved. The next section is the ***DECLARATIONS*** section and is where all input and output pins are defined and mapped to the final package if desired. If a pin is declared without a pin mapping then this can be done later in the tool chain with the constraint editor and the package tool. Thus, when writing generic ABEL code for reuse by other engineers, its best to ***not*** assign pin numbers, this way the code is as reusable as possible. In the declarations area you will see the general syntax to define a pin or group of pins is:

For outputs:

name ***PIN*** nn ***'ISTYPE'*** { ***'COM'*** | ***'REG'*** } ;

where **nn** is a number that indicated the package pin. N

There are other variants of this definition, but **COM** and **REG** are the most common. They stand for **COM**binatorial and **REG**istered. They simply tell the compiler that a particular pin is an output and is a simple combinatorial combination of other signals, or that the pin needs to retain state and thus needs to be sent thru a flip-flip, in other words **"registered".**

For inputs the syntax is a little easier:

name ***PIN*** nn;

Only the name and the pin number are needed. Furthermore, the ***DECLARATIONS*** section of the code is where all the pin definitions for inputs and outputs are made. Also, you will see groupings of pins to create easier to work with aliases, for example the data bus is defined as:

```
sram_data   = [d7..d0];
```

Thus, instead of individually naming d7, d6,…d1,d0 all the time in code, ***sram_data*** can be used instead. The **ABEL Design Manual** has more detail on these kinds of short cuts.

After all the inputs and outputs are defined, then its time for the ***EQUATIONS*** section of the code, this is where all the action takes place. Moreover, this part of the code is parallel or concurrent, everything more or less happens at the same time, so you have to think in parallel. The first chunk of code before the state machine definitions maps to Tasks 1 and 2 more or less. In this code fragment, the simple continuous combinatorial logic statements are made that related outputs to inputs as well as the clocks, resets, and other control signals are assigned for all the clocked logic aspects of the system. That is, you have to tell ABEL what you want clocking your flip flops for your registered outputs, as well as how you want them reset.

Next up, is the state machine that makes the CPLD look like a memory controller from the outside world. The first state is the reset state and here the logic loops waiting for a "00_2" binary code on the control inputs, once it sees it then if you clock the system then the data on the data bus is loaded into 4 flip flops which store the configuration bits, then the state machine transitions into the **RUN** state. Thus, the first state maps to Task 3. The remaining tasks are handled at once by a conditional logic tree that branches for latch, read, write and then sub-branches for addition or subtraction operations to implement the post increment / decrement for reads' and writes'. And that concludes the general overview of the ABEL driver code.

1.7.3 Installing and Running the Lattice Tool Chain

The Lattice tool chain is like any other CPLD/FPGA tool chain – ***complicated***. However, I can tell you that after working with Xilinx and Altera tools (the two top FPGA companies in the world), the Lattice tools are a little easier to work with for beginners. The following sections will get you started with the tools, but you really do have to read all the Lattice documentation, buy a programmer (or build one), and spend a good deal of time to learn to work with CPLDs.

In any event, to start with you are going to need to download and install the Lattice tool chain. Luckily you can get a free 6-month trial of **"ispLEVER"** which is Lattice's main tool. Here's the web page to download the application and all of its components:

http://www.latticesemi.com/products/designsoftware/isplever/ispleverstarter/index.cfm

Open the page up and you will see three steps outlined in a table;

Step 1 – Is the downloads area. You can download anything you want, but all you need is the **"Primary Module .exe, (220MB)"**, and the **"Help and User Guides Module .exe (50MB)"**. You can find these files on the CD located in the **TOOLS****LATTICE** directory, but they may not be the latest, so you might want to download the latest files if you have broadband (however, the files on the CD will work just fine).

Step 2 – Install the **Primary Module** and the **Help module**. They are .exe's, so simply launch each and follow instructions.

Step 3 – Licensing the software. You need a license to run ispLEVER starter kit, you can do that at this page:

http://www.latticesemi.com/licensing/flexlmlicense.cfm

Lattice simply needs your email address and they will send you a key.

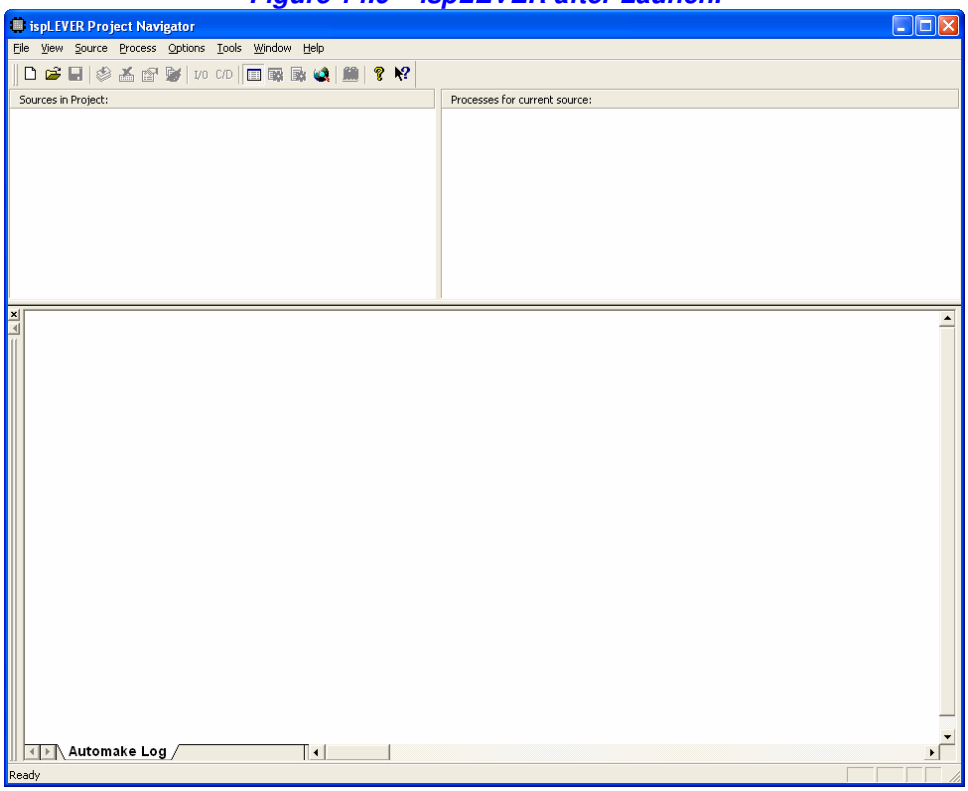
TIP	You are free to download the other modules for FPGAs including the FPGA Module, Precision RTL Synthesis Module, and the Synplify Synthesis Module. All of these are for more advanced functionality such as FPGA and Verilog/VHDL support and synthesis. But, since we only need ABEL to work with CPLDs the Primary Module will do the trick.

Now that you have the software installed and licensed, the next step is to re-program the CPLD with the original firmware if you want to experiment. Of course, I suggest that you read the online docs for ispLEVER cover to cover as well as the ABEL Design Manual. And of course you will need a download cable/programmer. You can either buy the USB or parallel cable or make one of your own (instructions shown in **Appendix C**).

1.7.4 Re-Programming the CPLD with the Standard Firmware for the HX512

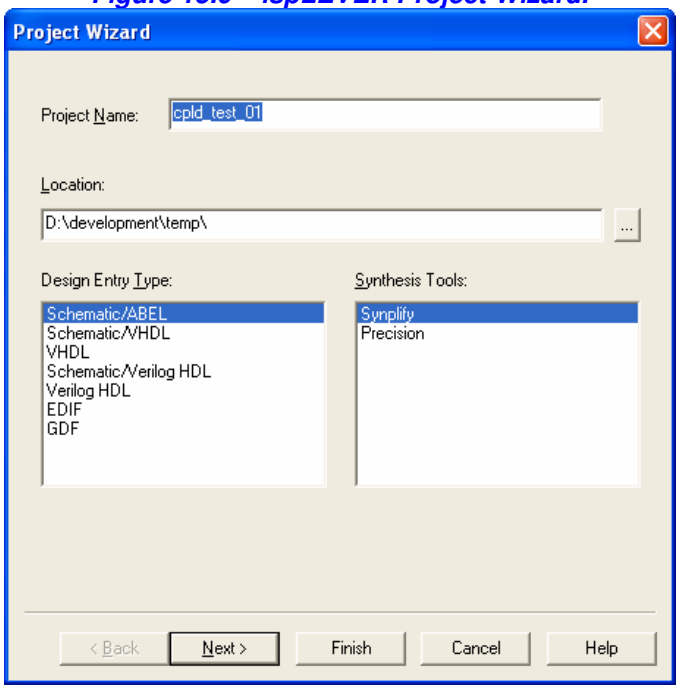
After you have installed and licensed ispLEVER, go ahead and launch the application, so we can run thru a few general steps of how to get things going. These are only hints though, your experience may vary due to setup or other changes.

Figure 14.0 – ispLEVER after Launch.



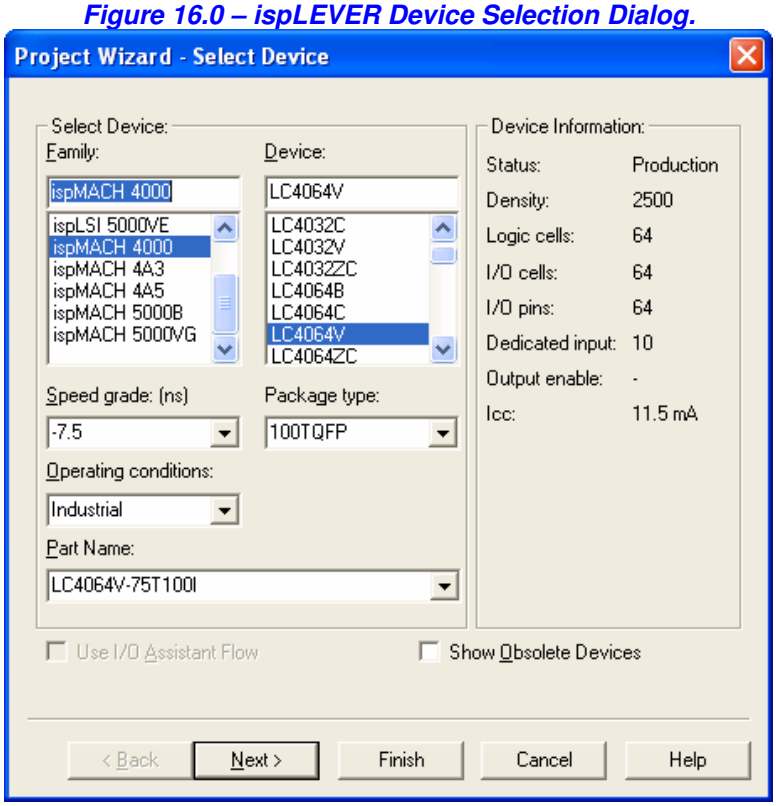
You should be presented with the very friendly ispLEVER project management system which should look something like that shown in Figure 14.0. Next, let's create a test project; select **<FILE→NEW PROJECT>** from the main menu bar, you should the new project dialog as shown in Figure 15.0.

Figure 15.0 – ispLEVER Project Wizard.



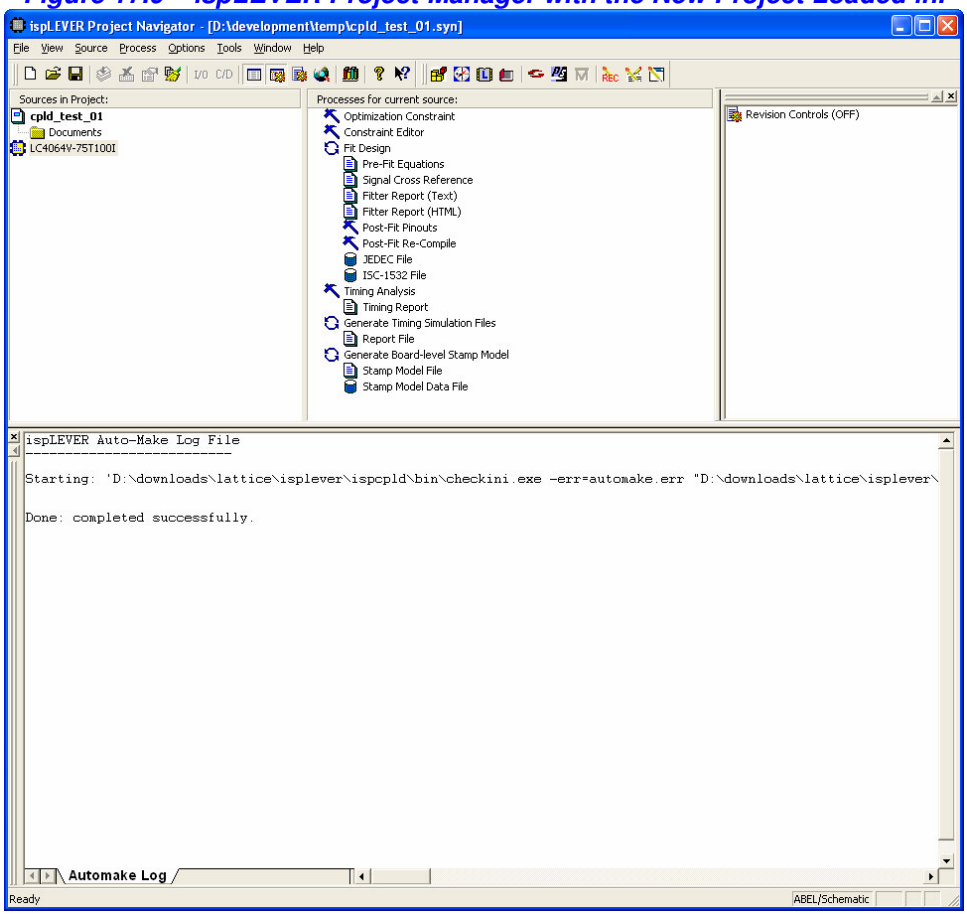
First, you need to give the project a name, I suggest **"CPLD_TEST_01"**, then browse and store the project somewhere where you can get to it, but you can delete it fairly easily if you make mistakes. In this case, I have selected my development drive **D:\DEVELOPMENT\TEMP** for fun. We are interested in creating a SCHEMATIC/ABEL design, so select that on the left side pane. On the right side pane, you shouldn't have the Synplify, Precision entries unless you installed the other modules. If you do have them, select either, since its irrelevant for ABEL design. Only if you pick Verilog, VHDL, etc. do you need these more advanced synthesizers. Go ahead and click **<NEXT>** when you are done with this dialog.

Figure 16.0 – ispLEVER Device Selection Dialog.



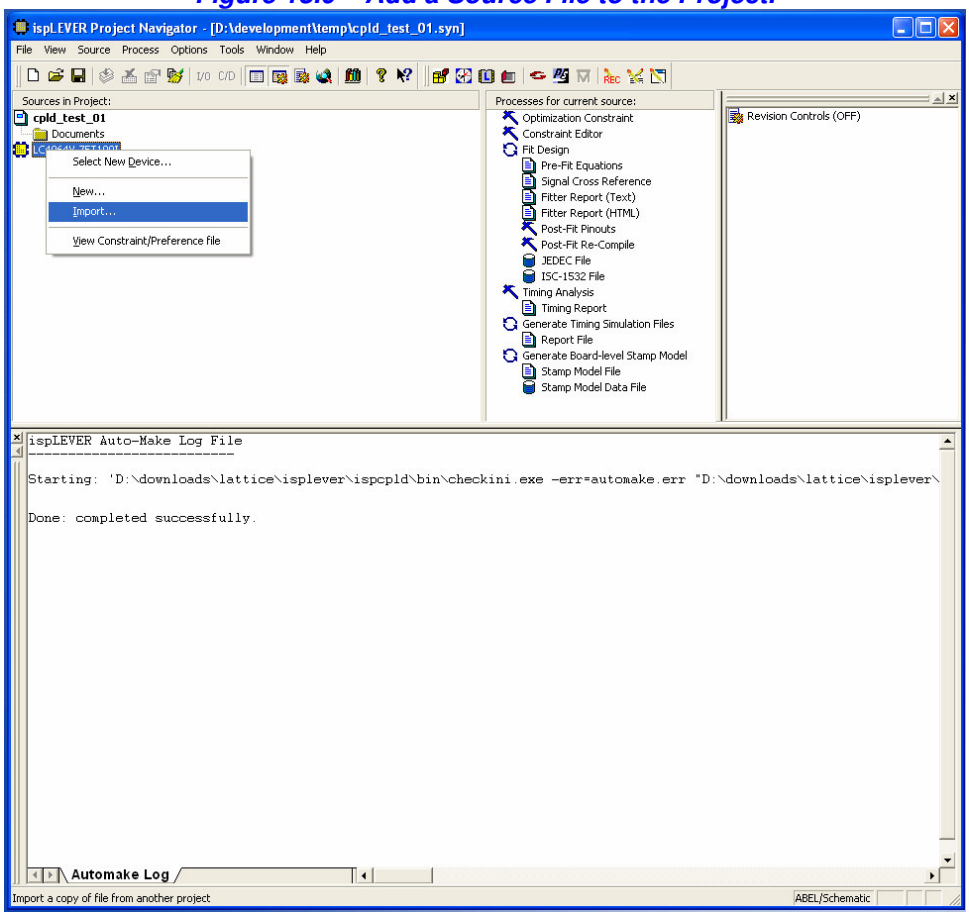
Next up is the device selection dialog which allows you to select the chip you want to target. Figure 16.0 shows the dialog after its been filled out completely. Make sure to select **"ispMACH"** for Family, **"LC4064V"** for Device, **"7.5ns"** for Speed Grade, **"100 TQFP"** for Package Type, **"Industrial"** for Operating Conditions, and finally the dialog should select the part for you based on all that data -- Part Name **"LC4064V-75T100I"**. If it doesn't select properly then activate the drop down on **Part Name** and make sure you select the **"LC4064V-75T100I"**. When you have filled the form out, take a look at the right pane to see some of the characteristics of the chip, when you are done press **<FINISH>** since the remaining dialogs aren't needed.

Figure 17.0 – ispLEVER Project Manager with the New Project Loaded in.



After clicking **<FINISH>** the system will pause for a moment and build the project, then you should see something like that shown in Figure 17.0. There should be three panes; to the left, the sources, the center pane shows the operations that have been completed as well as numerous tools, and finally on the right pane is the ***revision*** panel which we aren't too interested in.

Figure 18.0 – Add a Source File to the Project.



Let's load in the memory controller firmware that was pre-loaded in the CPLD already, and re-program the CPLD with the same software. Thus, we need to load the source file into the project in the appropriate place. Referring to Figure 18.0, the left sources pane, select the little yellow chip icon that is labeled **"LC4064-75T100I"**, right click, and select the **"Import"** item. This will launch a file browse dialog. Browse to your CD (or where you copied the CD) and find the file:

CD_ROOT:\SOURCES\hydra_sram_controller_03.abl

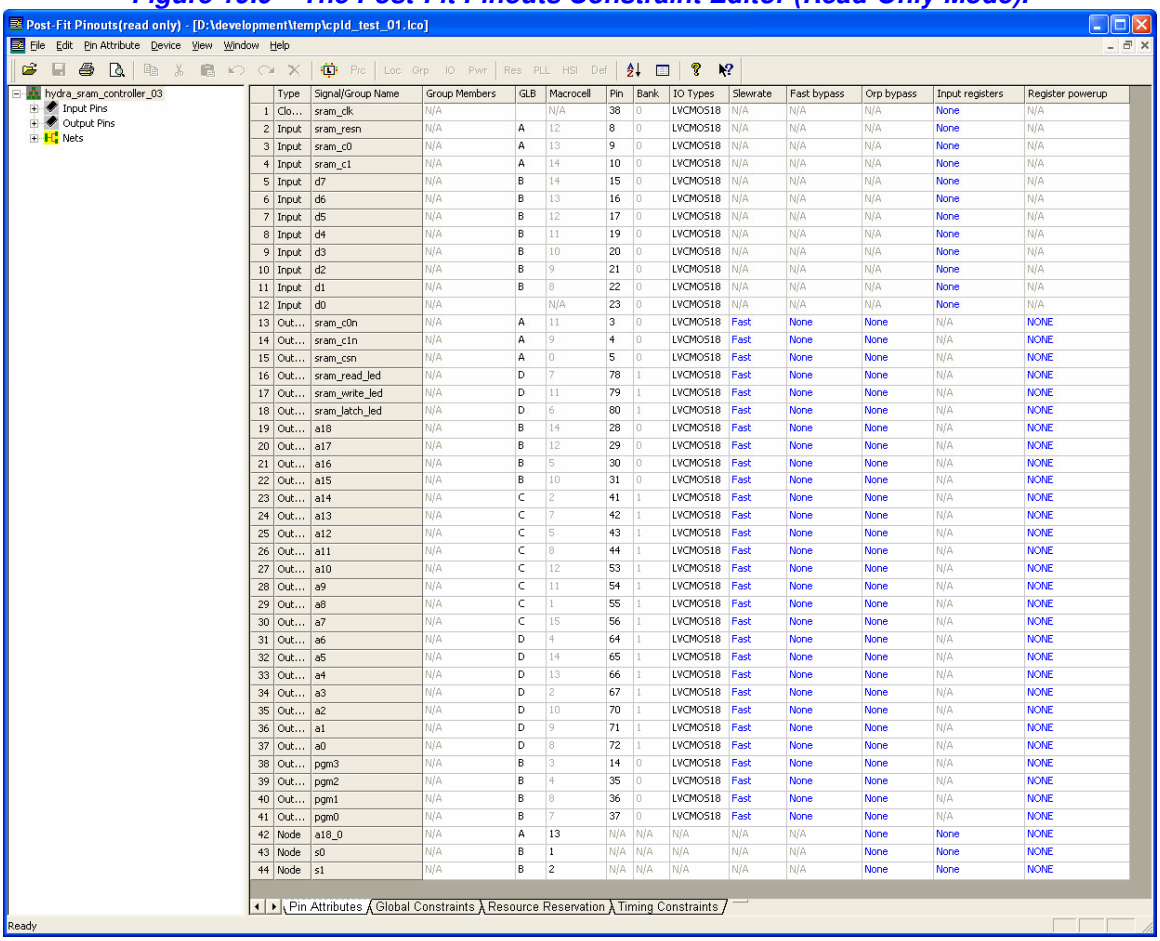
and select it. This will load the file into the project. After you load the file, ispLEVER may take from a few seconds to a moment to load the file and process it. This is unavoidable when dealing with silicon compilers, they are very complex and a few hundred lines of HDL can take moments to hours to compile and fit into a chip! Luckily, the memory controller program isn't too complex, so the worst delays you should see on a P3/P4 are in the seconds to maybe 1-2 minutes for various operations.

Anyway, the bottom output window should show the ABEL-HDL processor executed successfully without any problems. Next, we are going to build the project and get ready to load it into the chip. In the middle pane of the tool **"Processes for Current Source"**, go ahead and double click **"Fit Design"**, this will take a moment to process, so keep an eye on the bottom information pane as it works…there is a percent complete at the very bottom. Usually takes 2-5 minutes to build.

If everything goes well, you will see a **"Import Source Constraints"** dialog open. Its asking how you want to handle **"Constraints"** for now, tell it to **"IMPORT"** them. Constraints simply control the pins, the I/O types, and other physical, and timing constraints you might want/need to meet. For now we will use the constraints in the source file along with the system defaults.

In a moment, the process should complete and you will see an information message box that tells you the process is complete, but warnings were generated -- this is normal, click **<OK>**.

Figure 19.0 – The Post-Fit Pinouts Constraint Editor (Read-Only Mode).



	Type	Signal/Group Name	Group Members	GLB	Macrocell	Pin	Bank	IO Types	Slewrate	Fast bypass	Orp bypass	Input registers	Register powerup
1	Clo...	sram_clk	N/A		N/A	38	0	LVCMOS18	N/A	N/A	N/A	None	N/A
2	Input	sram_resn	N/A	A	12	8	0	LVCMOS18	N/A	N/A	N/A	None	N/A
3	Input	sram_c0	N/A	A	13	9	0	LVCMOS18	N/A	N/A	N/A	None	N/A
4	Input	sram_c1	N/A	A	14	10	0	LVCMOS18	N/A	N/A	N/A	None	N/A
5	Input	d7	N/A	B	14	15	0	LVCMOS18	N/A	N/A	N/A	None	N/A
6	Input	d6	N/A	B	13	16	0	LVCMOS18	N/A	N/A	N/A	None	N/A
7	Input	d5	N/A	B	12	17	0	LVCMOS18	N/A	N/A	N/A	None	N/A
8	Input	d4	N/A	B	11	19	0	LVCMOS18	N/A	N/A	N/A	None	N/A
9	Input	d3	N/A	B	10	20	0	LVCMOS18	N/A	N/A	N/A	None	N/A
10	Input	d2	N/A	B	9	21	0	LVCMOS18	N/A	N/A	N/A	None	N/A
11	Input	d1	N/A	B	8	22	0	LVCMOS18	N/A	N/A	N/A	None	N/A
12	Input	d0	N/A		N/A	23	0	LVCMOS18	N/A	N/A	N/A	None	N/A
13	Out...	sram_c0n	N/A	A	11	3	0	LVCMOS18	Fast	None	None	N/A	NONE
14	Out...	sram_c1n	N/A	A	9	4	0	LVCMOS18	Fast	None	None	N/A	NONE
15	Out...	sram_csn	N/A	A	0	5	0	LVCMOS18	Fast	None	None	N/A	NONE
16	Out...	sram_read_led	N/A	D	7	78	1	LVCMOS18	Fast	None	None	N/A	NONE
17	Out...	sram_write_led	N/A	D	11	79	1	LVCMOS18	Fast	None	None	N/A	NONE
18	Out...	sram_latch_led	N/A	D	6	80	1	LVCMOS18	Fast	None	None	N/A	NONE
19	Out...	a18	N/A	B	14	28	0	LVCMOS18	Fast	None	None	N/A	NONE
20	Out...	a17	N/A	B	12	29	0	LVCMOS18	Fast	None	None	N/A	NONE
21	Out...	a16	N/A	B	5	30	0	LVCMOS18	Fast	None	None	N/A	NONE
22	Out...	a15	N/A	B	10	31	0	LVCMOS18	Fast	None	None	N/A	NONE
23	Out...	a14	N/A	C	2	41	1	LVCMOS18	Fast	None	None	N/A	NONE
24	Out...	a13	N/A	C	7	42	1	LVCMOS18	Fast	None	None	N/A	NONE
25	Out...	a12	N/A	C	5	43	1	LVCMOS18	Fast	None	None	N/A	NONE
26	Out...	a11	N/A	C	8	44	1	LVCMOS18	Fast	None	None	N/A	NONE
27	Out...	a10	N/A	C	12	53	1	LVCMOS18	Fast	None	None	N/A	NONE
28	Out...	a9	N/A	C	11	54	1	LVCMOS18	Fast	None	None	N/A	NONE
29	Out...	a8	N/A	C	1	55	1	LVCMOS18	Fast	None	None	N/A	NONE
30	Out...	a7	N/A	C	15	56	1	LVCMOS18	Fast	None	None	N/A	NONE
31	Out...	a6	N/A	D	4	64	1	LVCMOS18	Fast	None	None	N/A	NONE
32	Out...	a5	N/A	D	14	65	1	LVCMOS18	Fast	None	None	N/A	NONE
33	Out...	a4	N/A	D	13	66	1	LVCMOS18	Fast	None	None	N/A	NONE
34	Out...	a3	N/A	D	2	67	1	LVCMOS18	Fast	None	None	N/A	NONE
35	Out...	a2	N/A	D	10	70	1	LVCMOS18	Fast	None	None	N/A	NONE
36	Out...	a1	N/A	D	9	71	1	LVCMOS18	Fast	None	None	N/A	NONE
37	Out...	a0	N/A	D	8	72	1	LVCMOS18	Fast	None	None	N/A	NONE
38	Out...	pgm3	N/A	B	3	14	0	LVCMOS18	Fast	None	None	N/A	NONE
39	Out...	pgm2	N/A	B	4	35	0	LVCMOS18	Fast	None	None	N/A	NONE
40	Out...	pgm1	N/A	B	8	36	0	LVCMOS18	Fast	None	None	N/A	NONE
41	Out...	pgm0	N/A	B	7	37	0	LVCMOS18	Fast	None	None	N/A	NONE
42	Node	a18_0	N/A	A	13	N/A	N/A	N/A	N/A	N/A	None	None	NONE
43	Node	s0	N/A	B	1	N/A	N/A	N/A	N/A	N/A	None	None	NONE
44	Node	s1	N/A	B	2	N/A	N/A	N/A	N/A	N/A	None	None	NONE

Now, under **"Fit Design"** double click, **"Post-Fit Pinouts"** and you will see a window open that looks like Figure 19.0. This is one of the constraint editors that allow you to change pin numbers if you wish, change I/O types and all kinds of other things; however, this version of it is read-only, so we can't mess anything up as we explore for now.

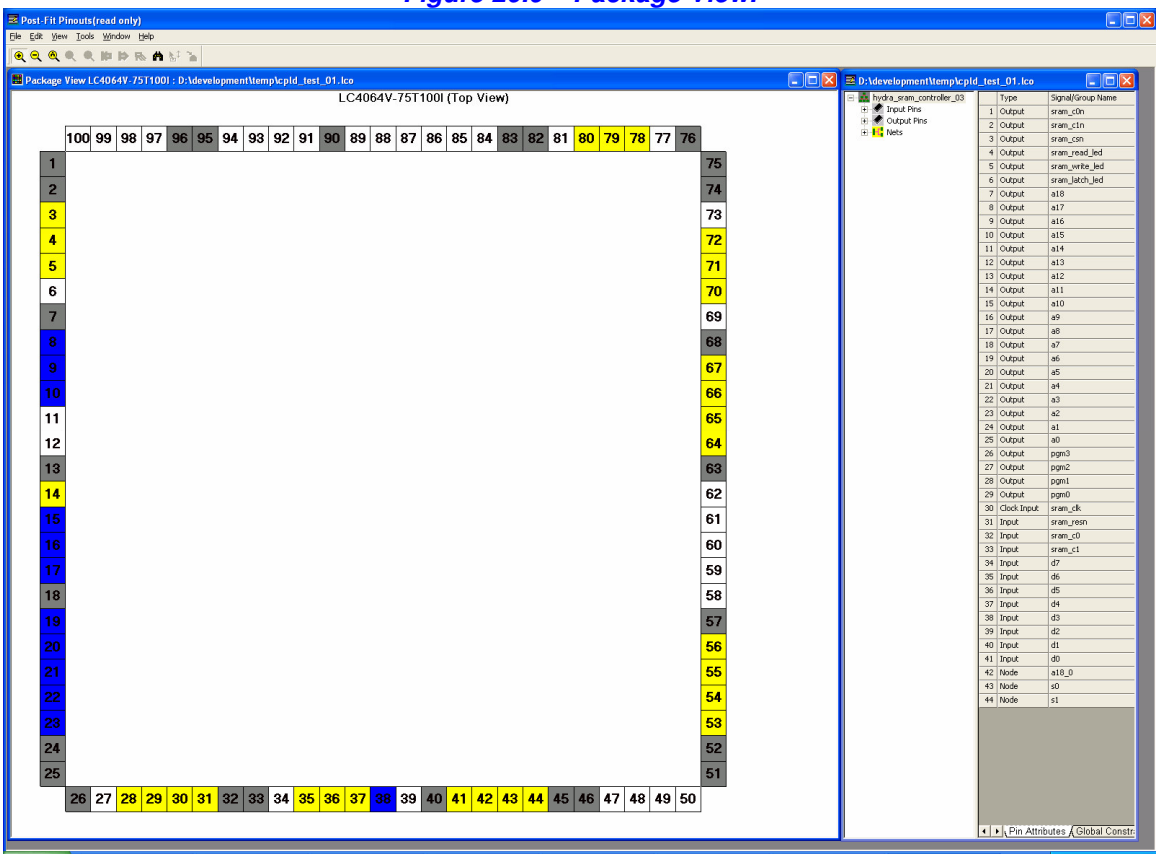
Go ahead and take a minute to see what's there. For example, try expanding the **"Inputs"**, **"Outputs"**, and "**Nets"** trees on the left pane. You will see the mapping of every single input, output, and internal working "nets" that are in the design. The right side of the window is much more complex and you want to be careful what you do here, since there isn't much of an undo with these kinds of CAD tools, so watch out! The columns represent different things relating to the architecture of the chip, the only thing we are interested is the **"IO Types"** and the **"Slew Rate"** columns. The IO Types column indicates the type of input/output that the pin is. As you can see everything defaulted to **"LVCMOS18"** which means 1.8V CMOS technology. We need to change all these to 3.3V technology since the Propeller runs at 3.3V, as well as the SRAM and the HYDRA expansion port. So, we need the I/O pads of CPLD to run at 3.3V to interface properly.

TIP	In reference to the "undo" comment in the paragraph above, you should make it a habit of hitting **<CTRL+S>** constantly when working on complex projects that you don't' want to lose. Also, I personally keep a backup version of the project I am working on with the letter "B" appended to the file name, every 20-30 mins, I resave both the primary and the backup. This way if I crash the computer while the project is open, I have the backup to fall back to and lose only 20-30 mins of work.

Next, you see the **"Slew Rate"** column. This has two settings; ***fast*** and ***slow***. Fast slew rate makes the signals change faster, but can induce noise on the lines. Thus, it's a good idea to slow the slew rate down to minimize noise unless you absolutely need the faster slew rate. We will see how to do this in a moment. Lastly, you will see tabs on the bottom of the

right window, right now we are looking at the **"Pin Attributes"** tab by default, go ahead and click the other tabs to see what they are. Again, they contain more constraint and performance information.

Figure 20.0 – Package View.

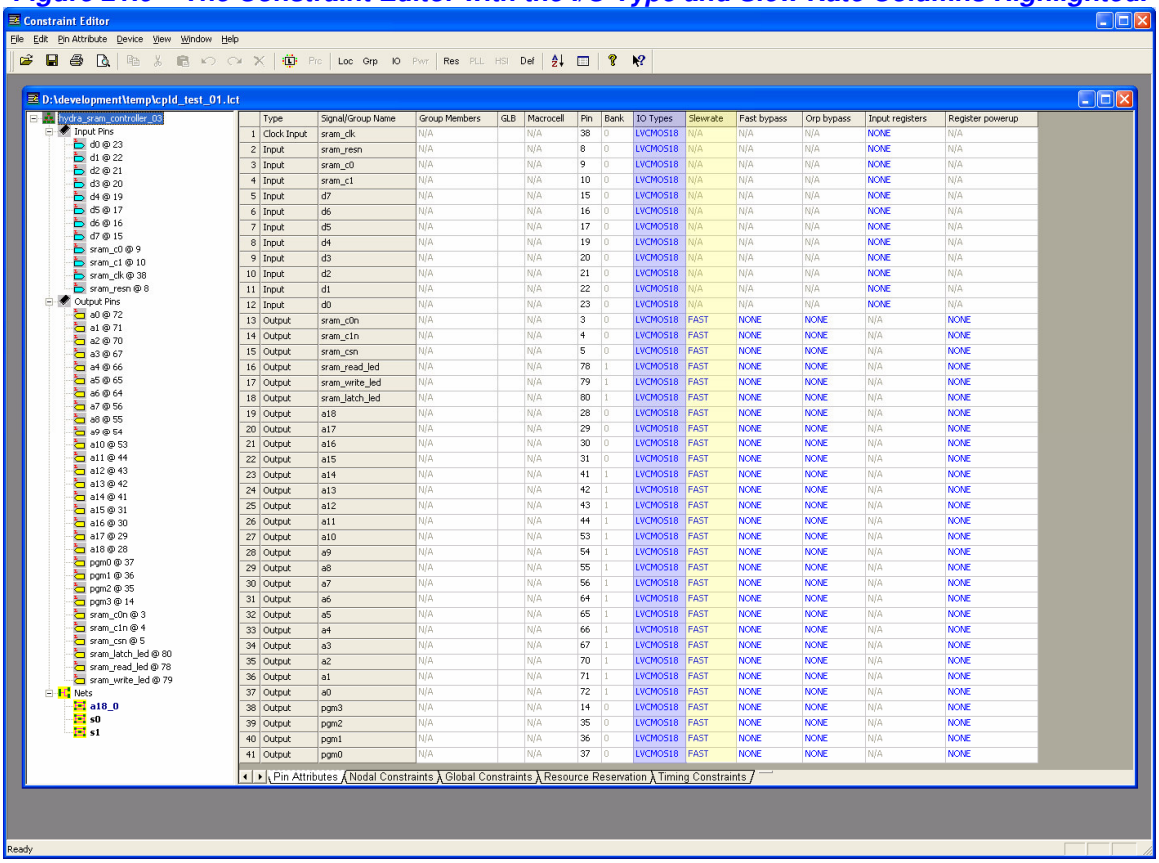


Finally, there are a lot of cool options in this tool view, but one of my favorite is the **"Package View"** which actually shows a top view of the chip package with the pins that are used (with highlighting), its great for planning layout and chip resource allocation. To get this view, select **<Device → Package View>** from the main menu and you will see the 100-pin QFP fill into the left pane. It will be too big, so zoom it out with the zoom tool (yellow buttons above the view). When you're done, you should see something like that shown in Figure 20.0.

1.7.4.1 Changing the I/O Types and Slew Rate

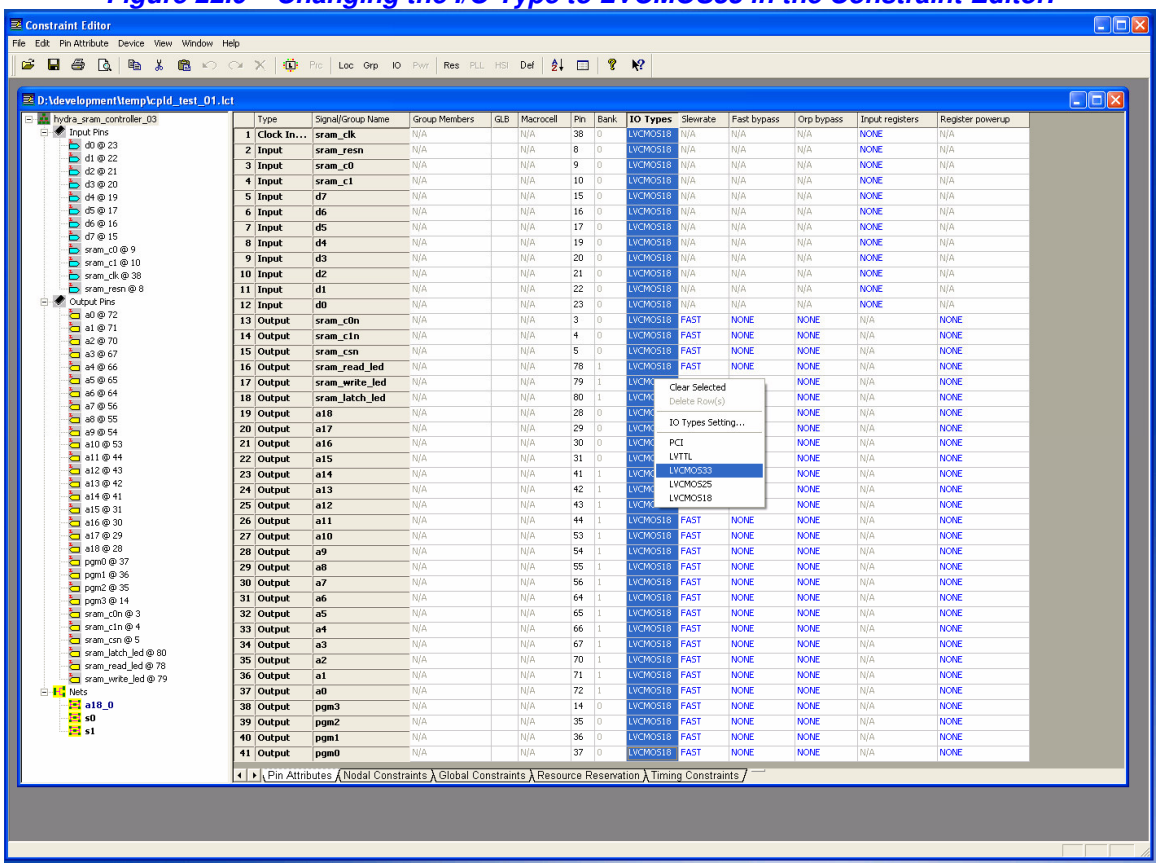
In the previous section we experimented with the Post-Fit Pinouts in ***read-only*** mode, now we are going to go ahead and launch the full blown constraint editor that allows changes to be made. To do this, make sure you have closed the Post-Fit Pinouts window and then locate the **"Constraint Editor"** link in the middle pane ("Processes for current source"), its right above the **"Fit Design"** link. Go ahead and click the **Constraint Editor** link and you should see something identical to what you saw before, take a moment to resize the windows and get everything into view. Also, make sure the **"Pin Attributes"** tab is selected at the bottom of the window.

Figure 21.0 – The Constraint Editor with the I/O Type and Slew Rate Columns Highlighted.



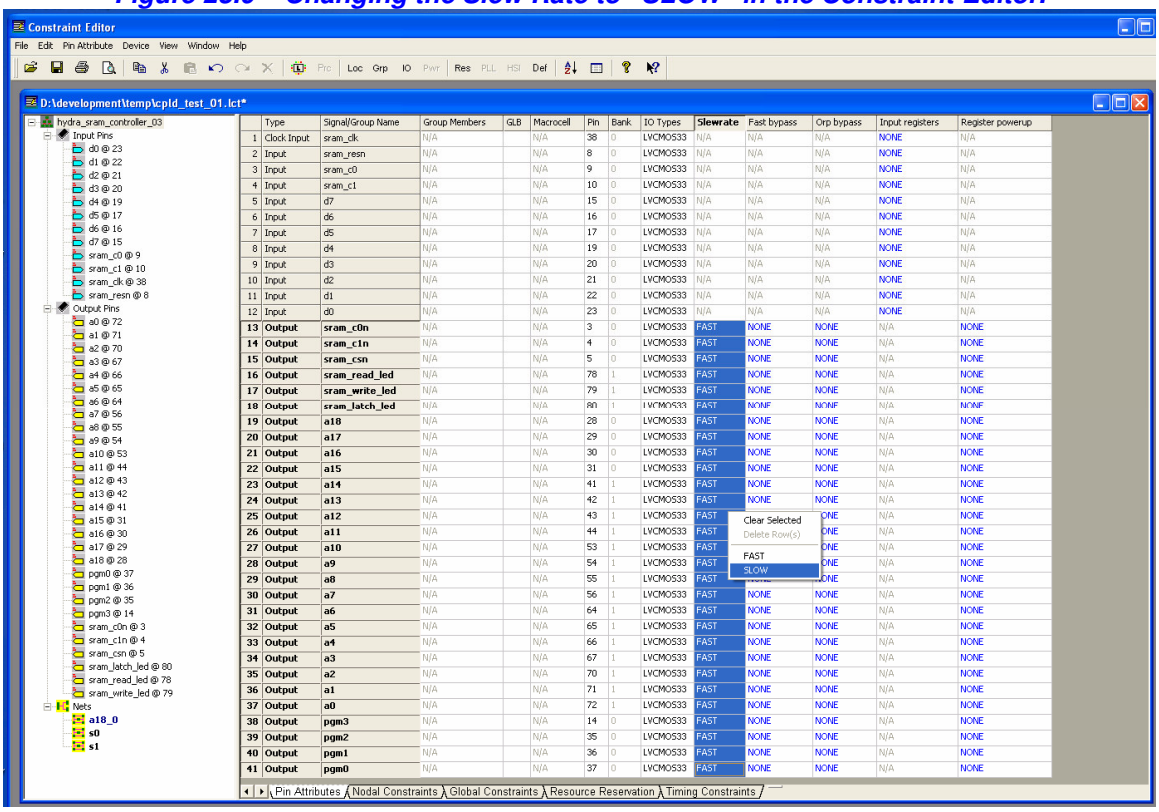
When you have everything set up and sized right, you should be able to see all 48 pins listed in each row as shown in Figure 21.0. Also, the two columns of interest are highlighted for reference. So, what we want to do is change both I/O Type to **"LVCMOS33"** and the Slew rate to **"SLOW"**. Let's do this one step at a time.

Figure 22.0 – Changing the I/O Type to LVCMOS33 in the Constraint Editor.



Referring to Figure 22.0, select the entire **"I/O Types"** column (select the top row then hold **<SHIFT>** and select the last row) and then right click the mouse, and select **"LVCMOS33"**. Now, to be safe let's slow the slew rate down.

Figure 23.0 – Changing the Slew Rate to "SLOW" in the Constraint Editor.



Now, we need to change the slew rate; however, if you refer to Figure 23.0, you will notice in the Slew rate column ***inputs*** do not have the slew rate option, this is because for inputs, it makes no sense to control slew rate, only for outputs does it make sense. Considering that, select all the outputs in the Slew rate column as shown in Figure 23.0, and then change them to **"Slow"** with a right click.

Save the constraint file with **<CTRL+S>** and then close the Constraint Editor window.

1.7.4.2 Generating the JEDEC File

The next step is to finally generate the JEDEC programming file which is used by yet another tool chain to actually program the CPLD, but we will get to that in a moment, for now, let's simply generate the JEDEC file. To do this, assuming you are looking back at the main **ispLEVER Project Navigator** MDI window, select **"JEDEC File"** in the middle pane under **"Processes for current source"**, You will get yet another warning dialog when its complete, simply click **<OK>**.

If you look in the directory that you built the project in, somewhere you should find the JEDEC file:

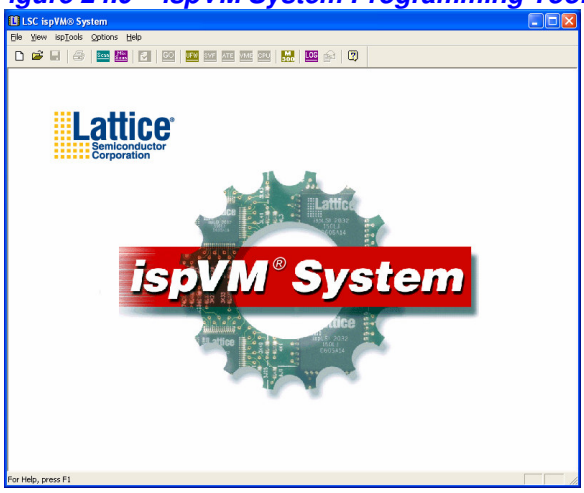
cpld_test_01.jed

Go ahead and open it with a text editor and look at the file, you will see a human readable file with pin definitions followed by a large binary bitmap (or fuse file) consisting of 1's and 0's. These are the actual program bits for the CPLD. At this point, you are finally ready to program your CPLD.

1.7.4.3 ispLEVER Setup for Programming the CPLD

To program any CPLD or FPGA there is a completely different tool needed that is totally external to ispLEVER, its integrated into ispLEVER for you convenience, but can be launched externally. Let's go ahead and launch the program, to do this navigate to the top menu bar on ispLEVER and select **<TOOLS → ispVM System>**, this launches the programming tool which is shown in Figure 24.0.

Figure 24.0 – ispVM System Programming Tool.



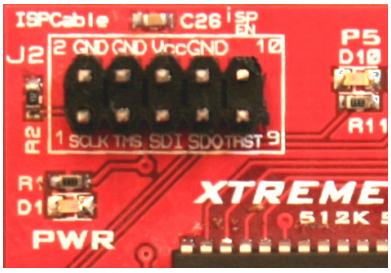
If all goes well, you should see the simple window shown in Figure 24.0. This tool is the main programming tool for all Lattice CPLDs/FPGAs, so it does far more than we need, thus the trick is to avoid as much as possible of it and just download our code into the HX512's CPLD. Therefore, there are a few steps to do this:

Step 1: Make sure that the programmer is physically connected to the PC and its driver is installed for either the USB/Parallel Port programming cable.

Step 2: Create what's called a **"Chain Configuration"** for our device and JEDEC file which can be used over and over, so that when you update the JEDEC file, you don't have to re-create the **"Chain Configuration"** file. Chain configuration is just a fancy name for programming file setup.

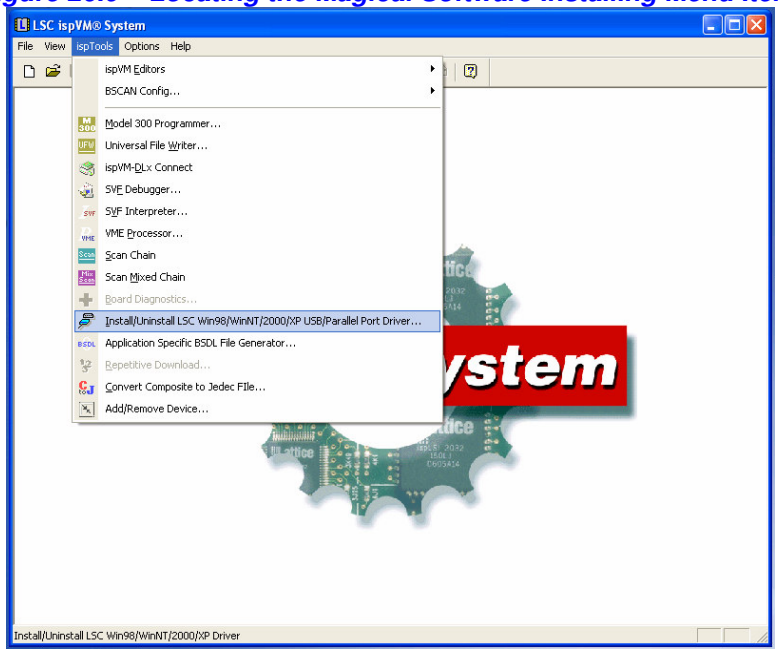
Step 3: Hook the programmer up to the HX512 (must be plugged into the HYDRA and the HYDRA powered) and download the code. Of course, you need to make sure that the programming pins from the programmer are in the right place on the HX512's programming port. Figure 25.0 shows a close up of the programming port header pin outs (which is identical to the standard 2x5 Lattice header configuration). You have to work a little to see the silk screen lettering, but its all there. Pin 1 is at the bottom left. The pin count in a stagger fashion so the bottom row is (1, 3, 5, 7, 9) left to right and the top row is (2, 4, 6, 8,10) left to right.

Figure 25.0 – The HX512's Lattice Programmer Port Pinouts.



Now, we will cover these steps in every detail.

Figure 26.0 – Locating the Magical Software Installing Menu Item.



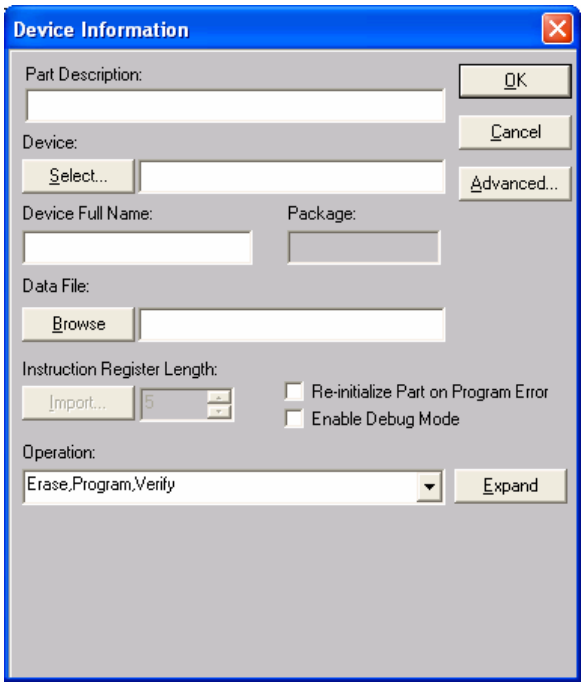
Assuming that you have bought either a USB or parallel programming cable (or made the parallel cable outlined in the appendix), you need to plug it into your computer of course. Then there is a second step where you must tell ispVM what drivers to use. To do this select **<ispTools → Install/Uninstall Win98/WinNT/2000/XP USB/Parallel Port Driver>**, as shown in Figure 26.0.

Figure 27.0 – The Lattice Semiconductor Driver Install/Uninstall Dialog.

Immediately after clicking this menu item the driver install/uninstall dialog will display as shown in Figure 27.0. Simply select ***both*** drivers, with ***"on-demand"*** start up and click **<Install>**. If all went well, you will see a **"Installation Complete/Successful"** message box, click **<OK>** and continue.

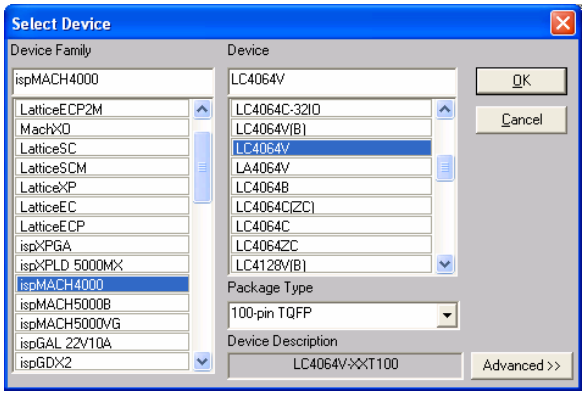
Now, that the driver(s) are installed, its time to create the **"Chain Configuration"** file. This file is a mini-project that describes the chip you are programming along with the JEDEC filename (created from your project). From the main menu bar of the ispVM System tool select **<File → New>** and you will see a blank chain configuration window created in the main view. The Chain Configuration window will have a line of text saying to "Add New Device…", to do this press the **<Insert>** key or from the main menu select **<Edit → Add Device>**, the **"Device Information"** dialog should open up which is where you need to enter all the pertinent information.

Figure 28.0 – Blank Device Information Dialog.



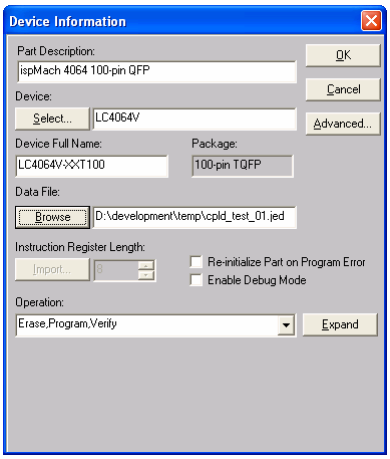
Referring to Figure 28.0, the Device Information Dialog needs to be filled out. There isn't anything too complicated, most of the stuff you have seen elsewhere. To start, the first thing that needs to be entered is the **"Part Description"**, you can enter any string you wish, but to keep synchronized with the example, go ahead and enter the string **"ispMach 4064 100-pin QFP"**. Next is the **"Device"** entry edit box, go ahead and press the **<Select>** button which will open yet another dialog box that you can use to find the exact device string for our device.

Figure 29.0 – The Device Selection Dialog Box.



Referring to Figure 29.0, select the **"Family"** as **"ispMACH4000"** and the **"Device"** as **"LC4064V"** and click **<OK>**. This will transfer the information back to the Device Information Dialog for you. Now, we are almost done. In the **"Data File"** area immediately below the **"Device Full Name"** field you need to locate the file you want to use to program the CPLD. This is simple the JEDEC file that is generated by the ispLEVER tool which is named the root project name with .JED appended to it. Go ahead and navigate with the **<Browse>** button and find the JEDEC file in your project working directory, if you used all the same names as in the example we have been working with, then the JEDEC file should be named "**cpld_test_01.jed**".

Figure 30.0 – Completed Device Information Dialog.



Finally, at the very bottom of the Device Information Dialog is the **"Operation"** selection box, the default entry is **"Erase, Program, Verify"**, this is what we want so leave it as is. Now compare your filled out dialog to the one in Figure 30.0, if everything looks the same click **<OK>**, we are done with this dialog.

Now, before moving on, let's save this configuration file -- press **<CTRL+S>** on the keyboard, or use **<File → Save>** from the main menu and a file save dialog will open. I the dialog hasn't dumped you into your project working directory, navigate to your directory and then save the configuration file as **"LC4064V.XCF"** and click **<OK>**.

1.7.4.4 Programming the CPLD

At this point, if you have been following the example, ispVM should be ready to go and our JEDEC program is ready to be downloaded into the CPLD; however, the most important thing to verify is that the programmer is connected to the HX512 properly. If you bought a Lattice programmer then you should have a 2x5 header or fly wire connector, You need to connect the programming interface lines to the HX512's programming port (they are labeled on the PCB). The signals that are needed are shown in Table 5.0.

Table 5.0 – Programmer Interface Connection from Lattice Programmer to HX512 Programming Interface.

Lattice Programmer Signal (color)		*HX512 Programmer Port Signal (pin #)*	
TCK	(White)	SCLK	(1) (bottom row, 1st pin from left)
TMS	(Purple)	TMS	(3) (bottom row, 2nd pin from left)
TDI	(Orange)	SDI	(5) (bottom row, 3rd pin from left)
TDO	(Brown)	SDO	(7) (bottom row, 4th pin from left)
VCC	(Red)	VCC	(6) (top row, 3rd pin from left)
GND	(Black)	GND	(8) (top row, 4th pin from left)
Notes: Trst/ispEN are not used and VCC is 3.3V in this case.			

Figure 31.0 – A Close up of the Programming Lines Interfaced to the HX512's Programming Port.

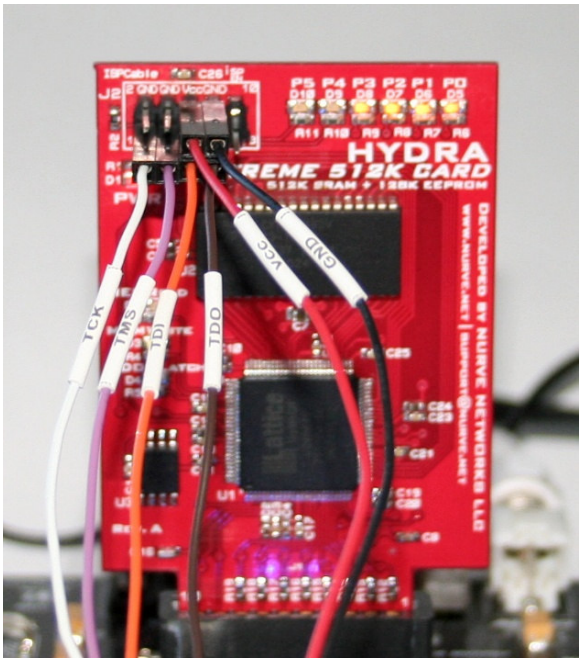


Figure 31.0 shows the programming lines inserted into the HX512's programming port. Notice that the port is numbered with the bottom row being the odd pins and the top row being the even pins, starting with pin 1 at the bottom left, thus, the bottom pins left to right are 1,3,5,7,9, the top row left to right is 2,4,6,8,10. In this case, I am use the "fly wire" single signal conductors from a Lattice programmer, notice the color coding and labels as well.

The HX512's interface header is of course compatible with the Lattice 2x5 programming header specification. These files on the CD explain the ispDownload cables (which you will get when you buy one):

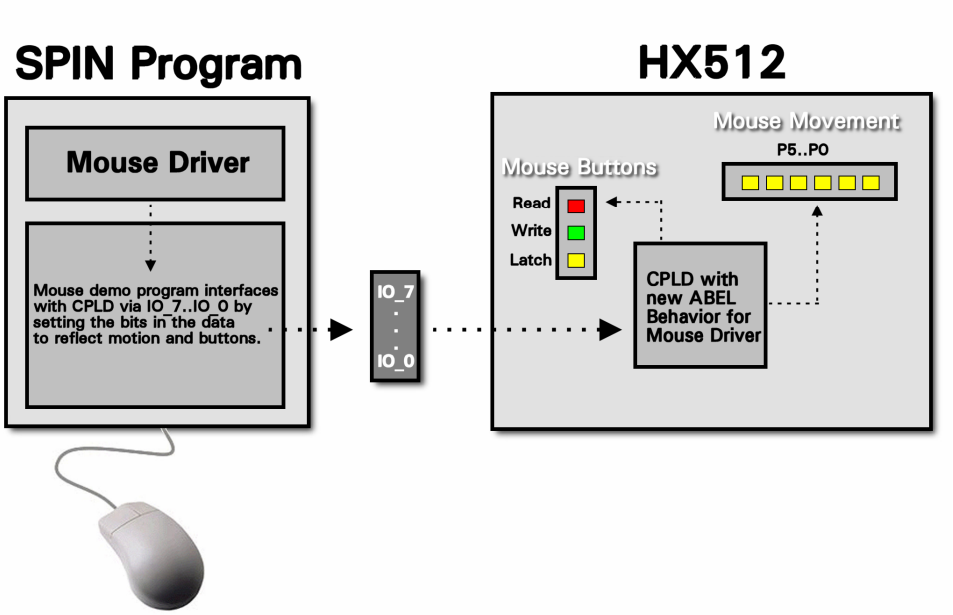
CD_ROOT:\DOCS\docs\datasheets\lattice*.*

After you have made sure to interface the proper signals with the HX512 from your programming cable (purchased or home made), then there is only one thing to do; download the code. Make sure your HYDRA is powered, it doesn't really matter what program is running, the CPLD doesn't care.

To download the JEDEC file either press the green **<GO>** button on the top toolbar of the ispVM tool or you can press **<CTRL+G>,** or you can select **<Project → Download>** from the main menu. If everything is connected, and your cable is working, then a little programming dialog with pop up and you will see a progress bar and the programming should take 3-4 seconds. When complete, you will see a **"Pass"** or **"Fail"** next to your configuration file entry in the ispVM window. Hopefully, you see a **"Pass"**. If you see fail, check cabling, power, etc. retry, if fails again, review the steps from the procedure, make sure you have the right chip and JEDEC file and try again.

That concludes re-programming the HX512 with the memory controller driver.

Figure 32.0 – A Block Diagram of the Mouse CPLD Demo.



1.7.5 Changing the Firmware – Mouse Light Show Demo

As a concrete, but simple demo of how to change the behavior of the CPLD and do something interesting, I have created a simple CPLD program that allows you to move the mouse around and press the buttons and the actions are displayed on the HX512 informational LEDs. Take a look at Figure 32.0 for a block diagram of what's going on.

There are two pieces to this application; the ABEL code that is downloaded into the CPLD and the Propeller program that runs on the HYDRA and interfaces to the HX512 board's CPLD via the HYDRA expansion port. The idea of the demo is simple; the demo boots up on the CPLD and watches the data bus bits (I/O_7...I/O_0), if bits I/O_1 or I/O_0 are active then it moves a little dot on the LEDs right/left (this is accomplished by a shift register in the CPLD program). Secondly, if the mouse buttons are pressed (left, right, middle) they are instantly indicated by the read / write / latch LEDs on the HX512.

These the demo program(s) show off how to create a new behavior and interface to it. This illustrates using CPLD for experimentation and other things other than a memory controller. The two files you will need are the SPIN demo which is located on the CD here:

CD_ROOT:\SOURCES\HX512K_MOUSE_DEMO_010.SPIN

and the ABEL program which is located on the CD here:

CD_ROOT:\SOURCES\mousetracker_01.abl

You can compile the program yourself as outlined in the previous sections, or you can just use the pre-compiled JEDEC file I have provided here:

CD_ROOT:\SOURCES\mousetracker_01.jed

TIP	If you compile the ABEL program and generate a JEDEC file, make sure to select the correct target chip, and make sure to alter the constraint file for LVCMOS33 and SLOW slew rate.

The sources for both the CPLD driver and the SPIN demo are rather short and are listed below for convenience. First, let's look at the ABLE CPLD driver shown in Listing 2.0 below:

Listing 2.0 – The CPLD Driver for the Mouse Demo.

```
MODULE MouseTracker_01

TITLE 'Mouse Tracking Demo'
// This fun demo shows how to use the CPLD on the HX512 for things other than a memory controller.
// In this case, the HYDRA application is going to track the mouse buttons, right and left motion.
// The 6 program LEDs will track the right/left motion of the mouse, while the 3 operation leds
// will track the buttons.

// HYDRA Expansion interface Signals
// IO_0 - Right                                 (pin 23 on Propeller).
// IO_1 - Left                                  (pin 22 on Propeller).
// IO_2 - Up    (unused by CPLD in this demo)   (pin 21 on Propeller).
// IO_3 - Down  (unused by CPLD in this demo)   (pin 20 on Propeller).
// IO_4 - Mouse Left Button                     (pin 19 on Propeller).
// IO_5 - Mouse Middle Button                   (pin 18 on Propeller).
// IO_6 - Mouse Right Button                    (pin 17 on Propeller).
// IO_7 - Unused                                (pin 16 on Propeller).
// RESn - Reset Line                            (pin xx on Propeller).
// USB_RXD - Clock                              (pin 30 on Propeller).

DECLARATIONS

    // inputs
    clock       pin 38;             // general system clock coming from USB_RXD on HYDRA
    !reset      pin 8;              // hydra RESn reset line (active low), notice inversion to make active high
    io7..io0    pin 15..17,19..23;  // data bus lines from HYDRA, IO_7..IO_0

    // outputs
    led5..led0 pin 85, 87, 14, 35, 36, 37 istype 'reg'; // the mouse tracking lights, use the program LEDs

    // create a set for the mouse direction leds
    move_leds = [led5..led0];

    // define leds for mouse buttons
    led_read  pin 78;
    led_write pin 79;
    led_latch pin 80;

EQUATIONS

    // set clock
    move_leds.clk = clock;

    // simple combinational logic for buttons left, middle, right, are mapped
    // top to botton on the read, write, latch leds
    led_read  = io4;
    led_write = io5;
    led_latch = io6;

    // test for reset, if so center the little tracking led
    when (move_leds == [0,0,0,0,0,0]) then { move_leds = [0,0,1,0,0,0]; }

    // test for mouse move right? If so shift right
    else when (io0 == 1) then { move_leds = [led0, led5, led4, led3, led2, led1]; }

    // test for mouse move left? If so shift left
    else when (io1 == 1) then { move_leds = [led4, led3, led2, led1, led0, led5]; }

    else { move_leds = move_leds; }

END
```

As you can see, barely a page long. Next, Listing 3.0 shows the SPIN based demo that makes calls to the CPLD via the HYDRA expansion interface:

Listing 3.0 – SPIN Driver for Mouse Demo.

```
CON

  _clkmode = xtal2 + pll4x            ' enable external clock and pll times 4
  _xinfreq = 10_000_000 + 0000        ' set frequency to 10 MHZ plus some error

  ' SRAM bus interface pin constants
  SRAM_CTRL_0 = 1 ' NET_RX_CLK       (expansion pin 10)
  SRAM_CTRL_1 = 2 ' NET_TX_DATA      (expansion pin 9)

  SRAM_STROBE = 30 ' USB_RXD (Prop TX ----> USB_RXD Host) (expansion pin 19)

  SRAM_IO_7 = 23 ' IO_7 (pin 28)
  SRAM_IO_6 = 22
  SRAM_IO_5 = 21
  SRAM_IO_4 = 20
  SRAM_IO_3 = 19
  SRAM_IO_2 = 18
  SRAM_IO_1 = 17
  SRAM_IO_0 = 16  ' IO_0 (pin 21)

  ' mouse buttons
  MOUSE_MIDDLE   = 2
  MOUSE_RIGHT    = 1
  MOUSE_LEFT     = 0

OBJ
  mouse : "mouse_iso_010.spin"       ' instantiate a mouse object

PUB Start

  'start mouse on pingroup 2 (Hydra mouse port)
  mouse.start(2)

  ' set the data bus and clock to outputs, so we can talk to CPLD
  InitalizeIO
```

```
  ' main loop, track mouse and buttons and send to CPLD interface as agreed up in interface spec
  repeat

    ' test for right/left movement
    if (mouse.delta_x > 0)
      OUTA[ SRAM_IO_0 ] := 1
      OUTA[ SRAM_IO_1 ] := 0
    elseif (mouse.delta_x < 0)
      OUTA[ SRAM_IO_1 ] := 1
      OUTA[ SRAM_IO_0 ] := 0
    else
      OUTA[ SRAM_IO_0 ] := 0
      OUTA[ SRAM_IO_1 ] := 0

    ' test for buttons
    OUTA[ SRAM_IO_4 ] := mouse.button(MOUSE_LEFT)
    OUTA[ SRAM_IO_5 ] := mouse.button(MOUSE_MIDDLE)
    OUTA[ SRAM_IO_6 ] := mouse.button(MOUSE_RIGHT)

    ' the CPLD needs a heartbeat for all the registered logic
    Pulse_Clock

' //////////////////////////////////////////////////////////////////////

PUB Pulse_Clock
' Pulses the clock line on the CPLD which is ultimately attached to pin 38 of the CPLD

  OUTA[ SRAM_STROBE ] := $01
  OUTA[ SRAM_STROBE ] := $00

' //////////////////////////////////////////////////////////////////////

PUB InitalizeIO
' Initializes the IO for HYDRA<->CPLD interface,
' in this case we just need to set the data bus to output as well as the clock strobe line

  ' set bus I/O directions for data bus
  OUTA[ SRAM_IO_7..SRAM_IO_0 ] := $00
  DIRA[ SRAM_IO_7..SRAM_IO_0 ] := $FF            ' $FF ouput, $00 input

  ' set strobe
  OUTA[ SRAM_STROBE ]          := %0000000_0     ' clear strobe
  DIRA[ SRAM_STROBE ]          := $01            ' set to output
```

1.8 Summary

The HX512 enhances the HYDRA's abilities substantially by allowing more complex applications to be developed. Moreover, with re-programming of the CPLD other personalities and behaviors can be programmed such as crude bit blitting, simple DSP algorithms and much more. We hope to see many amazing applications developed with the HYDRA XTREME 512K SRAM Card!

Appendices

The following appendices contain very useful information including schematics, drivers, re-programming the CPLD and more.

A. HX512 Circuit Schematics
B. Lattice ispMach 4064 Details and Signal Descriptions
C. Building Your Own Lattice ISP Programmer
D. Using the HX512 without the HYDRA
E. HX512 API Driver sources

Appendix A. HX512 Circuit Schematics

The HX512 consists of a number of passive components; resistors, capacitors, and LEDs as well as three primary VLSI chips composed of the 512K SRAM, CPLD, and 128K EEPROM. The manufacturer of the 512K SRAM and 128K EEPROM may very, but the CPLD will always be a Lattice ispMach 4064 100-pin QFP. For reference, the part numbers for all three chips are:

- 512K SRAM – Cypress Semiconductor – CY7C1049DV33 10VXI , 36-pin SOJ (or drop in replacement 10-12 ns).
- 128K Serial EEPROM – Atmel Corp – 24C1024W SU27, 8-pin SOIC (or drop in replacement).
- CPLD – Lattice Semiconductor LC4064V-75T100-10I, 100-pin QFP (no substitutes).

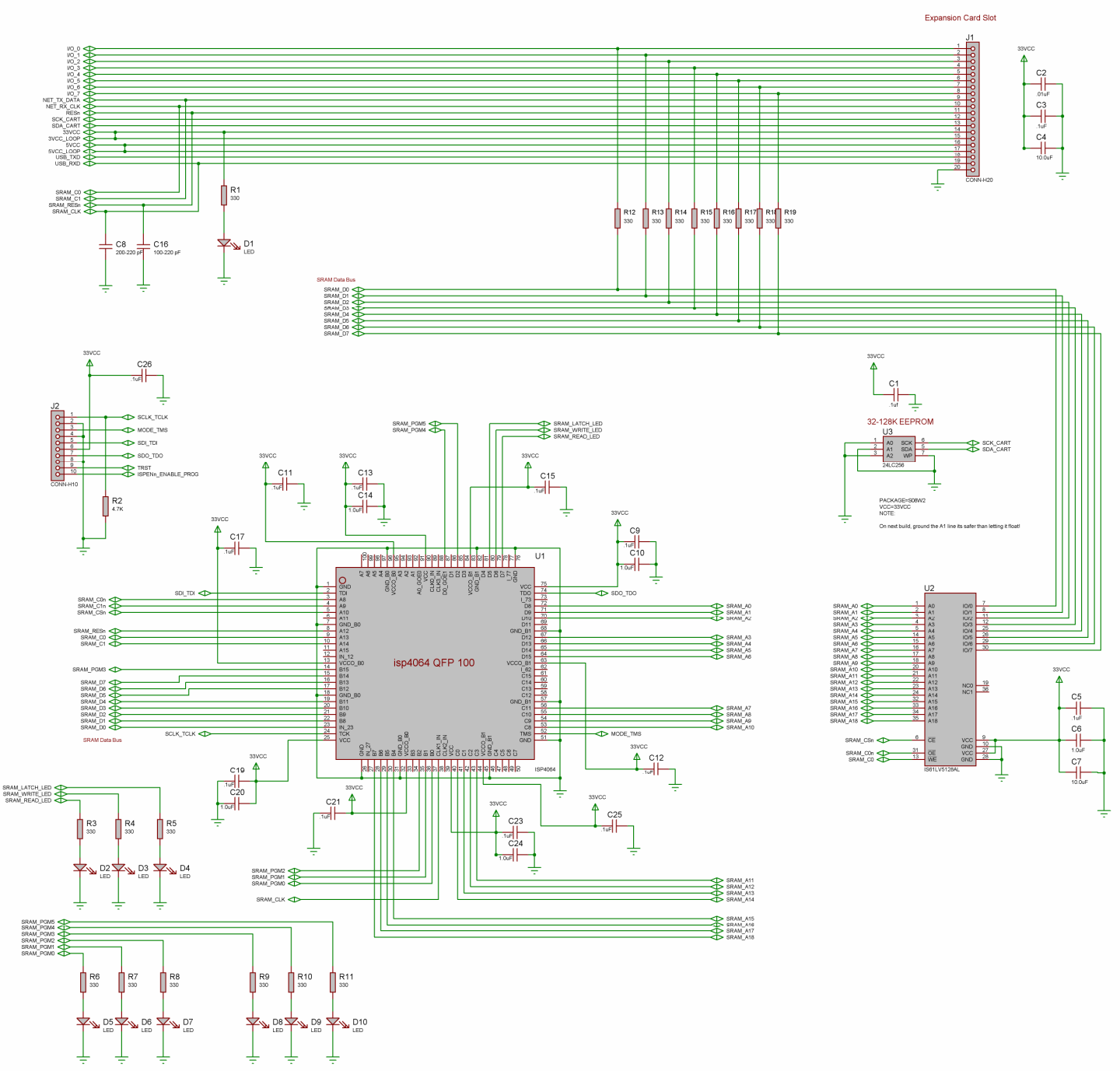
Datasheets for all these parts can be found on the CD in the following location:

CD_ROOT:\DOCS\DATASHEETS*.*

Figure 33.0 is the complete design of the HYDRA XTREME 512K SRAM Memory Card. For closer inspection, a bitmap version of the file is located on the CD here:

CD_ROOT:\SCHEMATICS\HX512CARD_schematic_01.png

Figure 33.0 – Schematic for HYDRA XTREME 512K Memory Card.



Note: All passive components are surface mount size 0603, except C7 which is 0805 size.

Appendix B. Lattice ispMach 4064 Details and Signal Descriptions

The Lattice CPLD used in the HX512 is a member of the ispMACH 4000 series part # **LC4064V-75T100C**. The part # is decoded as follows:

"4064" - Means there are 64 logic blocks.
"75T" - Means there is a 7.5 ns propagation delay.
"V" - Means supports 3.3V/5V I/O, but the core is always 3.3V.
"100" - Means the chip is a 100-pin TQFP.

There are numerous documents on the Lattice Semiconductor website at the following URL location:

http://www.latticesemi.com/products/cpldspld/ispmach4000bcv.cfm

However, to save you time, you can find the data sheets and other important documents on the CD in the following location:

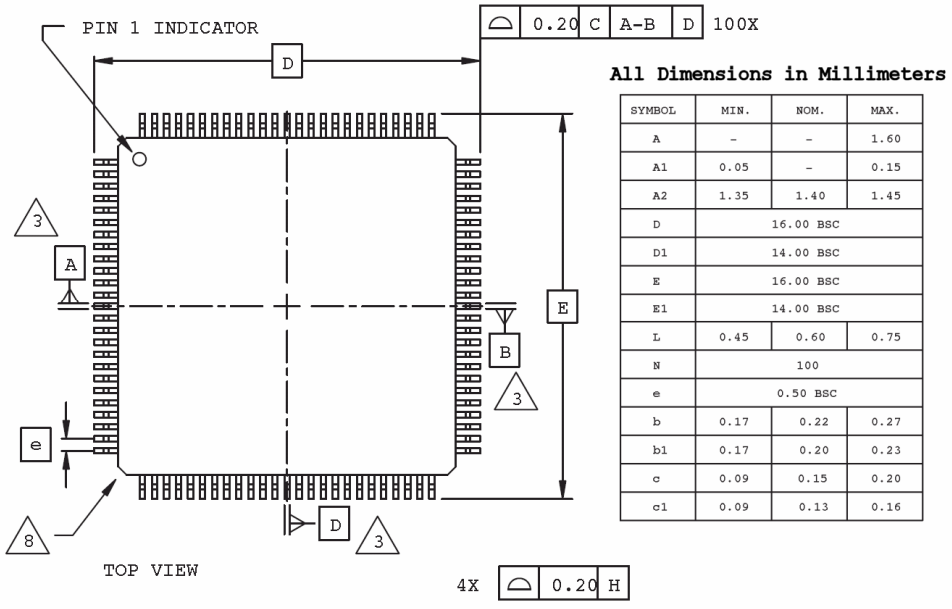
CD_ROOT:\DOCS\DATASHEETS\LATTICE

The most important document to read is the file **ISPM4K.PDF**, it describes in detail the architecture of the CPLD family and how they work.

ispMach 4064 Signal Descriptions

The pinout of the 4064 is shown in the previous schematic figure, so no need to replicate that here. If you are interested in the exact mechanical specifications of the 100-pin TQFP then refer to document **LATTICE_PKG.PDF** within the aforementioned directory. It contains all of the mechanical specifications for all the chips in the family including the 100-pin version used in the HX512. Figure 34.0 shows an abridged mechanical diagram for reference. The pins are labeled 1 to 100 in a counter clockwise fashion. The pin classes are; power, I/O, clock, and programming. Table 6.0 lists each pin on the 100-pin 4064 (refer to the leftmost pane that is highlighted).

Figure 34.0 – The ispMach 4064 100-Pin Package Mechanical Drawing.



All Dimensions in Millimeters

SYMBOL	MIN.	NOM.	MAX.
A	-	-	1.60
A1	0.05	-	0.15
A2	1.35	1.40	1.45
D		16.00 BSC	
D1		14.00 BSC	
E		16.00 BSC	
E1		14.00 BSC	
L	0.45	0.60	0.75
N		100	
e		0.50 BSC	
b	0.17	0.22	0.27
b1	0.17	0.20	0.23
c	0.09	0.15	0.20
c1	0.09	0.13	0.16

Table 6.0 – isp4064 100-pin QFP Pin Descriptions.

Pin Number	Bank Number	ispMACH 4064V/B/C/Z		ispMACH 4128V/B/C/Z		ispMACH 4256V/B/C/Z	
		GLB/MC/Pad	ORP	GLB/MC/Pad	ORP	GLB/MC/Pad	ORP
1	-	GND	-	GND	-	GND	-
2	-	TDI	-	TDI	-	TDI	-
3	0	A8	A^8	B0	B^0	C12	C^3
4	0	A9	A^9	B2	B^1	C10	C^2
5	0	A10	A^10	B4	B^2	C6	C^1
6	0	A11	A^11	B6	B^3	C2	C^0
7	0	GND (Bank 0)	-	GND (Bank 0)	-	GND (Bank 0)	-
8	0	A12	A^12	B8	B^4	D12	D^3
9	0	A13	A^13	B10	B^5	D10	D^2
10	0	A14	A^14	B12	B^6	D6	D^1
11	0	A15	A^15	B13	B^7	D4	D^0
12*	0	I	-	I	-	I	-
13	0	VCCO (Bank 0)	-	VCCO (Bank 0)	-	VCCO (Bank 0)	-
14	0	B15	B^15	C14	C^7	E4	E^0
15	0	B14	B^14	C12	C^6	E6	E^1
16	0	B13	B^13	C10	C^5	E10	E^2
17	0	B12	B^12	C8	C^4	E12	E^3
18	0	GND (Bank 0)	-	GND (Bank 0)	-	GND (Bank 0)	-
19	0	B11	B^11	C6	C^3	F2	F^0
20	0	B10	B^10	C5	C^2	F6	F^1
21	0	B9	B^9	C4	C^1	F10	F^2
22	0	B8	B^8	C2	C^0	F12	F^3
23*	0	I	-	I	-	I	-
24	-	TCK	-	TCK	-	TCK	-
25	-	VCC	-	VCC	-	VCC	-
26	-	GND	-	GND	-	GND	-
27*	0	I	-	I	-	I	-
28	0	B7	B^7	D13	D^7	G12	G^3
29	0	B6	B^6	D12	D^6	G10	G^2
30	0	B5	B^5	D10	D^5	G6	G^1
31	0	B4	B^4	D8	D^4	G2	G^0
32	0	GND (Bank 0)	-	GND (Bank 0)	-	GND (Bank 0)	-
33	0	VCCO (Bank 0)	-	VCCO (Bank 0)	-	VCCO (Bank 0)	-
34	0	B3	B^3	D6	D^3	H12	H^3
35	0	B2	B^2	D4	D^2	H10	H^2
36	0	B1	B^1	D2	D^1	H6	H^1
37	0	B0	B^0	D0	D^0	H2	H^0
38	0	CLK1/I	-	CLK1/I	-	CLK1/I	-
39	1	CLK2/I	-	CLK2/I	-	CLK2/I	-
40	-	VCC	-	VCC	-	VCC	-
41	1	C0	C^0	E0	E^0	I2	I^0

Table 6.0 – isp4064 100-pin QFP Pin Descriptions (continued).

Pin Number	Bank Number	ispMACH 4064V/B/C/Z		ispMACH 4128V/B/C/Z		ispMACH 4256V/B/C/Z	
		GLB/MC/Pad	ORP	GLB/MC/Pad	ORP	GLB/MC/Pad	ORP
42	1	C1	C^1	E2	E^1	I6	I^1
43	1	C2	C^2	E4	E^2	I10	I^2
44	1	C3	C^3	E6	E^3	I12	I^3
45	1	VCCO (Bank 1)	-	VCCO (Bank 1)	-	VCCO (Bank 1)	-
46	1	GND (Bank 1)	-	GND (Bank 1)	-	GND (Bank 1)	-
47	1	C4	C^4	E8	E^4	J2	J^0
48	1	C5	C^5	E10	E^5	J6	J^1
49	1	C6	C^6	E12	E^6	J10	J^2
50	1	C7	C^7	E14	E^7	J12	J^3
51	-	GND	-	GND	-	GND	-
52	-	TMS	-	TMS	-	TMS	-
53	1	C8	C^8	F0	F^0	K12	K^3
54	1	C9	C^9	F2	F^1	K10	K^2
55	1	C10	C^10	F4	F^2	K6	K^1
56	1	C11	C^11	F6	F^3	K2	K^0
57	1	GND (Bank 1)	-	GND (Bank 1)	-	GND (Bank 1)	-
58	1	C12	C^12	F8	F^4	L12	L^3
59	1	C13	C^13	F10	F^5	L10	L^2
60	1	C14	C^14	F12	F^6	L6	L^1
61	1	C15	C^15	F13	F^7	L4	L^0
62*	1	I	-	I	-	I	-
63	1	VCCO (Bank 1)	-	VCCO (Bank 1)	-	VCCO (Bank 1)	-
64	1	D15	D^15	G14	G^7	M4	M^0
65	1	D14	D^14	G12	G^6	M6	M^1
66	1	D13	D^13	G10	G^5	M10	M^2
67	1	D12	D^12	G8	G^4	M12	M^3
68	1	GND (Bank 1)	-	GND (Bank 1)	-	GND (Bank 1)	-
69	1	D11	D^11	G6	G^3	N2	N^0
70	1	D10	D^10	G5	G^2	N6	N^1
71	1	D9	D^9	G4	G^1	N10	N^2
72	1	D8	D^8	G2	G^0	N12	N^3
73*	1	I	-	I	-	I	-
74	-	TDO	-	TDO	-	TDO	-
75	-	VCC	-	VCC	-	VCC	-
76	-	GND	-	GND	-	GND	-
77*	1	I	-	I	-	I	-
78	1	D7	D^7	H13	H^7	O12	O^3
79	1	D6	D^6	H12	H^6	O10	O^2
80	1	D5	D^5	H10	H^5	O6	O^1
81	1	D4	D^4	H8	H^4	O2	O^0
82	1	GND (Bank 1)	-	GND (Bank 1)	-	GND (Bank 1)	-

Table 6.0 – isp4064 100-pin QFP Pin Descriptions (continued).

Pin Number	Bank Number	ispMACH 4064V/B/C/Z		ispMACH 4128V/B/C/Z		ispMACH 4256V/B/C/Z	
		GLB/MC/Pad	ORP	GLB/MC/Pad	ORP	GLB/MC/Pad	ORP
83	1	VCCO (Bank 1)	-	VCCO (Bank 1)	-	VCCO (Bank 1)	-
84	1	D3	D^3	H6	H^3	P12	P^3
85	1	D2	D^2	H4	H^2	P10	P^2
86	1	D1	D^1	H2	H^1	P6	P^1
87	1	D0/GOE1	D^0	H0/GOE1	H^0	P2/OE1	P^0
88	1	CLK3/I	-	CLK3/I	-	CLK3/I	-
89	0	CLK0/I	-	CLK0/I	-	CLK0/I	-
90	-	VCC	-	VCC	-	VCC	-
91	0	A0/GOE0	A^0	A0/GOE0	A^0	A2/GOE0	A^0
92	0	A1	A^1	A2	A^1	A6	A^1
93	0	A2	A^2	A4	A^2	A10	A^2
94	0	A3	A^3	A6	A^3	A12	A^3
95	0	VCCO (Bank 0)	-	VCCO (Bank 0)	-	VCCO (Bank 0)	-
96	0	GND (Bank 0)	-	GND (Bank 0)	-	GND (Bank 0)	-
97	0	A4	A^4	A8	A^4	B2	B^0
98	0	A5	A^5	A10	A^5	B6	B^1
99	0	A6	A^6	A12	A^6	B10	B^2
100	0	A7	A^7	A14	A^7	B12	B^3

*This pin is input only.

Appendix C. Building Your Own Lattice ISP Programmer

As mentioned before you can buy a Lattice ISP programmer from Lattice rather than build one. Here is the information and links once again:

ISP Download Cables for PCs

http://www.latticesemi.com/products/developmenthardware/programmingcables.cfm

The specific part #'s for the parallel port version and USB version are as follows:

- ispDOWNLOAD Cable Parallel Port (HW-DLN-3C) - $65.00
- ispDOWNLOAD Cable USB (HW-USBN-2A) - $149.00 (recommended if you can afford it)

And here's a link to the online store where you can buy either:

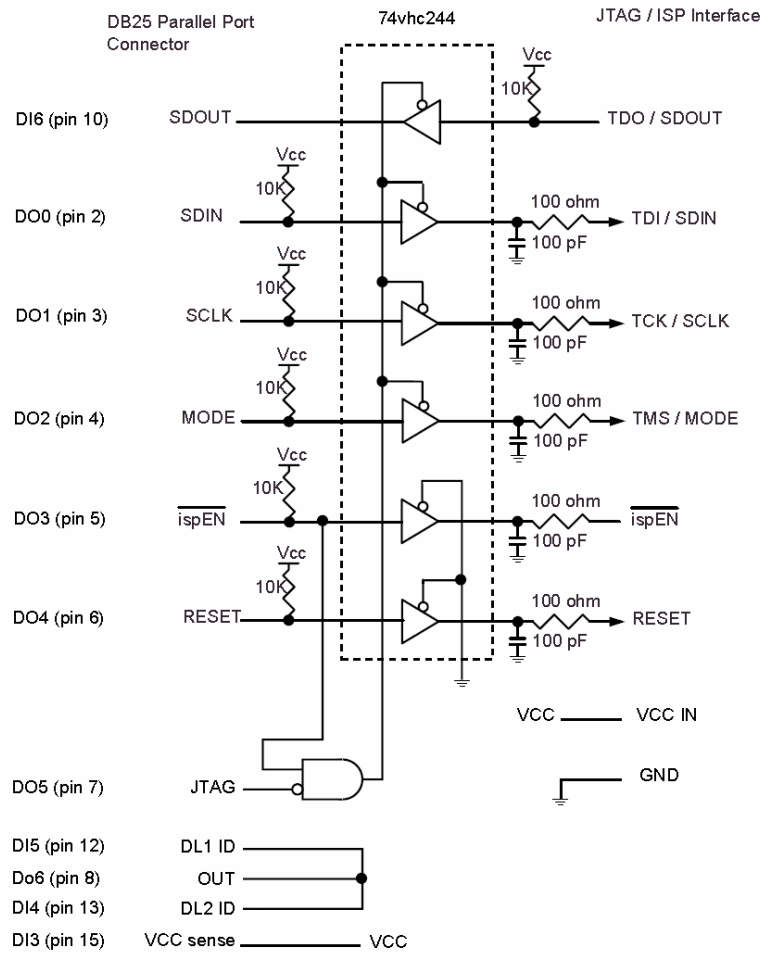
http://www.latticesemi.com/store/hardware.cfm

However, it's a little cheaper to make one yourself if you have the time and patience. Plus, if you ever decide to do more with CPLDs then its always good to learn how to build programmers since you can build them right into your designs. We are going to cover the designs for two different ***parallel port*** based programmers (USB is much too complex). If you build either of them then **ispLEVER** will detect them as if it would a real Lattice parallel port programmer and you will be in business.

Lattice Reference Design

The first design is based on a Lattice Application Note. Figure 35.0 shows the general design for a DIY parallel port programmer.

Figure 35.0 – Lattice Designed Parallel Port Programmer.



You can build this programmer from a single 74HC244, AND gate, resistors, capacitors, and a couple connectors. The actual application note is located on the CD here:

CD_ROOT:\DOCS\DATASHEETS\LATTICE\lattice_ispdownload.pdf

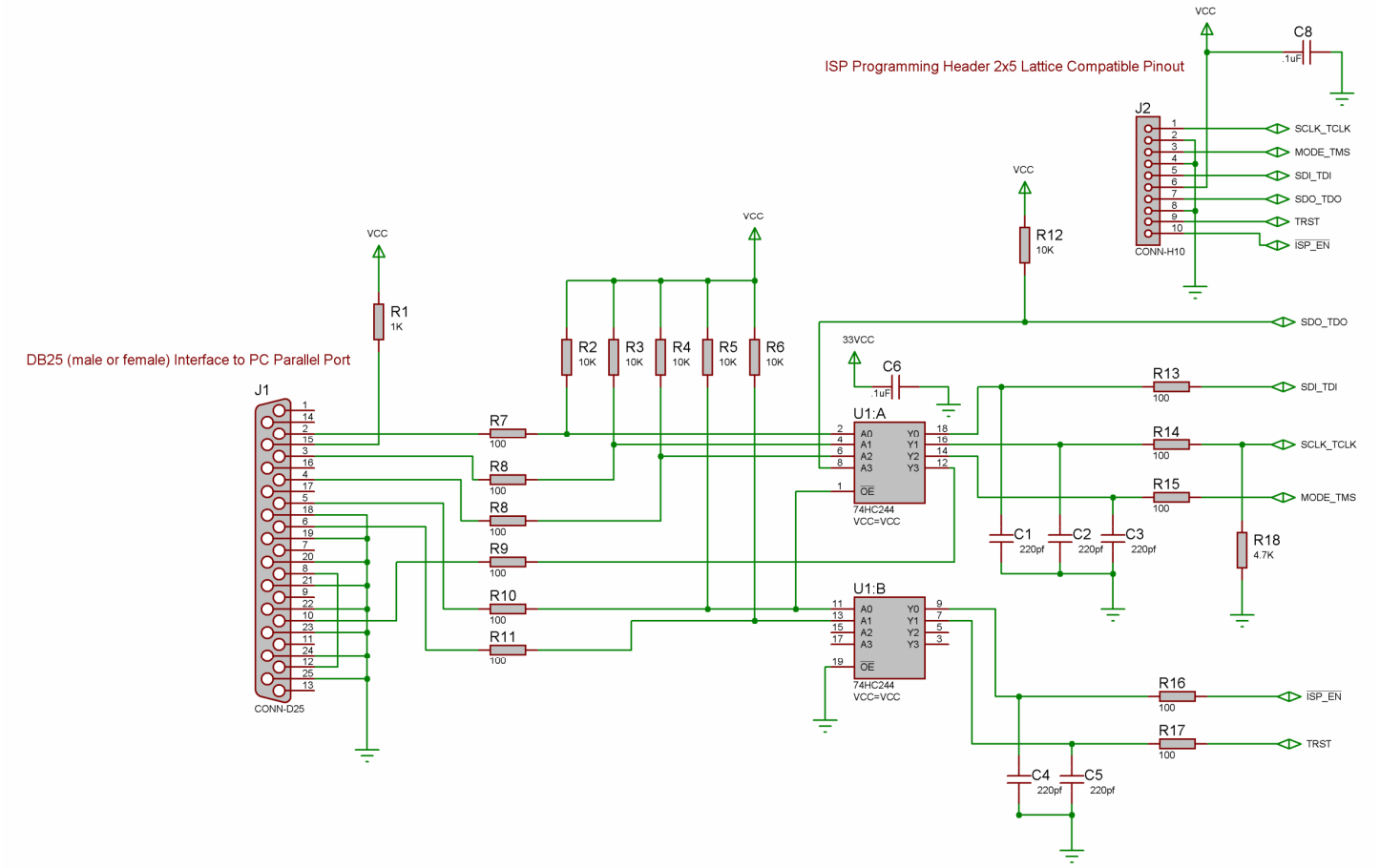
If you build this design, then you will need to make sure that the operating VCC of the circuit is the same as the programming voltage of the CPLD (usually 3.3V). Thus, you would power the programmer from either the target board's VCC supply of 3.3V OR you would power the target board from the programmers 3.3V supply. Bottom line, is someone has to provide power and it has to be the right voltage.

TIP	Usually the CPLD core operating voltage is the programming voltage.

A Sample Detailed Design

Based on the Lattice design and my own research, I was able to remove the single AND gate and get the part count down on the design. Figure 36.0 are the results of my design for yet another parallel port programmer that you can build.

Figure 36.0 – Detailed ISP Programmer with AND Gate Removed.



A copy of the design in bitmap form is on the CD here:

CD_ROOT:\ SCHEMATICS\isp4064_programmer_schematic_01.png

The parts list or **"Bill of Materials"** (BOM) is shown in Table 7.0.

Table 7.0 – Bill of Materials for Home Made Lattice ISP Programmer.

Reference ID	*Value/Type*	*Description*
Connectors		
J1	DB25	DB25 Female (or male) Solder or PCB mount.
J2		
Resistors		
R1-R6, R12	10K Ohm	10K resistor 1/4th watt, 5% tolerance.
R7-R11, R13-R17	100 Ohm	100 resistor 1/4th watt, 5% tolerance.
Capacitors		
C1-C5	220pF	220 pico Farad, ceramic capacitor.
ICs		
U1	74HC244	20-pin Octal tri-state buffer.

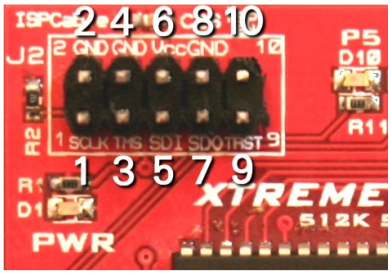
Final Construction Tips and Tricks

When building the design, you can use either a male or female DB25 connector. The important thing is to realize that the PC's parallel port is FEMALE, so you need to make sure you have a cable that can go from the PC to the programmer. I have a lot of MALE-MALE cables, so I tend to design FEMALE DB25s onto everything.

The voltage issue was brought up in the previous section -- either the target board or the programmer has to supply the power to the system. And that voltage must match the required programming voltage of the CPLD (which is usually the core voltage of the CPLD, in our case 3.3V, but some cores run at 1.8V, an others at 5.0V). From what I can tell from Lattice support and documentation, even if you have the I/O pads powered at 5.0V its irrelevant, the core voltage is what the I/O pads on the programming pins TDI, TDO, CLK, TMS, etc. use.

Finally, the output header configuration is also always a question, so I suggest using a standard 2x5 header and connect the programming signals to it exactly the same as Lattice does this way, its easy to bus the signals using a 2x5 ribbon cable from your programmer to the HX512.. Figure 37.0 shows a top view of how the signals should be, this is consistent with Lattice's recommendation and the HX512's header is laid out this way. So I suggest you use this configuration.

Figure 37.0 – Programming Header Pinout Top View.



If you use this 2x5 header configuration then you can buy a nice 2x5 header cable straight thru parallel from Digikey or other vendors. For example, here's a TYCO Electronics part that works perfectly that you can look up on www.digikey.com:

Part # A3AAG-1018G-ND (18" model)

Appendix D. Using the HX512 without the HYDRA

There is absolutely no reason why you can't use the HX512 with other devices or with other microcontroller based products that you want to add external memory to. All you really need is the 20-pin edge card interface and you are up and running. The HX512 card's interface is mechanically 20-contacts with 10-contacts per side and 0.1" spacing between contact centers. The depth the card inserts into the edge connector is approximately 5/16". The pins are numbered from 1 to 10, right to left (pin 1 is on the far right with the card facing you), and pins 11-20 on the back side numbered left to right. Figure 38.0 shows a top view of the expansion port on the HYDRA for reference and Figure 39.0 shows the simple circuit diagram of the interface.

Figure 38.0 – Top View of the HYDRA Expansion Port Interface.

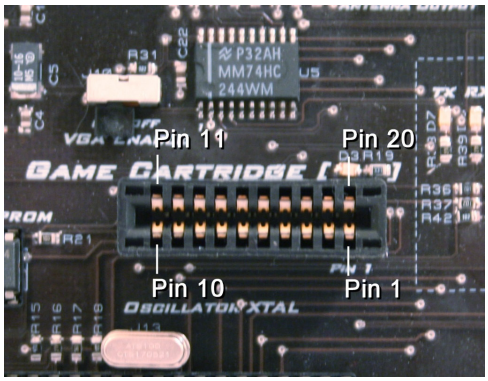
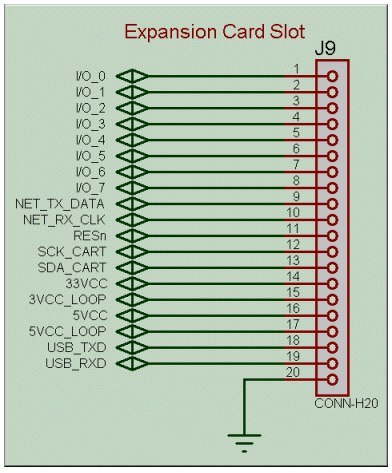


Figure 39.0 – Circuit Diagram of the Expansion Port Signals.



The only thing about the card that is potentially wasteful is the EEPROM, but you can surely find a use for it on your other projects. Thus, with a simple 20-pin edge connector you can interface the HX512 to your other projects and get a SRAM interface with only a few lines, plus a CPLD!

Appendix E. HX512 API Driver sources

Listing 4.0 below is a complete listing of the ASM driver. The ASM driver is listed since its very complex, but worth the time to understand it. It not only serves as the driver for the HX512, but shows many advanced techniques that you can use to write your own drivers for other applications such as graphics, math, TV/VGA generation and more.

Listing 4.0 – The ASM HX512 Driver.

```
'////////////////////////////////////////////////////////////////////////
' CONSTANTS SECTION ///////////////////////////////////////////////////////
'////////////////////////////////////////////////////////////////////////

CON


  ' SRAM bus interface pin constants
  SRAM_CTRL_0 = 1 ' NET_RX_CLK       (expansion pin 10)
  SRAM_CTRL_1 = 2 ' NET_TX_DATA      (expansion pin 9)

  SRAM_STROBE = 30 ' USB_RXD (Prop TX ----> USB_RXD Host) (expansion pin 19)

  SRAM_IO_7 = 23 ' IO_7 (pin 28)
  SRAM_IO_6 = 22
  SRAM_IO_5 = 21
  SRAM_IO_4 = 20
  SRAM_IO_3 = 19
  SRAM_IO_2 = 18
  SRAM_IO_1 = 17
  SRAM_IO_0 = 16  ' IO_0 (pin 21)

    ' sram commands
  SRAM_CMD_WRITE   = %000000_00
  SRAM_CMD_READ    = %000000_01
  SRAM_CMD_LOADLO  = %000000_10
  SRAM_CMD_LOADHI  = %000000_11


'////////////////////////////////////////////////////////////////////////////
' VARIABLES SECTION ///////////////////////////////////////////////////////////
'////////////////////////////////////////////////////////////////////////////

VAR

  long  cogon, cog

  ' sram parameter passing area for ASM driver, this starting address always holds, the command, parameter ptr, and return value, in that order
  long sram_command             ' holds command to SRAM driver, also the starting address of this LONG is assumed to be the start of all parms
  long sram_parameter_list_ptr  ' holds pointer to parameter list to driver
  long sram_return_value        ' holds result of sub-function (if there is one)

'////////////////////////////////////////////////////////////////////////////
' OBJECT DECLARATION SECTION //////////////////////////////////////////////////
'////////////////////////////////////////////////////////////////////////////


'////////////////////////////////////////////////////////////////////////////
' PUBLIC FUNCTIONS ////////////////////////////////////////////////////////////
'////////////////////////////////////////////////////////////////////////////

PUB SRAM_Start_ASM_Driver(sram_init_program)
' this function starts the ASM SRAM driver up and sends the 4-bit initialization program code to it as well as initializes the IO pins for
' proper operation the control word or "program" instructs the SRAM controller to either post inc/dec on reads/write or neither

' Parameters: sram_init_program - this data word (lower 4-bits only) is used to program the behavior of the controller, see below
'
'
'  4-bit format
'
'  |  3   2   |  1   0
'  |  sr  r0  |  sw  w0
'  pgm3.............pgm0
'
'  sw   - sign bit for write post increment/decrement (1=add, 0=subtract).
'  w0   - 1 bit magnitude for write post increment/decrement (w1 ignored in this version).
'
'  sr   - sign bit for read post increment/decrement  (1=add, 0=subtract).
'  r0   - 1 bit magnitude for read post increment/decrement (r1 ignored in this version).
'
'
'  In most cases, its recommended that controller is initialized with both post increment on read/write
'  which is the value %0000_1111, these program bits will show up on the LEDs to the top right of the SRAM card

  ' if the driver is running kill it, however, there is no way to reset the controller, so the program loaded into the program
  ' from RESET will remain there until another reset
  SRAM_Stop_ASM_Driver

  ' set command in global shared variable
  sram_command            := sram_init_program
  ' set starting address of parameters passed to sub-functions, this is a pointer to pointer, in this case NULL since this operation
  ' has no parms
  sram_parameter_list_ptr := 0

  ' start the driver, return status, set up cog id variables
  return ( cogon := (cog := cognew(@SRAM_Driver_Entry, @sram_command)) > 0 )

'////////////////////////////////////////////////////////////////////////////

PUB SRAM_Stop_ASM_Driver
' stops sram driver
'
' Parameters: none.

  if cogon~
    cogstop(cog)
```

```
'/////////////////////////////////////////////////////////////////////////////
' These functions makes up the public interface to the ASM SRAM driver, the client
' calls the functions by setting globals that are being "monitored" by driver
'/////////////////////////////////////////////////////////////////////////////

PUB SRAM_Write64K_A(addr16, data8)
' this function writes 8-bit data to the first 64K of the SRAM
'
' Parameters: addr16 - 16-bit address to write to
'             data8  - 8-bit data to write

  sram_parameter_list_ptr := @addr16
  sram_command            := _Write64K ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

'/////////////////////////////////////////////////////////////////////////////

PUB SRAM_WriteAuto_A(data8)
' this function writes data to the SRAM at the current address ptr, then the SRAM controller updates the ptr
' depending on its initially programmed inc/dec behavior
'
' Parameters: data8  - 8-bit data to write
'

  sram_parameter_list_ptr := @data8
  sram_command            := _WriteAuto ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

'/////////////////////////////////////////////////////////////////////////////

PUB SRAM_Read64K_A(addr16)
' this function reads a byte of data from the first 64K of the SRAM
'
' Parameters: addr16 - 16-bit address to read from

  sram_parameter_list_ptr := @addr16
  sram_command            := _Read64K ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

  return sram_return_value

'/////////////////////////////////////////////////////////////////////////////

PUB SRAM_ReadAuto_A
' this function reads a byte of data from the currently addressed byte in the SRAM by the address pointer
' if the SRAM controller is programmed for auto inc/dec after read then it will take place automatically and the
' address pointer will be updated by the SRAM controller
'
' Parameters: None.

  sram_parameter_list_ptr := 0
  sram_command            := _ReadAuto ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

  return sram_return_value

'/////////////////////////////////////////////////////////////////////////////

PUB SRAM_LoadAddr64K_A(addr16)
' this function sets the SRAM controllers address latch to the sent 16-bit address, clears the upper 3-bits of the address as well
'
' Parameters: addr16 - 16-bit address to set the SRAM address latch to, upper 3-bits is zero'ed

  sram_parameter_list_ptr := @addr16
  sram_command            := _LoadAddr64K ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

  return sram_return_value

'/////////////////////////////////////////////////////////////////////////////

PUB SRAM_LoadAddr512K_A(addr19)
' this function sets the SRAM controllers address latch to the sent 19-bit address, loads the lower 16-bits directly then walks/advances to the final
' 19-bit address if needs
'
' Parameters: addr19 - 19-bit address to set the SRAM address latch to

  sram_parameter_list_ptr := @addr19
  sram_command            := _LoadAddr512K ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

  return sram_return_value

'/////////////////////////////////////////////////////////////////////////////

PUB SRAM_LoadAddrLow_A(addr8)
' this function sets the SRAM controllers lower 8-bits of address latch to the sent 8-bit address
'
' Parameters: addr8 - 8-bit address to set lower 8-bits of SRAM address latch to

  sram_parameter_list_ptr := @addr8
  sram_command            := _LoadAddrLow ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

  return sram_return_value

'/////////////////////////////////////////////////////////////////////////////

PUB SRAM_LoadAddrHi_A(addr8)
' this function sets the SRAM controllers upper 8-bits of the 16-bit address latch to the sent 8-bit address, the upper 3-bit address
' which is not addressable directly is reset to 000 during this operation
'
' Parameters: addr8 - 8-bit address to set upper 8-bits (15..8) of SRAM address latch to

  sram_parameter_list_ptr := @addr8
```

```
  sram_command            := _LoadAddrHi ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

  return sram_return_value

' //////////////////////////////////////////////////////////////////////////

PUB SRAM_MemSet_A(addr19, data8, num_bytes)
' this function sets a contiguous block of SRAM to a byte value anywhere in the 512K and of any length
'
' Parameters: addr19    - 19-bit address to start memory set at
'             data8     - 8-bit data to write
'             num_bytes - number of bytes to write

  sram_parameter_list_ptr := @addr19
  sram_command            := _MemSet ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

' //////////////////////////////////////////////////////////////////////////

PUB SRAM_MemSum_A(addr19, num_bytes)
' this function sums a contiguous block of SRAM and returns the 32-bit result
'
' Parameters: addr19    - 19-bit address to start memory sum at
'             num_bytes - number of bytes to sum

  sram_parameter_list_ptr := @addr19
  sram_command            := _MemSum ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

  return sram_return_value

' //////////////////////////////////////////////////////////////////////////

PUB SRAM_MemCopy_A( dest_addr19, src_addr19,  num_bytes)
' this function copies a number of bytes from one address in the SRAM to another (non-overlapping)
' NOTE: this function uses a "cache" to speed the operation, the cache is located in COG ram, and thus has size limits due
' to the code space, currently the cache is located at "sram_cache" and has a size of SRAM_CACHE_PAGE_SIZE
'
' Parameters: dest_addr19 - 19-bit destination address in SRAM
'             src_addr19  - 19-bit source address in SRAM
'             num_bytes   - number of bytes to copy

  sram_parameter_list_ptr := @dest_addr19
  sram_command            := _MemCopy ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

' //////////////////////////////////////////////////////////////////////////

PUB MM_Copyto_SRAM_A( dest_addr19, src_addr16,  num_bytes)
' this function copies bytes anywhere in the propellers 64K main memory into the SRAMs 512K memory
'
' Parameters: dest_addr19 - 19-bit destination address in SRAM
'             src_addr16  - 16-bit source address in propeller main memory
'             num_bytes   - number of bytes to copy

  sram_parameter_list_ptr := @dest_addr19
  sram_command            := _MM_Copyto_SRAM ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

' //////////////////////////////////////////////////////////////////////////

PUB SRAM_Copyto_MM_A( dest_addr16, src_addr19,  num_bytes)
' this function copies bytes from the 512K SRAM to anywhere in the propellers 64K main memory
'
' Parameters: dest_addr16 - 16-bit destination address in propeller main memory
'             src_addr19  - 19-bit source address in SRAM
'             num_bytes   - number of bytes to copy

  sram_parameter_list_ptr := @dest_addr16
  sram_command            := _SRAM_Copyto_MM ' always set command last, so command doesn't start before parameter addresses are in

  repeat while sram_command

'//////////////////////////////////////////////////////////////////////////

CON

SRAM_DBUS_BIT_SHIFT     = 16
SRAM_CTRL_BIT_SHIFT     = 1
SRAM_STROBE_BIT_SHIFT   = 30

' commands for controller, pre-shifted to left 1-bit, so we don't have to do it during runtime, small enough to fit in constants
SRAM_CMD_WRITE_SHIFTED   = %00000_00_0
SRAM_CMD_READ_SHIFTED    = %00000_01_0
SRAM_CMD_LOADLO_SHIFTED  = %00000_10_0
SRAM_CMD_LOADHI_SHIFTED  = %00000_11_0

' SRAM controller commands (functions)
_Wait           = 0 ' do nothing command

_Write64K       = 1 ' write byte to lower 64K fast mode
_Read64K        = 2 ' read byte from lower 64K fast mode

_Write512K      = 3 ' write byte anywhere in 512K memory, slower (not implemented)
_Read512K       = 4 ' read byte from anywhere in memory, slower  (not implemented)

_WriteAuto      = 5 ' writes to the current address sram controller is set to, then auto inc/dec executes if programmed
_ReadAuto       = 6 ' writes to the current address sram controller is set to, then auto inc/dec executes if programmed

_LoadAddr64K    = 7 ' loads a 16-bit address (0..64K-1) directly into the low and high address latches, clears the upper 3-bits of address
_LoadAddr512K   = 8 ' loads a 19-bit address (0..512K-1) into address buffer, by advancing if necessary using dummy reads

_LoadAddrLow    = 9 ' loads only the lower 8-bits of address into address buffer
_LoadAddrHi     = 10 ' load only the uppper 8-bits of address into address buffer, also clears upper 3-bits, good to select 256 bytes "pages"

_MemSet         = 11 ' fills memory anywhere in the 512K region with a byte value
_MemCopy        = 12 ' copies a number of bytes in the SRAM from source to destination, doesn't support overlapping copies
```

```
_MM_Copyto_SRAM = 13 ' copies bytes from the Propeller's main memory to the SRAM's 512K space
_SRAM_Copyto_MM = 14 ' copies bytes from the SRAMs 512K to the Propeller's main memory

_ReadAddr       = 15 ' returns the current value of the 19-bit address buffer in the SRAM controller (not implemented)

_MemSum         = 16 ' sum a region of memory and returns the 32-bit result, helps with diagnostics and DSP stuff

' SRAM cache constants
SRAM_CACHE_PAGE_SIZE = 192 ' number of bytes in local COG memory cache page

'/////////////////////////////////////////////////////////////////////////////

DAT
              org $000                                  ' set the code emission for COG at $000

SRAM_Driver_Entry
              ' this is the entry point for the ASM SRAM driver, the driver is meant as an example to be used from other
              ' programs, but is by no means the most optimized it could be. moreover, to get optimal performance from the sram you
              ' must blend the sram access code right inline with your COG code since the difference between maximum speed is a few instructions
              ' any overhead can slow the system down considerable. the sram can operation faster than the prop can push it, so all bottlenecks
              ' are in the prop access in most cases. the sequence of steps to access the sram are basically bit banging the control lines and
              ' writing and reading the 8-bit bus in parallel

              ' the driver is commanded by means of a global shared memory region where the caller/client passes commands into, and the
              ' driver is "listening" for a new command, when a command is detected, the its executed and the driver resets the global command
              ' inidicating that the command is complete, this way the client doens't try to send commands until the current command is processed

              ' additionally the communication parameter passing area consists of 3 vars that must be declared by the client in the following order
              ' and format:
              '
              ' sram parameter passing area, this starting address always holds, the command, parameter ptr, and return value, in that order
              '
              ' long sram_command             ' holds command to SRAM driver, also the starting address of this LONG is assumed
                                                ' to be the start of all parms

              ' long sram_parameter_list_ptr  ' holds pointer to parameter list to driver

              ' long sram_return_value        ' holds result of sub-function (if there is one)

              ' so the client on start up passes the addressed of @sram_command from his globals, and the driver records this, then the client
              ' must place the command into sram_command, and set sram_parameter_list_ptr to point to the parameters for the function in question,
              ' in many cases, this might be the first parm on the stack or some other location, by using this extra layer of indirection the SRAM
driver
              ' has the most flexibility
              ' finally, if there are any results or return valuse they will show up in sram_return_value

              ' the driver also makes use of a COG memory cache to speed up SRAM <-> SRAM copies, performance degrades considerably when the source
              ' and target addresses are over the 64K directly addressable region of the SRAM, thus by at least caching blocks we can minimize how
              ' many seeks need to be made, the cache is located in the DAT section at the end of the program, make it whatever you wish that will
fit:
              ' sram_cache - cache
              ' its size must be set in BYTES in this variable located in the CON section:
              ' SRAM_CACHE_PAGE_SIZE = 192
              '
              ' which leave about 40 longs left for more code space!! So if you want to increase the cache size you will have to comment out some of
the
              ' functions

              mov sram_parms_base_ptr, par              ' copy boot parameter into sram_parms_base_ptr
              rdlong sram_cmd, sram_parms_base_ptr      ' read the SRAM initialization command from client, store in cmd
                                                        ' this is only done the 1st time, after this the main command waiting loop is entered
              mov sram_cmd_parms_ptr_ptr, par
              add sram_cmd_parms_ptr_ptr, #4            ' set pointer to pointer where parms are, this ptr must be dereferenced each function call

              mov sram_result_ptr, par
              add sram_result_ptr, #8                   ' store pointer to return variable global that driver uses

              ' set the IO direction and states for SRAM interface
              mov outa, SRAM_CTRL_MASK                  ' write 1's to interface, control lines only, so we don't accidentally fire a program clock

              ' one by one set pin groups to output
              or dira, SRAM_DBUS_CTRL_STROBE_MASK       ' set I/Os for sram interface all to outputs for now

              and outa, nSRAM_CTRL_MASK                 ' set control lines to code "00" which means next clock strobe read program


              ' initialize the memory controller, need to put the command on the data bus, and pulse strobe
              and outa, nSRAM_DBUS_MASK                 ' outa = (outa & !sram_dbus_mask), make hole for data

              mov r1, sram_cmd                          ' r1 = sram_cmd, which during startup is the program code for sram controller
              shl r1, #SRAM_DBUS_BIT_SHIFT              ' r1 = sram_cmd < 16, place data into proper position
              or outa, r1                               ' finally, outa = (outa & !sram_dbus_mask) | sram_cmd

              ' now pulse the strobe line
              or outa, SRAM_STROBE_MASK                 ' strobe = 1
              and outa, nSRAM_STROBE_MASK               ' strobe = 0

              ' now that we are done, set DBUS to inputs, leave control to outputs.
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' at this point, we have the following status
              ' MM[ sram_cmd ] -> current command from client
              ' MM [sram_cmd_parms_ptr, sram_cmd_parms_ptr+1, sram_cmd_parms_ptr+2,...,sram_cmd_parms_ptr+n ] -> sram parameters in MM

              ' write 0 out to sram command to signify first command was processed
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

'/////////////////////////////////////////////////////////////////////////////
' MAIN COMMAND WAITING LOOP
'/////////////////////////////////////////////////////////////////////////////

SRAM_Cmd_Wait_Loop
             ' enter into command loop waiting for command
              rdlong sram_cmd, sram_parms_base_ptr      wz                      ' read command from MM in global shared variable
       if_z   jmp #SRAM_Cmd_Wait_Loop                                         ' if non-zero then execute command, else continue to loop

              ' retreive latest pointer to parameters
              rdlong sram_cmd_parms_ptr, sram_cmd_parms_ptr_ptr ' sram_cmd_parms_ptr -> parameter list starting address for 0...(n-1) parameters

              ' ok now we basically need to do case (sram_cmd) and for each value execute the code body
```

```
              mov r0, #SRAM_Jump_Table          ' r0 = base address of jump table
              add r0, sram_cmd                  ' r0 = r0 + cmd
              movs :Read_Jumpvec, r0            ' access vector address in jump table at [r0 + cmd] -> destination of jmp (self modify code)
              nop       ' wait a second for pre-fetch, let self modifying code complete downstream

:Read_Jumpvec mov r1, 0 ' dummy 0 value has been overwritten with jump vector above
              jmp r1    ' jump to sub-function starting address

' this is an inline jump table, more or less implements an assembly language "case" statement
SRAM_Jump_Table  ' to save memory convert to words or bytes later, but means more code above to perform select logic
                 ' table holds starting address of each sub-function

              long Wait_           '= 0 , do nothing command (DONE)

              long Write64K_       '= 1 , write byte to lower 64K fast mode (DONE)
              long Read64K_        '= 2 , read byte from lower 64K fast mode (DONE)

              long Write512K_      '= 3 , write byte anywhere in 512K memory, slower (not implemented),
                                   '      instead load the address with LoadAddr512K_ then use ReadAuto/WriteAuto

              long Read512K_       '= 4 , read byte from anywhere in memory, slower  (not implemented)

              long WriteAuto_      '= 5 , writes to the current address sram controller is set to, then auto inc/dec executes if programmed (DONE)
              long ReadAuto_       '= 6 , writes to the current address sram controller is set to, then auto inc/dec executes if programmed (DONE)

              long LoadAddr64K_    '= 7 , loads a 16-bit address (0..64K-1) directly into the low and high address latches,
                                   '      also clears the upper 3-bits of address (DONE)

              long LoadAddr512K_   '= 8 , loads a 19-bit address (0..512K-1) into address buffer, by advancing if necessary using dummy reads (DONE)

              long LoadAddrLow_    '= 9 , loads only the lower 8-bits of address into address buffer (DONE)
              long LoadAddrHi_     '= 10, load only the uppper 8-bits of address into address buffer,
                                   '      also clears upper 3-bits, good to select 256 bytes "pages" (DONE)

              long MemSet_         '= 11 , fills memory anywhere in the 512K region with a byte value (DONE)
              long MemCopy_        '= 12 , copies a number of bytes in the SRAM from source to destination, doesn't support overlapping copies (DONE)

              long MM_Copyto_SRAM_ '= 13 , copies bytes from the Propeller's main memory to the SRAM's 512K space (DONE)
              long SRAM_Copyto_MM_ '= 14 , copies bytes from the SRAMs 512K to the Propeller's main memory (DONE)

              long ReadAddr_       '= 15 , returns the current value of the 19-bit address buffer in the SRAM controller (not implemented)

              long MemSum_         ' = 16 ' sum a region of memory and returns the 32-bit result, helps with diagnostics and DSP stuff

'//////////////////////////////////////////////////////////////////////////
' COMMAND SUB-FUNCTION IMPLEMENTATIONS
'//////////////////////////////////////////////////////////////////////////
Wait_         ' = 0 ' do nothing command, should never get here

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////
Write64K_     ' = 1 ' write byte to lower 64K fast mode. Eg. Write64K(address16, data8)
              ' this sub-function writes a byte to the lower 64K of memory, both the low and high address latches are written to with the sent 64K
              ' address, then the data byte is written.
              ' parameters: two longs, starting address: sram_cmd_parms_ptr
              ' parm 0 (32-bit): address to write, 16-bits
              ' parm 1 (32-bit): data to write, 8-bits

              ' retrieve long holding 16-bit address
              rdlong sram_parm0, sram_cmd_parms_ptr
              mov r0, sram_cmd_parms_ptr
              add r0, #4

              ' retrieve long holding 8-bit data
              rdlong sram_parm1, r0

              ' call set address routing, exprext sram_parm0 = 16-bit address
              mov r7, sram_parm0
              call #SetAddr64K_Proc

              ' place 8-bit data on data bus ------------------------------------------------
              mov r0, sram_parm1
              and r0, #$FF                      ' mask lower 8-bits (precaution)
              shl r0, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

              and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r0                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' command lines should already be in output mode, so only need to write 2-bit command code for write memory
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_WRITE_SHIFTED   ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_WRITE_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK                 ' strobe = 1
              and outa, nSRAM_STROBE_MASK               ' strobe = 0

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////
Read64K_      ' = 2 ' read byte from lower 64K fast mode, Eg. Read64K(address16)
              ' this sub-function reades a byte from the lower 64K of memory, both the low and high address latches are written to with the sent 64K
              ' address, then the data byte is retrieved and stored in sram_result_ptr
              ' parameters: one long, starting address: sram_cmd_parms_ptr
              ' parm 0 (32-bit): address to read from, 16-bits

              ' retrieve long holding 16-bit address
              rdlong sram_parm0, sram_cmd_parms_ptr

              ' call set address routing, expect sram_parm0 = 16-bit address
              mov r7, sram_parm0
              call #SetAddr64K_Proc

              ' place data bus into read mode and retrieve 8-bit data ----------------------
              and dira, nSRAM_DBUS_MASK
```

```
              ' command lines should already be in output mode, so only need to write 2-bit command code for read memory
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_READ_SHIFTED   ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_READ_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK                 ' strobe = 1

              ' data is now on bus, retrieve it...
              mov r0, ina                       ' pull data from external pins
              shr r0, #SRAM_DBUS_BIT_SHIFT      ' shift the data 16 time to the right [23..16] is location of data pins
              and r0, #$FF                      ' mask the data to 8-bits

              ' write data back out to global client parameter passing area
              wrlong r0, sram_result_ptr

              ' finally finish the clocking of the read
              and outa, nSRAM_STROBE_MASK               ' strobe = 0

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////////
Write512K_    ' = 3 ' write byte anywhere in 512K memory, slower

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////////
Read512K_     ' = 4 ' read byte from anywhere in memory, slower

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////////
WriteAuto_    ' = 5 ' writes to the current address sram controller is set to, then auto inc/dec executes if programmed
              ' this sub-function writes a byte to the currently addressed memory location in the 512K memory
              ' parameters: one longs, starting address: sram_cmd_parms_ptr
              ' parm 0 (32-bit): data to write, 8-bits

              ' retrieve long holding 8-bit data
              rdlong sram_parm0, sram_cmd_parms_ptr

              ' now place data bus into output mode
              or dira, SRAM_DBUS_MASK

              ' place 8-bit data on data bus ------------------------------------------------
              mov r0, sram_parm0
              and r0, #$FF                      ' mask lower 8-bits (precaution)
              shl r0, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

              and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r0                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' command lines should already be in output mode, so only need to write 2-bit command code for write memory
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_WRITE_SHIFTED  ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_WRITE_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK         ' strobe = 1
              and outa, nSRAM_STROBE_MASK       ' strobe = 0

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////////
ReadAuto_    ' = 6 ' reads the current byte addressed by the sram controller, then auto inc/dec executes if programmed
             ' this sub-function reads a byte from the currently addressed memory location in the 512K memory
             ' parameters: none

             ' place data bus into read mode and retrieve 8-bit data ----------------------
              and dira, nSRAM_DBUS_MASK

              ' command lines should already be in output mode, so only need to write 2-bit command code for read memory
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_READ_SHIFTED   ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_READ_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK                 ' strobe = 1

              ' data is now on bus, retrieve it...
              mov r0, ina                       ' pull data from external pins
              shr r0, #SRAM_DBUS_BIT_SHIFT      ' shift the data 16 time to the right [23..16] is location of data pins
              and r0, #$FF                      ' mask the data to 8-bits

              ' write data back out to global client parameter passing area
              wrlong r0, sram_result_ptr

              ' finally finish the clocking of the read
              and outa, nSRAM_STROBE_MASK               ' strobe = 0

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done
```

```
 '////////////////////////////////////////////////////////////////////////////
LoadAddr64K_   ' = 7 ' loads a 16-bit address (0..64K-1) directly into the low and high address latches, clears the upper 3-bits of address

               ' this sub-function sets the SRAMs' 16-bit latch, low and high address latches are written to with the sent 64K
               ' parameters: one long, starting address: sram_cmd_parms_ptr
               ' parm 0 (32-bit): address to set latches to (lower 16-bit used)

                ' retrieve long holding 16-bit address
               rdlong sram_parm0, sram_cmd_parms_ptr

               ' call set address routing, exprext sram_parm0 = 16-bit address
               mov r7, sram_parm0
               call #SetAddr64K_Proc

               ' reset data bus to input before leaving
               mov outa, #0
               and dira, nSRAM_DBUS_MASK

               ' command complete reset global, so caller/client can issue another command
               mov r0, #0
               wrlong r0, sram_parms_base_ptr

               jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'////////////////////////////////////////////////////////////////////////////
LoadAddr512K_  ' = 8 ' loads a 19-bit address (0..512K-1) into address buffer, by advancing if necessary using dummy reads

               ' this sub-function simple sets the addressing latch to the sent location in the 512K memory
               ' after which a readauto or writeauto would normally be performed
               ' if the address is <= 64K then the address is directly latched into the 16-bit address latc
               ' if the address is > 64K the $FFFF is latched into the address latch, then the memory controller is "walked" with dummy reads
               ' until its to the write location
               ' NOTE: this function only works with SRAM controller pre-programmed with post increment on reads.
               ' parameters: two longs, starting address: sram_cmd_parms_ptr
               ' parm 0 (32-bit): address to write, 19-bits used

               ' retrieve long holding 19-bit address, store in sram_parm0
               rdlong sram_parm0, sram_cmd_parms_ptr
               mov r0, sram_cmd_parms_ptr             ' advance pointer to next parameters which holds data
               add r0, #4

              ' retrieve long holding 8-bit data store in sram_parm1
               rdlong sram_parm1, r0

               mov r7, sram_parm0
               call #SetAddr512K_Proc

               ' reset data bus to input before leaving
               mov outa, #0
               and dira, nSRAM_DBUS_MASK

               ' command complete reset global, so caller/client can issue another command
               mov r0, #0
               wrlong r0, sram_parms_base_ptr

               jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'////////////////////////////////////////////////////////////////////////////
LoadAddrLow_   ' = 9 ' loads only the lower 8-bits of address into address buffer

               ' this sub-function sets the SRAMs' lower 8 bits of 16-bit address latch, more or less updating the 0-255 address, but leaving the
upper 8-bits
               ' of the latch as well as the internal upper 3-bits which are not accessible unless under controller control
               ' parameters: one long, starting address: sram_cmd_parms_ptr
               ' parm 0 (32-bit): address to set latches to (lower 8-bit used)

                ' retrieve long holding low 8-bits of 16-bit address
               rdlong sram_parm0, sram_cmd_parms_ptr

               ' now place data bus into output mode
               or dira, SRAM_DBUS_MASK

               ' place lower 8-bits of address on data bus ------------------------
               mov r0, sram_parm0
               and r0, #$FF                      ' mask lower 8-bits
               shl r0, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

               and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
               or  outa, r0                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

               ' command lines should already be in output mode, so only need to write 2-bit command code for load low address
               and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
               or outa, #SRAM_CMD_LOADLO_SHIFTED ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_LOADLO_SHIFTED)

               ' finally clock the strobe line and tell the sram controller to complete the operation
               or outa, SRAM_STROBE_MASK         ' strobe = 1
               and outa, nSRAM_STROBE_MASK       ' strobe = 0

               ' reset data bus to input before leaving
               mov outa, #0
               and dira, nSRAM_DBUS_MASK

               ' command complete reset global, so caller/client can issue another command
               mov r0, #0
               wrlong r0, sram_parms_base_ptr

               jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'////////////////////////////////////////////////////////////////////////////
LoadAddrHi_    ' = 10 ' load only the uppper 8-bits of address into address buffer, also clears upper 3-bits, good to select 256 bytes "pages"

               ' this sub-function sets the SRAMs' upper 8 bits of 16-bit address latch (19-total bits, 16 addressable)
               ' more or less updating the 0-255 page address, but leaving the lower 8-bits
               ' of the latch alone. Also,the internal upper 3-bits which are not accessible unless under controller control are reset to 000
               ' parameters: one long, starting address: sram_cmd_parms_ptr
               ' parm 0 (32-bit): address to set latches to (upper 8-bit used)

                ' retrieve long holding upper 8-bits of 16-bit address
               rdlong sram_parm0, sram_cmd_parms_ptr

               ' now place data bus into output mode
               or dira, SRAM_DBUS_MASK
```

```
              ' place upper 8-bits of 16-bit address onto data bus -------------------------
              mov r0, sram_parm0
              shr r0, #8                        ' move upper 8-bits into lower 8-bits
              and r0, #$FF                      ' mask lower 8-bits (precaution)
              shl r0, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

              and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r0                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' command lines should already be in output mode, so only need to write 2-bit command code for load high address
              and outa, nSRAM_CTRL_MASK          ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_LOADHI_SHIFTED  ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_LOADHI_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK                 ' strobe = 1
              and outa, nSRAM_STROBE_MASK               ' strobe = 0

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////
MemSet_       ' = 11 ' fills memory anywhere in the 512K region with a byte value
              ' parameters: three longs, starting address: sram_cmd_parms_ptr
              ' parm 0 (32-bit): destination address, 19-bits used
              ' parm 1 (32-bit): data to write, 8-bits used
              ' parm 2 (32-bit): number of bytes to fill/set  32-bit (only makes sense to move 512K at a time.

              ' retrieve long holding 19-bit address, store in sram_parm0
              rdlong sram_parm0, sram_cmd_parms_ptr
              mov r0, sram_cmd_parms_ptr             ' advance pointer to next parameters which holds data
              add r0, #4

             ' retrieve long holding 8-bit data store in sram_parm1
              rdlong sram_parm1, r0
              add r0, #4

              ' retrieve long holding number of bytes (32-bit value)
              rdlong sram_parm2, r0

              ' advance memory pointer to starting address
              mov r7, sram_parm0
              call #SetAddr512K_Proc

              ' now place data bus into output mode
              or dira, SRAM_DBUS_MASK

              ' place 8-bit data on data bus -----------------------------------------------
              mov r0, sram_parm1
              and r0, #$FF                      ' mask lower 8-bits (precaution)
              shl r0, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

              and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r0                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' command lines should already be in output mode, so only need to write 2-bit command code for write memory
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_WRITE_SHIFTED  ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_WRITE_SHIFTED)

              ' now we are ready to fill/set the memory, simply toggle the clock line (assumes auto increment on write)
:AddrAdvance
              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK                 ' strobe = 1
              and outa, nSRAM_STROBE_MASK               ' strobe = 0
              djnz sram_parm2, #:AddrAdvance            ' repeat while sram_parm2 (num_bytes) > 0

:AddrAdvanceEnd

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr


              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////
MemCopy_      ' = 12 ' copies a number of bytes in the SRAM from source to destination, doesn't support overlapping copies, will do it, but results
              ' won't be as expected.
              ' NOTE: Function operates by reading blocks of memory at a time and moving it, the blocks are stored locallly in a "sram cache", this
cache
              ' can be any size, suggested 64-256 bytes for any kind of performance
              ' parameters: three longs, starting address: sram_cmd_parms_ptr
              ' parm 0 (32-bit): destination address, 19-bits used
              ' parm 1 (32-bit): source address, 19-bits used
              ' parm 2 (32-bit): number of bytes to copy 32-bit (only makes sense to move 512K at a time.

              ' first compute number of pages and bytes that need copying
              ' copy pages, one at a time from SRAM to cache back to SRAM in cache size blocks 64-256 bytes at a time
              ' then copy remaining bytes

              ' READ PARAMETERS FROM CALLER ------------------------------------------------------------------------------
              rdlong sram_parm0, sram_cmd_parms_ptr  ' sram_parm0 hold destination address 19-bit
              mov r0, sram_cmd_parms_ptr             ' advance pointer to next parameters which holds data
              add r0, #4

             ' retrieve long holding 19-bit source address store in sram_parm1
              rdlong sram_parm1, r0
              add r0, #4

              ' retrieve long holding number of bytes (32-bit value) to copy
              rdlong sram_parm2, r0

               ' parameters now retrieved, at this point we have
```

```
              ' sram_parm0 = dest_addr
              ' sram_parm1 = src_addr
              ' sram_parm2 = num_bytes_to_copy

              ' MAIN MEMCPY LOOP ------------------------------------------------------------------------------------

              ' we need to copy a number of bytes from SRAM to SRAM, too slow to copy if copying splits between 64K mark, thus
              ' use internal COG memory as a cache and copy "pages" from 64-256 bytes at a time from SRAM -> COG Cache -> SRAM

              mov num_bytes_to_copy, sram_parm2         ' this is the total number of bytes requested to be copied from client



:CopyLoop ' ------------------------------------------------------------------------------------------------------------------ LOOP
              cmp num_bytes_to_copy, #SRAM_CACHE_PAGE_SIZE    wc, wz      ' copy one page at a time, whatever memory permits
        if_ae jmp #:CopyMorePages               ' if num_bytes_to_copy >= SRAM_CACHE_PAGE_SIZE then..
              'else num_bytes_to_copy < SRAM_CACHE_PAGE_SIZE then..

              mov num_copy_bytes, num_bytes_to_copy     ' set number of bytes to copy this pass
              mov num_bytes_to_copy, #0                 ' and we are done next time around

              jmp #:SetSrcAddr
              ' end else

:CopyMorePages ' if num_bytes_to_copy >= SRAM_CACHE_PAGE_SIZE then..
              mov num_copy_bytes, #SRAM_CACHE_PAGE_SIZE       ' lets copy the maximum page size this pass
              sub num_bytes_to_copy, #SRAM_CACHE_PAGE_SIZE    ' adjust the bytes left to copy next pass, when num_bytes_to_copy == 0, we are done!
              ' end if



              ' COPY NEXT SRAM PAGE (OR LESS) INTO COG CACHE--------------------------------------------------------------
:SetSrcAddr
              ' starting address of local cog SRAM cache
              mov sram_cache_entry_ptr, #sram_cache

              ' set source address in memory latch for reads
              mov r7, sram_parm1                ' r7 = source address
              call #SetAddr512K_Proc


             ' place data bus into read mode and retrieve 8-bit data ----------------------------------------------------
              and dira, nSRAM_DBUS_MASK

              ' command lines should already be in output mode, so only need to write 2-bit command code for read memory
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_READ_SHIFTED   ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_READ_SHIFTED)

              ' outer most "page size" loop, this will either be <= SRAM_CACHE_PAGE_SIZE, we need to copy this many bytes, but we need to perform
              ' the copies in groups of 4, since the COG memory is LONG based and the SRAM is BYTE based, thus this disparity causes
              ' a second interior loop or step to read the 4 bytes out at a time, position them in a long then write the long to the cache


              mov curr_byte, #0                 ' current byte loop counter, need to copy a total of num_copy_bytes from SRAM to cache

:ByteCacheReadLoop ' ------------------------------------------------------------------------------------------------------ LOOP

              ' read bytes 0,1,2,3 from SRAM sequentially and store into cache_entry which will then be written to cache
              ' (note if only 1-3 bytes need to be written then this is redundant, but mork worth to conditionally test it
              mov cache_byte, #4                ' cache byte index in packed long, used as counter here
              mov num_shifts, #0                ' number of shifts per byte copy iteration, shifts byte read into packed position in long
              mov cache_entry, #0               ' used to hold packed LONG constructed via 4 bytes read from SRAM
                                                ' then this is written at once to local COG cache

:ReadBytesIntoCacheEntryLoop ' -------------------------------------------------------------------------------------------- LOOP

              ' clock the strobe line and tell the sram controller to initiate the read and auto inc/dec
              or outa, SRAM_STROBE_MASK         ' strobe = 1

              ' data is now on bus, retrieve it...
              mov r0, ina                       ' pull data from external pins
              shr r0, #SRAM_DBUS_BIT_SHIFT      ' shift the data 16 time to the right [23..16] is location of data pins
              and r0, #$FF                      ' mask the data to 8-bits
              shl r0, num_shifts                ' r0 = r0 << num_shifts, used to place bits into correct LONG byte position
              or cache_entry, r0                ' store byte 0 into cache entry
              add num_shifts, #8                ' num_shifts = num_shifts + 8, next byte read needs to be shifted 8 more bits

              ' finally finish the clocking of the read
              and outa, nSRAM_STROBE_MASK       ' strobe = 0

              djnz cache_byte, #:ReadBytesIntoCacheEntryLoop        ' while cache_byte > 0, loop


              ' we have the next packed LONG of bytes from the SRAM read in, now its time to store them in the cache and advance
              ' cache pointer..
              ' COGMEM[ sram_cache_entry_ptr++ ]  = cache_entry
              movd :WriteCache, sram_cache_entry_ptr  ' modify destination address, so we can write to local COG memory with packed cache entry
              add sram_cache_entry_ptr, #1            ' increment cache pointer for next iteration
                                                      ' (also give pre-fetch time to complete modify downstream)

:WriteCache   mov 0, cache_entry                      ' self modifying code, dummy "0" destination modified above receives packed 4-byte sram data


              add curr_byte, #4               wc, wz  ' bytes always copied 4 at a time, so at worst case an extra 3 bytes will be copied into cache
                                                      ' but has no effect on final write operation, which will always copy out EXACTLY the
                                                      ' correct number of bytes

              cmp curr_byte, num_copy_bytes   wc, wz
         if_b jmp #:ByteCacheReadLoop                 ' loop while current byte being copied < total number of bytes to copy


              ' WRITE CACHED COG PAGE BACK TO SRAM ------------------------------------------------------------------------

              ' at this point we have a cache page full of bytes (in long size chunks), could be a full page or less than full size
              ' if total number of bytes to copy was less than cache page size OR we are on the last copy and copying the remaining
              ' left over bytes from the last full page size copy

              ' advance address pointer to destination location in SRAM
              mov r7, sram_parm0                ' r7 = destination address
              call #SetAddr512K_Proc


              ' copy bytes from COG cache into sram now...
```

```
              ' perform what setup we can outside the loop


              ' place data bus into write mode and prepare to write cache page out to sram ----------------------
              or dira, SRAM_DBUS_MASK

              ' command lines should already be in output mode, so only need to write 2-bit command code for write memory
              and outa, nSRAM_CTRL_MASK          ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_WRITE_SHIFTED   ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_WRITE_SHIFTED)

              ' outer most "page size" loop, this will either be <= SRAM_CACHE_PAGE_SIZE, we need to copy this many bytes, but we need to perform
              ' the copies in groups of 4, since the COG memory is LONG based and the SRAM is BYTE based, thus this disparity causes
              ' some work to have to be done to extract the packed bytes from the long and then send them to sram

              mov curr_byte, #0                ' byte loop counter, need to copy num_copy_bytes from cache to SRAM completing the memory move
                                               ' (a page at a time)

:ByteCacheWriteLoop
              ' need to do this more or less
              ' SRAM [ curr_byte ] = (COG_CACHE [ curr_byte / 4 ] >> (curr_byte MOD 4) ) && $FF

              ' compute COG index in sram cache to read from (long data containing packed bytes)
              mov cache_index, curr_byte
              shr cache_index, #2              ' cache_index = curr_byte / 4

              ' compute the byte index in the long to read
              mov cache_byte, curr_byte
              and cache_byte, #3               ' cache_byte = curr_byte mod 4

              ' at this point we know the long in the cache memory and the byte within the long we need to read
              mov r0, #sram_cache
              add r0, cache_index

              movs :ReadCache, r0              ' want to read COGMEM_SRAM_CACHE [ cache_index ], modify source operand downstream
              nop                              ' put some work from downstream here to optimize, but leave for now to see what's going on

:ReadCache    mov cache_entry, 0               ' cache_entry = COGMEM_SRAM_CACHE [ cache_index ], dummy "0" is overwritten with actual index
                                               ' by self modify code

              ' now we simply need to extract the proper byte from the cache entry and write it to sram

              mov num_shifts, cache_byte
              shl num_shifts, #3               ' num_shifts = cache_byte * 8

              shr cache_entry, num_shifts      ' place the requested byte into the lower 8-bits position

              ' we have everything we need, now let's write the data
              mov r0, cache_entry
              and r0, #$FF                     ' mask lower 8-bits (precaution)
              shl r0, #SRAM_DBUS_BIT_SHIFT     ' shift data into position

              and outa, nSRAM_DBUS_MASK        ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r0                     ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK        ' strobe = 1
              and outa, nSRAM_STROBE_MASK      ' strobe = 0


              add curr_byte, #1                      ' if (++curr_byte < num_copy_bytes) then repeat loop
              cmp curr_byte, num_copy_bytes wc, wz
       if_b   jmp #:ByteCacheWriteLoop


              ' done with current cache page pass, may or may not have read/write an entire page, either way update source and destination pointers
              add sram_parm0, num_copy_bytes
              add sram_parm1, num_copy_bytes

              ' END OF MAIN COPY LOOP, TEST IF MORE BYTES NEED TO BE COPIED, AND LOOP BACK ----------------------------

              ' if num_bytes_to_copy > 0 then continue copying pages, jump back up to outer most loop
              cmp num_bytes_to_copy, #0 wc ,wz
       if_a   jmp #:CopyLoop


              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop          ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////////
MM_Copyto_SRAM_ ' = 13 ' copies bytes from the Propeller's main memory to the SRAM's 512K space
                ' NOTE: assumes that controller is in auto inc after write mode
                ' parameters: three longs, starting address: sram_cmd_parms_ptr
                ' parm 0 (32-bit): destination address in sram, 19-bits used
                ' parm 1 (32-bit): source address in main memory, 16-bits used (propeller only has 64K address space, lower 32K RAM, upper 32K ROM)
                ' parm 2 (32-bit): number of bytes to copy 32-bit (only makes sense copy up to 64K though, since that's how big the prop memory is

              ' READ PARAMETERS FROM CALLER ----------------------------------------------------------------------
              rdlong sram_parm0, sram_cmd_parms_ptr  ' sram_parm0 hold destination address 19-bit
              mov r0, sram_cmd_parms_ptr             ' advance pointer to next parameters which holds data
              add r0, #4

             ' retrieve long holding 16-bit source address referring to main memory store in sram_parm1
              rdlong sram_parm1, r0
              add r0, #4

              ' retrieve long holding number of bytes (32-bit value) to copy
              rdlong sram_parm2, r0


              ' parameters now retrieved, at this point we have
              ' sram_parm0 = dest_addr in SRAM
              ' sram_parm1 = src_addr in prop main memory (MM)
              ' sram_parm2 = num_bytes_to_copy to copy

              ' we want to perform the following algorithm in the abstract...
              'for (index = 0; index < num_bytes_to_copy; index++)
```

```
              '  SRAM[dest_addr + index] = MM[src_addr + index]

              ' this isn't too bad, since we are not copying sram to sram, we only have to place the sram at the destination address to write to

              ' advance SRAM address pointer to destination location in SRAM
              mov r7, sram_parm0                      ' r7 = destination address in SRAM
              call #SetAddr512K_Proc

              ' place data bus into write mode and prepare to write byte stream from main memory
              or dira, SRAM_DBUS_MASK

              ' command lines should already be in output mode, so only need to write 2-bit command code for write memory
              and outa, nSRAM_CTRL_MASK               ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_WRITE_SHIFTED        ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_WRITE_SHIFTED)

              '  ------------------------------------------------------------------------------------------------ LOOP
              ' at loop entry, sram_parm2 holds total number of bytes requested to be copied from client
:CopytoSramLoop

              rdbyte r0, sram_parm1                   ' r0 = MM [ src_addr ]
              ' we have everything we need, now let's write the data

              and r0, #$FF                    ' mask lower 8-bits (precaution)
              shl r0, #SRAM_DBUS_BIT_SHIFT    ' shift data into position

              and outa, nSRAM_DBUS_MASK       ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r0                    ' outa = (outa & !sram_dbus_mask) | (r0 << 16)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              ' SRAM [ dest_addr ] <- r0 <- MM [ src_addr]
              or outa, SRAM_STROBE_MASK       ' strobe = 1
              and outa, nSRAM_STROBE_MASK     ' strobe = 0

              ' SRAM address will auto increment, but have to increment source MM address manually
              add sram_parm1, #1

              djnz sram_parm2, #:CopytoSramLoop  ' repeat while sram_parm2 (number of bytes to copy ) > 0

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop         ' return to main command fetch loop when done

'////////////////////////////////////////////////////////////////////////////
SRAM_Copyto_MM_ ' = 14 ' copies bytes from the SRAMs 512K to the Propeller's main memory
                ' NOTE: assumes that controller is in auto inc after write mode
                ' parameters: three longs, starting address: sram_cmd_parms_ptr
                ' parm 0 (32-bit): source address in main memory, 16-bits used (propeller only has 64K address space, lower 32K RAM, upper 32K ROM)
                ' parm 1 (32-bit): destination address in sram, 19-bits used
                ' parm 2 (32-bit): number of bytes to copy 32-bit (only makes sense copy up to 64K though, since that's how big the prop memory is

              ' READ PARAMETERS FROM CALLER -------------------------------------------------------------------------
              rdlong sram_parm0, sram_cmd_parms_ptr  ' sram_parm0 hold destination address in main memory 16-bit used
              mov r0, sram_cmd_parms_ptr             ' advance pointer to next parameters which holds data
              add r0, #4

             ' retrieve long holding 19-bit source address referring to SRAM store in sram_parm1
              rdlong sram_parm1, r0
              add r0, #4

              ' retrieve long holding number of bytes (32-bit value) to copy
              rdlong sram_parm2, r0


              ' parameters now retrieved, at this point we have
              ' sram_parm0 = dest_addr in main memory (MM)
              ' sram_parm1 = src_addr in SRAM
              ' sram_parm2 = num_bytes_to_copy to copy

              ' we want to perform the following algorithm in the abstract...
              'for (index = 0; index < num_bytes_to_copy; index++)
              '  MM[dest_addr + index] = SRAM[src_addr + index]

              ' this isn't too bad, since we are not copying sram to sram, we only have to place the sram at the source location to read from

              ' advance SRAM address pointer to source location in SRAM
              mov r7, sram_parm1                      ' r7 = source address in SRAM
              call #SetAddr512K_Proc

              ' place data bus into read mode and prepare to read byte stream from SRAM
              and dira, nSRAM_DBUS_MASK

              ' command lines should already be in output mode, so only need to write 2-bit command code for read memory
              and outa, nSRAM_CTRL_MASK               ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_READ_SHIFTED         ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_READ_SHIFTED)

              '  ------------------------------------------------------------------------------------------------ LOOP
              ' at loop entry, sram_parm2 holds total number of bytes requested to be copied from client
:CopytoMainMemoryLoop

              rdbyte r0, sram_parm1                   ' r0 = MM [ src_addr ]
              ' we have everything we need, now let's write the data

              ' clock the strobe line and tell the sram controller to initiate the read and auto inc/dec
              or outa, SRAM_STROBE_MASK       ' strobe = 1

              ' data is now on bus, retrieve it...
              mov r0, ina                     ' pull data from external pins
              shr r0, #SRAM_DBUS_BIT_SHIFT    ' shift the data 16 time to the right [23..16] is location of data pins
              and r0, #$FF                    ' mask the data to 8-bits

              ' write the data to main memory
              wrbyte r0, sram_parm0

              ' finally finish the clocking of the read
              and outa, nSRAM_STROBE_MASK     ' strobe = 0

              ' SRAM address will auto increment, but have to increment source MM address manually
```

```
              add sram_parm0, #1

              djnz sram_parm2, #:CopytoMainMemoryLoop  ' repeat while sram_parm2 (number of bytes to copy ) > 0

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr


              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done

'//////////////////////////////////////////////////////////////////////////
ReadAddr_       ' = 15 ' returns the current value of the 19-bit address buffer in the SRAM controller


              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done


'//////////////////////////////////////////////////////////////////////////
MemSum_       ' = 16 ' sum a region of memory and returns the 32-bit result, helps with diagnostics and DSP stuff
              ' parameters: three longs, starting address: sram_cmd_parms_ptr
              ' parm 0 (32-bit): source address to sum, 19-bits used
              ' parm 1 (32-bit): number of bytes to sum  32-bit (only makes sense to move 512K at a time.

              ' retrieve long holding 19-bit address, store in sram_parm0
              rdlong sram_parm0, sram_cmd_parms_ptr
              mov r0, sram_cmd_parms_ptr             ' advance pointer to next parameters which holds data
              add r0, #4

              ' retrieve long holding number of bytes (32-bit value)
              rdlong sram_parm1, r0

              ' advance memory pointer to starting address
              mov r7, sram_parm0
              call #SetAddr512K_Proc

              ' place data bus into read mode and prepare to read byte stream from SRAM
              and dira, nSRAM_DBUS_MASK

              ' command lines should already be in output mode, so only need to write 2-bit command code for read memory
              and outa, nSRAM_CTRL_MASK              ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_READ_SHIFTED        ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_READ_SHIFTED)

              ' now we are ready to read/sum the memory, simply toggle the clock line (assumes auto increment on read)
              mov r1, #0                        ' r1 holds sum
:AddrAdvance
              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK              ' strobe = 1

              ' data is now on bus, retrieve it...
              mov r0, ina                       ' pull data from external pins
              shr r0, #SRAM_DBUS_BIT_SHIFT      ' shift the data 16 time to the right [23..16] is location of data pins
              and r0, #$FF                      ' mask the data to 8-bits
              add r1, r0                        ' sum += data

              ' finally finish the clocking of the read
              and outa, nSRAM_STROBE_MASK            ' strobe = 0

              djnz sram_parm1, #:AddrAdvance         ' repeat while sram_parm2 (num_bytes) > 0

:AddrAdvanceEnd

              ' write sum out to caller
              wrlong r1, sram_result_ptr

              ' reset data bus to input before leaving
              mov outa, #0
              and dira, nSRAM_DBUS_MASK

              ' command complete reset global, so caller/client can issue another command
              mov r0, #0
              wrlong r0, sram_parms_base_ptr

              jmp #SRAM_Cmd_Wait_Loop           ' return to main command fetch loop when done


'//////////////////////////////////////////////////////////////////////////
' END OF SRAM COMMANDS
'//////////////////////////////////////////////////////////////////////////

'//////////////////////////////////////////////////////////////////////////
' INTERNAL "HELPER" FUNCTIONS TO SAVE CODE SPACE
'//////////////////////////////////////////////////////////////////////////

SetAddr64K_Proc
              ' internal sub-routine that sets the 64K lower 16-bit latch address
              ' expects r7 holding 16-bit address to advance to

              ' now place data bus into output mode
              or dira, SRAM_DBUS_MASK

              ' place lower 8-bits of 16-bit address onto data bus ------------------------

              and r7, #$FF                      ' mask lower 8-bits
              shl r7, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

              and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r7                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' command lines should already be in output mode, so only need to write 2-bit command code for load low address
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_LOADLO_SHIFTED ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_LOADLO_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK         ' strobe = 1
              and outa, nSRAM_STROBE_MASK       ' strobe = 0

              ' place upper 8-bits of 16-bit address onto data bus ------------------------
              mov r7, sram_parm0
```

```
              shr r7, #8                        ' move upper 8-bits into lower 8-bits
              and r7, #$FF                      ' mask lower 8-bits (precaution)
              shl r7, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

              and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r7                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' command lines should already be in output mode, so only need to write 2-bit command code for load high address
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_LOADHI_SHIFTED ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_LOADHI_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK         ' strobe = 1
              and outa, nSRAM_STROBE_MASK       ' strobe = 0

SetAddr64K_Proc_Ret ret

'////////////////////////////////////////////////////////////////////////////

SetAddr512K_Proc
              ' internal sub-routine that sets the 512K 19-bit address latch completely
              ' expects r7 holding 19-bit address to advance to
              ' side effects destroys, r0, r6, r7

              ' compare desired target address to $FFFF, if less than simply latch address and write data, else
              ' write $FFFF and then advance to final location with dummy reads
              mov r6, r7                        ' make copy of address in r6

              cmp r7, MAX_SHORT          wz, wc
      if_be   jmp #:LoadAddr                    ' if addr <= $FFFF then load it into 16-bit latch, jump to latch code

              mov r7, MAX_SHORT                 ' else latch $FFFF then advance to write location in SRAM using post increment on read


:LoadAddr    ' we are at the target address either by a short 16-bit direct latch or by advancing to the location via dummy reads...
             ' either way, we can now write the data as usual
             ' now place data bus into output mode

'////////////////////////////////////////////////////////////////////////////////////////////////
' EXCISE this code later and merge into load address generic call, this is just a little faster ///////
'////////////////////////////////////////////////////////////////////////////////////////////////

              or dira, SRAM_DBUS_MASK

              ' place lower 8-bits of 16-bit address onto data bus -----------------------
              mov r0, r7
              and r0, #$FF                      ' mask lower 8-bits
              shl r0, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

              and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r0                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' command lines should already be in output mode, so only need to write 2-bit command code for load low address
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_LOADLO_SHIFTED ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_LOADLO_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK         ' strobe = 1
              and outa, nSRAM_STROBE_MASK       ' strobe = 0

              ' place upper 8-bits of 16-bit address onto data bus -------------------------
              mov r0, r7
              shr r0, #8                        ' move upper 8-bits into lower 8-bits
              and r0, #$FF                      ' mask lower 8-bits (precaution)
              shl r0, #SRAM_DBUS_BIT_SHIFT      ' shift data into position

              and outa, nSRAM_DBUS_MASK         ' outa = (outa & !sram_dbus_mask), make hole for data
              or  outa, r0                      ' outa = (outa & !sram_dbus_mask) | (sram_parm0 << 16)

              ' command lines should already be in output mode, so only need to write 2-bit command code for load high address
              and outa, nSRAM_CTRL_MASK          ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_LOADHI_SHIFTED  ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_LOADHI_SHIFTED)

              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK                ' strobe = 1
              and outa, nSRAM_STROBE_MASK              ' strobe = 0

'////////////////////////////////////////////////////////////////////////////////////////////////
'////////////////////////////////////////////////////////////////////////////////////////////////

              ' now advance memory, set up read address in 512K -----------------------------

              ' advance the address pointer by using the post increment behavior (if needed)
              sub r6, r7 wz, wc                 ' compute difference between addr19 and latched address
       if_z   jmp #:AddrAdvanceEnd              ' if (addr19 - $FFFF) > 0 then perform advance, else no advance needed, jump over code

              ' place data bus into read mode for dummy read, data is ignored
              and dira, nSRAM_DBUS_MASK

              ' command lines should already be in output mode, so only need to write 2-bit command code for read memory
              and outa, nSRAM_CTRL_MASK         ' clear control lines to "00", make hole for command
              or outa, #SRAM_CMD_READ_SHIFTED   ' outa = (outa & nSRAM_CTRL_MASK) | (SRAM_CMD_READ_SHIFTED)

              ' advance address pointer (addr19 - $FFFF) times, Eg. ($10000 - $FFFF) = 1, one clock then exit

:AddrAdvance
              ' finally clock the strobe line and tell the sram controller to complete the operation
              or outa, SRAM_STROBE_MASK                ' strobe = 1
              and outa, nSRAM_STROBE_MASK              ' strobe = 0
              djnz r6, #:AddrAdvance                   ' repeat while sram_parm0 (difference from target and latched)  > 0

:AddrAdvanceEnd

SetAddr512K_Proc_Ret ret

'////////////////////////////////////////////////////////////////////////////////////////////////

DAT

' general purpose registers
r0                      long      $0
r1                      long      $0
r2                      long      $0
```

```
r3                          long      $0
r4                          long      $0
r5                          long      $0
r6                          long      $0
r7                          long      $0

' 32-bit constants, masks, anything that is greater than 9-bits and can't be represented as an immediate
SRAM_CTRL_MASK              long      %0000_0000_0000_0000_0000_0000_0000_0110
SRAM_DBUS_MASK              long      %0000_0000_1111_1111_0000_0000_0000_0000
SRAM_STROBE_MASK            long      %0100_0000_0000_0000_0000_0000_0000_0000
SRAM_DBUS_CTRL_STROBE_MASK  long      %0100_0000_1111_1111_0000_0000_0000_0110

nSRAM_CTRL_MASK             long      %1111_1111_1111_1111_1111_1111_1111_1001
nSRAM_DBUS_MASK             long      %1111_1111_0000_0000_1111_1111_1111_1111
nSRAM_STROBE_MASK           long      %1011_1111_1111_1111_1111_1111_1111_1111
nSRAM_DBUS_CTRL_STROBE_MASK long      %1011_1111_0000_0000_1111_1111_1111_1001

' used for debugging
DEBUG_LED_MASK              long      %0000_0000_0000_0000_0000_0000_0000_0001
nDEBUG_LED_MASK             long      %1111_1111_1111_1111_1111_1111_1111_1110

' math constants
MAX_INT                     long      $FFFFFFFF               ' largest integer also -1 in 2's complement
MAX_SHORT                   long      $0000FFFF
ZERO                        long      $00000000

' sram interface variables and working variables
sram_parms_base_ptr         long      $0 ' pointer to main memory where the sram interface parameter passing area is
                                        ' 0 - sram command
                                        ' 1 - pointer to sram parameters from caller

sram_cmd                    long      $0 ' the requested sram controller command
sram_cmd_parms_ptr          long      $0 ' pointer to sram parameters
sram_cmd_parms_ptr_ptr      long      $0 ' pointer to pointer pointing at sram parameters
sram_result_ptr             long      $0 ' pointer to global used to hold result from driver

' local storage for all the sram parameters needed for function call
sram_parm0                  long      $0
sram_parm1                  long      $0
sram_parm2                  long      $0
sram_parm3                  long      $0

' this is the cache storage for sram to sram memory copy/moves, we need a temporary buffer to move the data (64-256 bytes is a good choice)
' resize depending on what code you include, can use RES to save typing, but used data statements to be able to initialize cache for different
reasons..
' if you need more cache size you can comment out functional chunks of the ASM driver code that you aren't using
sram_cache                  long 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ' 64 bytes per line of storage
                            long 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

' cache algorithm working vars
num_bytes_to_copy           long 0 ' total number of bytes to copy
num_copy_bytes              long 0 ' number of bytes to copy this pass, page 0 to SRAM_CACHE_PAGE_SIZE
curr_byte                   long 0 ' current byte being processed
num_shifts                  long 0 ' number of times to shift operand
sram_cache_entry_ptr        long 0 ' points to sram cache entry being processed
cache_entry                 long 0 ' a data entry from the cache
cache_byte                  long 0 ' a byte in cache entry
cache_index                 long 0 ' index into the cache
```

NOTES