Author: Adrian M Whatley

Date: 7 October 1997

File: scx1swod.doc

Revision: 6

Silicon Cortex Software Design

INTRODUCTION

This document describes the outline design of the software to control the Silicon Cortex board being developed under the leadership of Rodney Douglas of the Institute of Neuroinformatics in Zürich[1], and funded by the the US Office of Naval Research (ONR). The hardware design is being undertaken by Steve Deiss of Applied Neurodynamics, Encinitas, California. This document only deals with the software that runs on the silicon cortex board. It is intended to communicate the software design to involved parties, and as a means of obtaining criticism of the design from otherwise uninvolved software engineers.

REVISION HISTORY

Rev.1: 16/05/94	First partial draft for review. VME host interface, command processing, AE ISR and queue, AE mapping, domain bus SRAM filters, NVRAM management and boot loader modules not yet described.
Rev. 2: 16/05/94	The DAC is now a parallel device accessed via MUXB. AEBs are now polled rather than being read on interrupt. All but boot loader module now has some detail.
Rev. 3: 16/06/94	Changes arising from away day and new information on hardware from SD. New references added. All modules other than boot loader now fully described. This version for review at meeting at RM on 24/06/94.
Rev. 4: 27/06/94	Changes arising from design review. Also major change to SCX/host communication protocol description in VME host interface module section and addition of domain clock speed programming.
Rev. 5: 30/01/95	Final revision before handover.
Rev. 6: 06/10/97	Bring specification up to date with changes made to software, especially Appendix A, Booting.

1 Formerly of the Medical Research Council Anatomical Neuropharmacology Unit in Oxford.

CONTENTS

INTRODUCTION....1
REVISION HISTORY....1
CONTENTS....2
1 REFERENCES
....
2
13 OVERVIEW
....
3
14 Multi-neuron chips and address-events....3
15 Generalisation of address-event buses....4
16 Analog parameters and synaptic weights....4
17 Digital, latched parameters....4
18 SCX hardware features....4
19 DESIGN CONCEPT
....
5
20 Key functionality....5
21 Interrupts....6
22 Data flow diagram....6
23 Prototyping....8
24 Booting....8
25 Interrupts....8
26 Alternatives....9
27 Polled versus interrupt driven AE processing....9
28 IDENTIFIED COMPONENTS
....
11
29 Parameters and synaptic weights module....11
30 Timer module....14
31 VME host interface module....14
32 Command processing module....16
33 MUX FIFO module....16
34 AE mapping module....17
35 AE mapping, method 1....18
36 AE mapping, method 2....18
37 AEB FIFO module....20
38 Front panel LEDs and scope triggers module....21
39 Domain AE buses module....22
40 NVRAM management module....22
41 Boot loader....23
APPENDIX A - BOOTING
....
24
16-Bit Parallel EPROM Mode....24
16-Bit Parallel I/O Mode....24
Summary....26
GLOSSARY
....
27

1 REFERENCES

2 *An Analog VLSI System for Stereoscopic Vision,* Misha Mahowald, 1994, esp. Chapter 3, *The Silicon Optic Nerve*

3 *Event Address Bus,* Version 1.03, Tony Matthews, 19th October 1993

4 *Cortex Board 1 Documentation,* Steve Deiss, Rodney Douglas, Mike Fischer, Misha Mahowald, Tony Matthews, January 20, 1994

5 *Cortex Board Summary Specification*, Stephen R. Deiss, Applied Neurodynamics, March 12, 1994

6 *Description of the Silicon Cortex Board,* Steve Deiss, Applied Neurodynamics

7 *SCX-1 Programming Model,* Steve Deiss, ANdt, 082194, (3e)

8 *SCX-1 Registers & Ports,* Steve Deiss, ANdt, 082194, (3e)

9 *TMS320C5x User's Guide*, Texas Instruments, 2547301-9721 revision D, January 1993

10 *TMS320C5x Evaluation Module Technical Reference*, Texas Instruments, 2617584-9741 revision *, May 1992

11 *MULTIPLEXOR MAPPING FOR MNC SCX1*
[Misha Mahowald, 10/5/94]

12 *SN74ABT7820 512 x 18 x 2 FIRST-IN, FIRST OUT MEMORY* [data sheet], Texas Instruments, SCAS206A - D4503, August 1991 - Revised August 1992

12. *VMEbus A practical companion,* Steve Heath, Butterworth Heinemann, ISBN 0-7506-1750-0

13 OVERVIEW

14 Multi-neuron chips and address-events

The Silicon Cortex (SCX) board carries two multi-neuron chips, and up to four additional multi-neuron chips on a daughter board. Each of the first generation of multi-neuron chips (MNCs) is an array of 36 silicon neurons[2], each of which has in turn 18 synapses[3] built upon a matrix that may be addressed by neuron number and parameter number. Useful networks of neurons have a high degree of connectivity. The typical number of connections from a silicon neuron will be about 32, and the maximum will be 256. To allow maximum flexibility, and because of MNC pinout limitations, the connections between neurons are made in software rather than hardware. The activation of a neuron is represented externally to the MNC as an address-event (AE), i.e. by the placing of the address of the neuron onto a bus, the local address-event bus (LAEB). This bus is connected (indirectly) to the data bus of a Texas Instruments TMS320C50 DSP. It is the responsibility of the software running on this DSP to translate AEs into synaptic destination addresses, and re-transmit them to the MNCs on a separate bus called MUXB. This is a one to many translation.

15 Generalisation of address-event buses

There are three further AE sources on the SCX board in addition to LAEB. These are two domain address-event buses (DAEB0 & DAEB1) used for communicating with other SCX boards and a VME address-event bus (VAEB) typically used for low volume injection of events from a host system for test purposes, logging AEs or communicating with digital processing applications. (The SCX board is implemented as a VME card.)

Address-event encoded input data from for example a silicon retina may arrive on any of the AE buses including LAEB in the case of sensors attached via the daughter board connector.

2 Future versions may have up to 256 or even 1024 neurons.
3 Future versions may have fewer, or up to at least 256 synapses per neuron.

In general, any AE arriving on any of the AE buses (including LAEB) may require translation into synaptic destination addresses for the local MNCs and/or routing onto any or all of the other AE buses. In the case of the domain buses only, this routing may or may not involve translating one address into another.

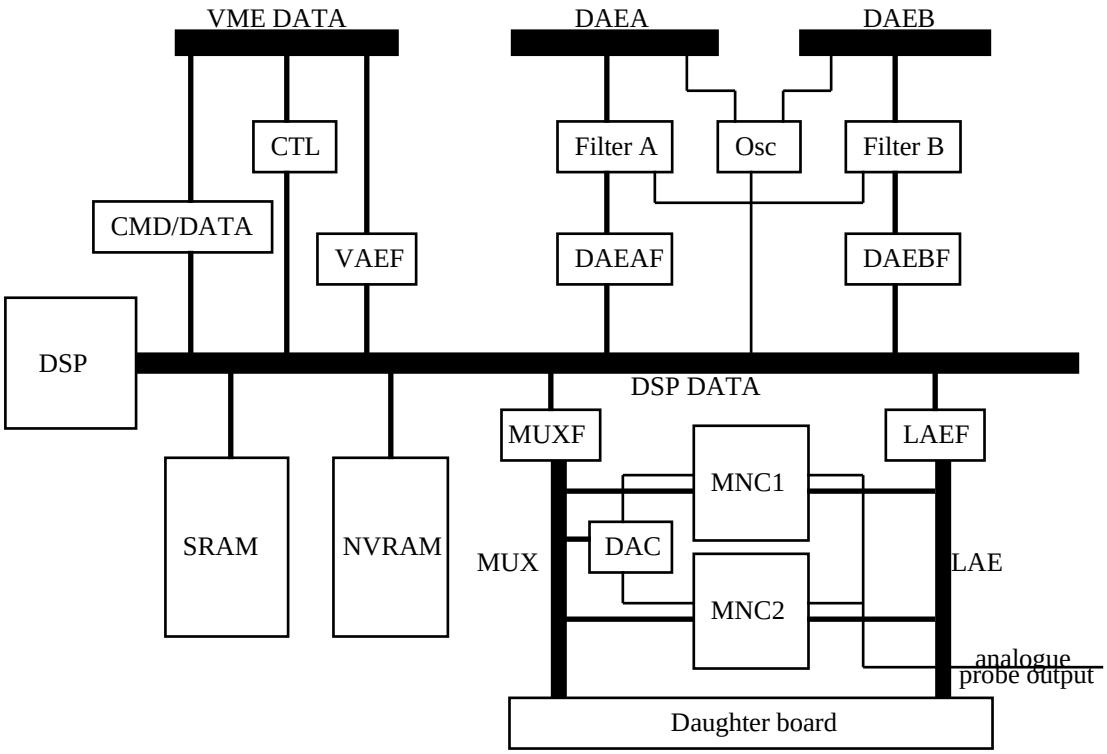
16 Analog parameters and synaptic weights

Analog parameters and synaptic weights are held on the MNCs as voltages on capacitors. These values need to be periodically refreshed by the DSP similar to the refreshing of DRAM. The DSP must periodically supply each MNC with the address of each synapse, and the correct analog voltage (via a DAC) to refresh each synaptic weight. Some neuron numbers (currently 0, 1, 2, 39 and 40) on each MNC do not represent neurons with synapses, but rather represent collections of parameters that globally affect the MNC. Refreshing these parameters is exactly like refreshing the synaptic weights on neurons 3 to 38.

17 Digital, latched parameters

Each MNC has a single analog output that acts as a probe into individual neurons on the chip. The source of this probe in terms of neuron number and compartment (position along the length of the neuron) and its mode (voltage sense or voltage clamp) must be selected by the DSP. The probe source and mode is determined by the MNC latching 0V and +5V analog values as digital 0 and 1 at certain parameter addresses (not restricted to any particular neuron numbers). The same mechanism is also used to set the most significant bits of the AEs generated by the neurons on an MNC, and for other purposes. Being latched, these parameters do not need to be refreshed.

18 SCX hardware features



Simplified block diagram of the SCX board hardware.

The CMD/DATA and CTL blocks connected to the VME data bus are compatible with the Message and Target Control registers used on the TMS320C5x evaluation module. This permits host and target software compatibility between the TI evaluation module and the Silicon Cortex. The SRAM consists of 64K words of zero wait state SRAM for program

memory and 64K words of zero wait state SRAM for data memory. The NVRAM consists of between four and thirty-two 32K word pages. All pages of NVRAM are accessible via I/O space, and one page is mapped into data space as global memory at reset. Note that all buses are connected to the DSP's data bus via FIFOs, and that each domain AE bus has a filter between it and its FIFO. These filters use 64K x 1 bit SRAMs to select which 16 bit AEs are passed across from the domain buses to the SCX board[4]. The block labelled Osc is the domain bus clock oscillator.

19 DESIGN CONCEPT

20 Key functionality

At its most basic, the software must perform the following functions:

- Refreshing of MNC analog parameters and synaptic weights.
- Setting the source and mode of the analog probe on each MNC.
- The one to many mapping of local AE (source) addresses into synaptic (destination) addresses.
- Loading of new parameter and synaptic weight values from the host system (both complete sets of values and individual values).
- Loading of new sets of source to destination address mappings from the host system.
- Booting, i.e. loading of code (either from the host system or the on-board NVRAM) and system initialisation.

21 Interrupts

Interrupts may be generated on various FIFO conditions, or when the host writes to the message register, or when it is time to refresh one of the MNC analog values. Given that the 'C50 hardware does not support nested interrupts, it is very important that each interrupt service routine (ISR) be kept as short as possible, so as to avoid losing data through FIFO overruns. To achieve this aim, each ISR will perform the minimum amount of work possible, typically just storing a data item in a buffer rather than carrying out any processing on that data item.

22 Data flow diagram

The current design has been arrived at principally by considering the data flow diagram (DFD) below. Note that this is a simplified view. It does not show the following features that would only serve to confuse the diagram while adding little that helps to understand the structure of the software:

- Hardware initialisation.
- The domain AE bus SRAM filters. These might be considered to be an extension of the "Address mappings" data.
- Data paths to allow the host to read back from the various tables and buffers in the system. This just adds arrows on the diagram back from the "Parameters and synaptic weights", "Analog probe select" and "Address mappings" blocks to the "Command processor" block, and from there back to the host.

4 Future versions may use 256K x 1 SRAMs to filter wider AEs.

- Maintenance of statistics. It may be useful to record statistics such as the number of AEs processed, the number of AEs forwarded onto other buses, the number of synaptic addresses generated, the number of parameter and synaptic weight refresh cycles executed etc.

- Error handling, for instance in response to a FIFO overrun, and the alerting of the host to such a condition.

- Control of front panel LEDs and scope trigger outputs.

- Uses of NVRAM other than for storing program code. Address mappings, parameters and synaptic weights may also be stored in NVRAM.

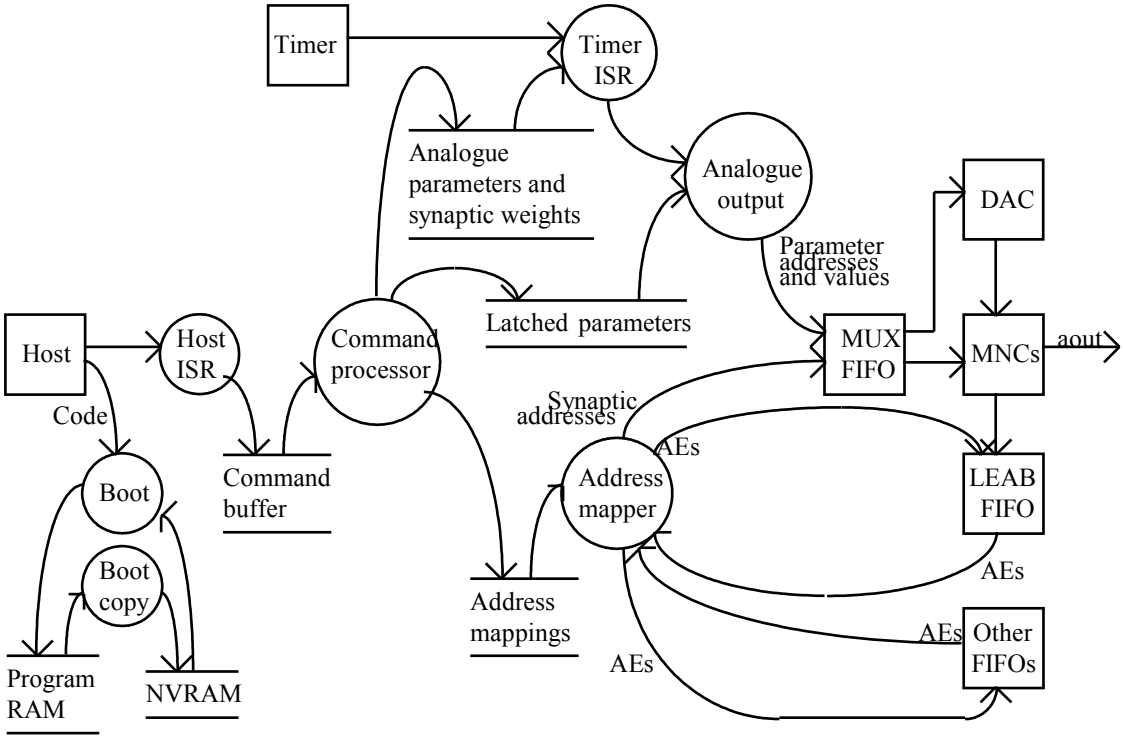
The DFD shows the following:

- The boot process, with code transfer possible from host to program RAM or NVRAM to program RAM.

- A boot image copying process that can transfer code from program RAM to NVRAM.

- A host ISR, responsible for receiving command and data words from the host and filling the command buffer.

- A command processor that reads commands from the command buffer and causes new parameter, synaptic weight or address mapping data to be transferred to the relevant module.



Data flow diagram.

- A timer ISR that takes parameter and synaptic weight addresses and values and passes them to the analog output process.

- The analog output process that writes parameter and synaptic weight values to the DAC and parameter and synapse addresses to the MNCs via the MUX FIFO.

- An address mapper that removes AEs from the AEB FIFOs, looks them up in an address mappings table to translate them into synaptic addresses and destination AEs and emits these to the MNCs and back to AEBs as required.

Note that although I have used the term process, this is just DFD terminology. I do not mean to imply that these will necessarily be separately executing processes in a multi-tasking environment.

23 Prototyping

24 Booting

Investigation of TI's on-chip boot loader reveals it is very simple in that it can only load a single contiguous section of code and therefore cannot directly load multi-section programs such as those produced by the TI C compiler. A more sophisticated form of boot loader or a boot loader assisting program will be required to boot the rest of the software if any part of it is written in C, unless a workaround can be used. Appendix A contains further details.

25 Interrupts

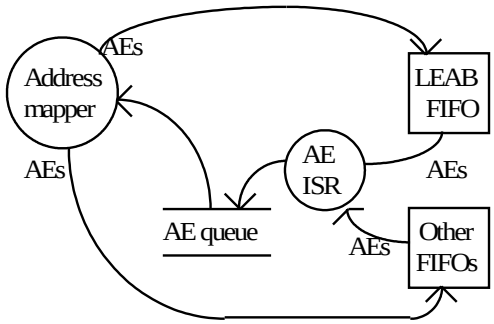
When an interrupt driven design was being considered (see following section) a reasonably realistic AE ISR was prototyped in TMS320C50 assembler that reads from a memory mapped I/O port and writes the value read to an AE queue implemented as a circular buffer. This routine was run on the TMS320C50 evaluation module (EVM) and timed at 1.4µs, including the time to execute a calling INTR instruction. This was with a 20MHz clock, zero memory wait states and one I/O wait state, but with no branch instructions in the routine, i.e. not allowing for any code to determine the source of the interrupt and hence which input port to read from. The processor on the SCX board was expected to run at 80MHz, at which speed the routine would run in 0.35µs. The wait states required on the SCX board were expected to be the same as those used on the EVM.

Now that the software is to poll the FIFOs rather than read from them under interrupt, this element of prototyping work has lost its direct relevance.

26 Alternatives

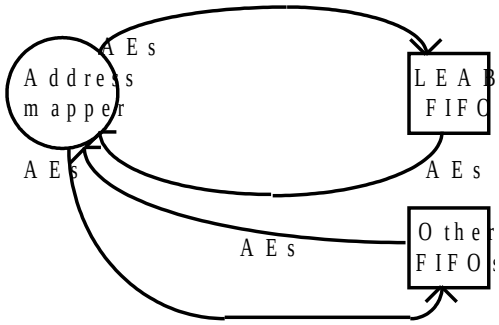
27 Polled versus interrupt driven AE processing

Originally, an interrupt driven design was considered. Whenever there was an AE to read from one of the FIFOs an interrupt would be generated. The AE would then be read and placed into a queue that is in turn emptied by a foreground process.



Interrupt driven AE processing

Instead of this it would be possible to have the foreground process simply poll the FIFOs and remove AEs directly from them for immediate translation. The FIFOs would be polled in a round-robin fashion to make sure that AEs from all four AEBs can be processed.

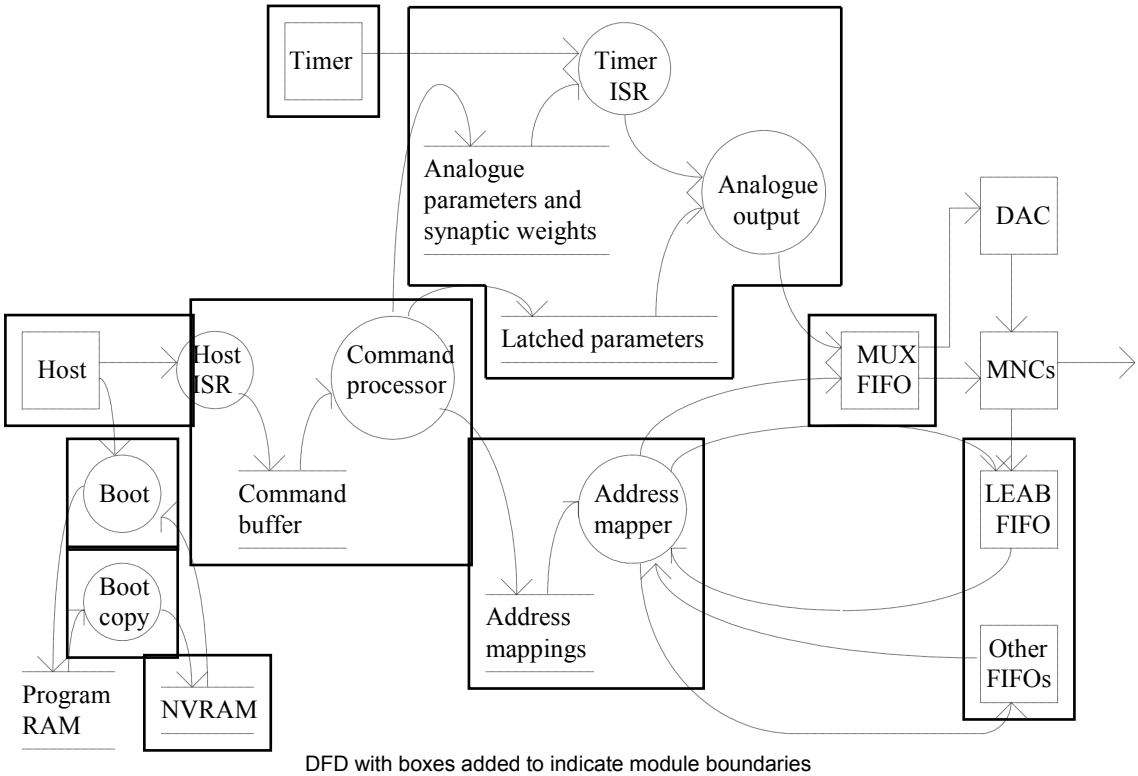


Polled AE processing

In the interrupt driven design, if there are ever any AEs in the highest priority (LAEB) FIFO, these must be transferred to the AE queue before any other processing of AEs on the domain buses or emptying of the AE queue can happen, i.e. with different FIFOs assigned to different interrupt levels, one FIFO could block all of the others and they might then overrun. Why have the AE queue being managed by software? Why not treat the FIFOs as the AE queue and just have the foreground process polling the FIFOs? The polled alternative is to be preferred because it has the merit of simplicity and ease of debugging, in addition to being less prone to FIFO overrun and hence less prone to data loss. Interrupts will of course still be used for other purposes: parameter and synaptic weight refresh (timer interrupts), host communications, and FIFO overrun signals.

28 IDENTIFIED COMPONENTS

The following sub-sections describe the components that have been identified by considering the DFD in terms of processes and objects.



DFD with boxes added to indicate module boundaries

29 Parameters and synaptic weights module

This module encapsulates tables of analog parameters and synaptic weights, routines to read from and write to these tables, code to refresh the MNCs (via the MUX FIFO module) based on the values in the tables, and routines to handle digital, latched parameters. In an idealised design, parameters and synaptic weights would be held separately as they have, in principle, little in common. However, as described in the overview above, the current implementation of an MNC presents a view to the outside world of parameters being treated as a special case of synaptic weight, and both analog parameters and synaptic weights need to be refreshed in an identical manner. Therefore in this design they will be held together in one module. In the host software they may still be treated separately if desired.

Analogue parameters and synaptic weights will be referred to together as refresh items and will be kept in one table. Digital latched parameters will be kept in another table. The parameter tables will hold address-value pairs rather than just an ordered list of values as the required addresses do not form a contiguous range. The tables may be held in program memory or in NVRAM if they are too large to hold in local data memory.

The following functions will be required:

- Set an individual latched digital parameter
- Set an individual refresh item
- Set a list of latched digital parameters
- Set a list of refresh items

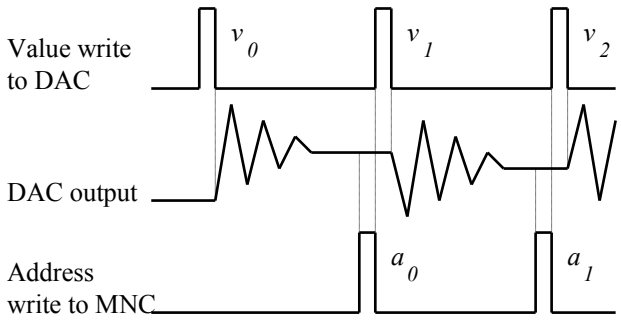
- Delete a latched digital parameter
- Delete a refresh item
- Reset the latched digital parameter table
- Reset the refresh item table
- Get the value of an individual latched digital parameter
- Get the 'written' status of an individual latched digital parameter
- Get an 'all-written' status for the latched digital parameters (i.e. have all the latched digital parameters been written to the MNCs yet)
- Get the value of an individual refresh item
- Set refresh mode
- Get refresh mode
- Set refresh interval
- Get refresh interval
- Start refresh
- Stop refresh
- Get status (refresh running, no. of refresh cycles executed etc.)
- Store state to NVRAM
- Load state from NVRAM
- Trigger scope on each refresh cycle

The set individual parameter functions will either modify an entry in the existing table or add a new parameter to the table as appropriate. Each parameter will contain the whole set of six chip select bits, so a single parameter may be latched by, or may refresh, multiple devices. Such multiple select parameters may be split into separate parameters or recombined into one if subsequent set individual parameter calls specify different sets of chip select bits.

The refresh interval supplied to this module will specify the maximum length of time that can be allowed to elapse before any given refresh item must be refreshed. This module will need to divide this by the number of analog parameters and weights currently being held in the table (plus 1 to allow latched parameters to be inserted into the output stream, see below) to obtain the interval required between refreshing the successive analog parameters and weights given in the table. It is expected that the refresh item voltages on the on-chip capacitors will decay at the rate of the order of 1mV/s and that each item will therefore need to be refreshed approximately once per second. Current MNCs have 492 analogue parameters and 648 synaptic weights, making a total of 1140 refresh items. Therefore one item needs to be refreshed approximately every 876μs.

As can be seen from the timing diagram below, it is necessary to wait for the DAC output to settle after writing to the DAC before writing an address to the MNC. This is achieved by not writing the address a_i until immediately before writing the value v_{i+1}. The diagram shows a

very long DAC settling time in proportion to the interval between refreshes simply to illustrate the principle. In practice the DAC settling time is likely to be 10μs or less, compared with the 876μs refresh interval.



Timing diagram for writing analog parameters and synaptic weights to the DAC and the MNC.

If the DAC settling time is sufficiently small, it would be possible to refresh many parameters and weights in a burst on one less frequent interrupt, rather than refreshing only one parameter at each interrupt. This approach would be particularly appropriate if many of the parameters and/or weights had the same value. In this case the analog value would only need to be set once before a group of addresses were output. Such an interrupt would necessarily take longer than an interrupt during which only one address and one value was written, therefore FIFO overrun and consequent data loss would be more likely to occur using this approach. However, this method of refresh will be available as a run-time selectable mode.

When required to set a latched parameter the module will write the parameter address and value to the latched parameter table (separate from the refresh item table) and mark it as an unwritten parameter. Unwritten latched parameters will then be inserted into the stream of parameter refreshes on subsequent call-backs if refresh is running and will be marked as having been written. Only 1 unwritten parameter can be inserted per cycle of the refresh table, otherwise some parameter(s) will be refreshed insufficiently often. If refresh is not running, a latched parameter value will be written to the DAC immediately using the write analog value and wait function in the MUX FIFO module. The table of latched parameters will also support a 'get' function.

This module will do little parameter checking on its functions. Any neuron and parameter address pair will be accepted by all of the 'set' functions. It is up to the host to ensure that the refresh item functions are called with addresses that make sense for those functions, and that the latched parameter functions are called with addresses that make sense for those functions.

The scope trigger function will set a flag such that one of the front panel scope triggers goes high at the start of the each refresh cycle and goes low when a chosen item is refreshed. The scope trigger outputs will be controlled via the front panel module.

30 Timer module

The timer module encapsulates the DSP's on-chip timer. It will provide the following functions:

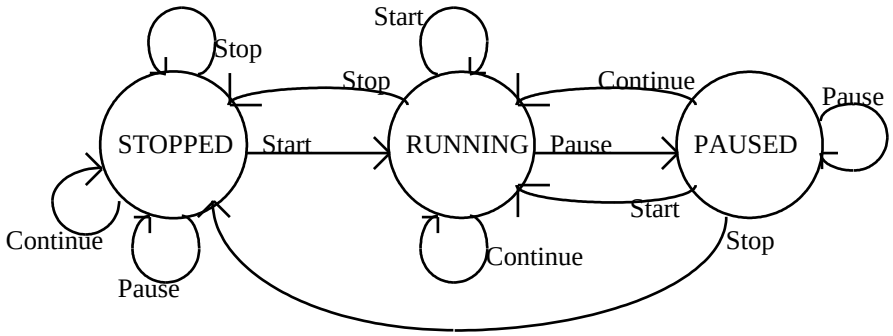
- Set period
- Set call-back address
- Start call-backs
- Stop call-backs
- Pause call-backs

- Continue call-backs
- Wait a number of microseconds

The set period function will cause the DSP's on chip timer to be programmed to generate interrupts at an appropriate rate, but the timer will only run and generate interrupts and call-backs between start and stop calls. When the timer is running, the call-back address will be called under interrupt once per period, unless the timer is paused. If the timer is paused, it will continue to run, but no timer interrupt will occur. On continuing from the paused state, any pending interrupt will be taken. Thus the pause and continue functions can be used to temporarily gate timer interrupts and hence call-backs.

Only one active call-back will be supported in the first implementation since this is all that is required. This could be extended in future revisions of the software to support multiple call-backs if necessary.

The wait function will always wait at least the given time irrespective of whether interrupts are enabled. This function is required to support the write analog value and wait function in the MUX FIFO module.



State transition diagram for timer stop, start, pause and continue.

31 VME host interface module

The VME host interface module will contain code to manipulate and service the VME host interface as distinct from the VAEB. The VAEB is just like the other AEBs in the system and does not interact with the host interface, therefore it is handled by the AEB FIFO module. The VME host interface module will contain the following routines.

- Initialisation routine
- Interrupt service routine for interrupts from the host
- Send command return value to host
- Send message to host

From a hardware point of view, the host interface is derived from the EVM host/target interface. It consists of a single message register that may be accessed by the target at either of two addresses in its I/O space and by the host at either of two addresses in its I/O space. In both cases, one of the two addresses is designated the command location; the other address is designated the data location. Associated with this are four flags, two visible to the host (TCMDINT, TDATAINT) and two to the target (HCMDINT, HDATAINT). A command interrupt flag (TCMDINT or HCMDINT) is set when the command location on the opposite side of the interface is written to and cleared when the command location on the same side of the interface is read. However, the read sets the command flag on the opposite side of the interface and this must be cleared by a dummy write (acknowledge) on the opposite side of the interface. A data interrupt flag (TDATAINT or HDATAINT) is set when the data location on

the opposite side of the interface is accessed and cleared when the data location on the same side of the interface is accessed. (Refer to the EVM technical reference.)

The host is a slow device in comparison with the SCX. Therefore it is desirable that multi-word commands be read from the host a word at a time. Each word transfer should generate a separate interrupt rather than having just the initial word of each multi-word command generating an interrupt during the servicing of which the remaining words are slowly handshaken in. However, certain conditions (e.g. a FIFO full condition) on the SCX board require that a message be immediately conveyed to the host, even in the middle of a sequence of words being transferred from the host.

The protocol for communicating between host and SCX must allow this even though only one register is available for transfers in both directions. This can easily be achieved provided that the host is interrupt driven. To send command words to the SCX, the host will write to its command or data location and the SCX ISR will read the word from its command or data location. To send messages to the host, the SCX will write to its command and data locations. When the host reads a message word from its command or data location, this will trigger an SCX interrupt that will act as a request for the next message word.

If an interrupt driven host cannot be assumed and the host must poll the TCMDINT and TDATAINT flags instead of being interrupt driven, then there will be a window between the host finding TCMDINT clear and writing a command word to the message register. During this window, the SCX might write to the message register thus setting TCMDINT too late to prevent the message register contents being overwritten by the command word from the host. To avoid this, the SCX will set the XF flag in the host control register to indicate a waiting message and the host must poll this bit in addition to TCMDINT and TDATAINT. When the host sees the XF bit set, it will complete any command that is in progress and then send a get message command that will return the message contents. This is not as immediate as might be desired, but prevents over-complicating the protocol in what will be a relatively slow polled system in any case. For any multi-word messages, the host will have to send multiple get message commands. This will require the queuing of messages to be sent to the host; the message queue will be maintained by the VME host interface module.

If there is no host attached to the SCX, requests to transmit a message to the host will simply be discarded. A host will be assumed to be absent unless and until an interrupt due to a command word write is seen.

Each word of the multi-word commands being sent by the host will be passed on to the command processing module and the words will be reconstructed into complete commands in that module. The command module will notify the VME host interface module when it has a complete command pending execution, and when a command has been executed the command module will call the VME host interface module back to send the return value back to the host.

32 Command processing module

The command processing module will be passed command words via an entry point called from the VME host interface module. These command words will be built up into complete commands in a buffer within this module. When a complete command has been assembled, one of two actions may be taken. 1 - A routine determined by the nature of the command will be called immediately (under interrupt). 2 - A flag will be set that will be polled by the main foreground address mapping routine in between reads of the AEB FIFOs. When the foreground routine finds this 'command ready to execute' flag is set, a routine determined by the nature of the command in the buffer will be called.

Variable length commands will be supported. A command that results in a call to a function that takes no parameters might be a single word. A command that results in a call to a function that has many parameters might have at least as many words as the function has parameters. A common format for the commands will be used, the first word of which might be thought of as an opcode. A bit in this word will indicate whether the command is to be executed immediately under interrupt, or posted for foreground execution later. The three

LSBs will give the number of further words in the command, unless these bits are all set, in which case the whole of the next word will be used to give the number of further words in the command.

For simplicity of protocol, two words of return value will always be passed back to the host for every function called. Thus a kind of RPC from host to SCX will be possible for certain functions in the SCX software. The host must wait for the return value from one command / function call before submitting another command.

33 MUX FIFO module

This module will encapsulate the functionality of the MUX FIFO. It will provide the following functions:

- Enable / disable FIFO
- Get FIFO enabled / disabled state
- Reset FIFO
- Set FIFO almost full and almost empty stop positions
- Enable / disable FIFO almost full interrupt
- Get FIFO almost full interrupt enable / disable state
- Get FIFO condition
- Get FIFO almost full stop position
- Get FIFO almost empty stop position
- Write address for analog load
- Write address representing synaptic event
- Write analog value to DAC and wait for output to settle
- Write analog value to DAC without waiting for output to settle
- Get last value written to DAC
- Get daughter board status (present or absent)
- Get statistics (number of writes, number of almost full events etc.)
- Trigger scope on write to particular synapse

When the FIFO is disabled it will be held in reset and writes to the FIFO will be ignored. The FIFO's default almost full and almost empty stop positions will be used unless the function to set the stop positions has been called. The values set by this function will only take effect when the FIFO is reset or is enabled having previously been disabled.

The difference between the two write address operations is the state required for the 'load' bit. When address and load values emerge from the FIFO, if the load bit is set, the MNC will treat the address as the address of an analog parameter or synaptic weight and load the analog level currently being output by the DAC. Both write address functions will accept two

parameters: (i) an address to write and (ii) a chip select value to select which chips the address is written to.

Either write analog value call will cause this module to write the new value to the DAC via the FIFO unless the new value is the same as the last value written. A 'last value written to DAC' local variable will be maintained that can be read using the get last value written call. If the 'wait' variant of the set analog value call is called and a new value was written to the DAC, the timer module will be called to wait the length of the DAC output settling time before the set analog call value returns. If the new value was the same as the old value, and therefore no new value was written to the DAC, then the wait variant of the call will behave like the no wait variant and return immediately. The MUX FIFO module handles writing to the DAC instead of there being a separate module to handle the DAC because the DAC is accessed via the MUX FIFO and writing to the DAC interacts with writing to other devices via the MUX FIFO.

The scope trigger function will cause one of the front panel scope triggers to be activated via the front panel module whenever the supplied address representing a particular synaptic event is sent to the FIFO.

In addition to the functions listed above, this module will contain an interrupt service routine. When MUX FIFO almost full interrupts are enabled and the MUX FIFO becomes almost full, the ISR will turn on a front panel LED via the front panel module and alert the host via the VME host interface module.

34 AE mapping module

This module is at the core of the application. Its main loop will inspect each AEB FIFO and the command processing module's 'command ready to execute' flag in turn. Ideally, address events would be read from each 512 word deep AEB FIFOs in proportion to the inward data rate on that FIFO. A crude approximation to this will be achieved by reading from the FIFOs according to the following (or a similar) scheme before passing on to the next FIFO (or 'command ready to execute').

FIFO condition	Number of words to read
Empty	0
Almost empty	min(64, number in FIFO)
None (> almost empty, < half full)	128
Half full	192
Almost full	256
Full	32

Only a few AEs are read in the full condition to prevent a 'run-away' bus occupying too much processor time. The FIFO full condition will be signalled to the host, and the host may intervene to determine and/or remedy the cause of the overrun.

Once per cycle of the AEB FIFOs the command processing module's 'command ready to execute' flag is examined and if set, the command is executed.

35 AE mapping, method 1

As each source AE is read from a FIFO, it is processed as follows. Firstly the address is looked up in an array in NVRAM that has a one word entry for each possible 16 bit AE. This array will generally be sparsely populated except for the area(s) corresponding to locally generated AEs. The words in this array will be pointers to structures of the following form:

```
{
        LAEB re-transmit flag
        DAEB0 translation address
        DAEB1 translation address
        VAEB re-transmit flag
        Number of synaptic (MUX) addresses, S
        {
                Chip selects
                Neuron number
                Synapse number
        } [ S ]
}
```

If either the VAEB or LAEB transmit flag is set, the source AE will be re-transmitted on the appropriate bus. Address translation is not required for these buses. If either domain bus translation address is non-zero, then that address will be transmitted on the appropriate DAEB[5]. Any number of synaptic addresses to be sent to the MUX FIFO can be specified in terms of chip, neuron and synapse numbers.

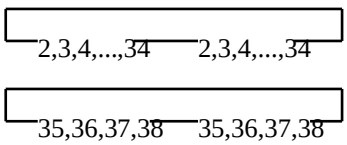
36 AE mapping, method 2

An alternative, less general, but more biologically representative method of processing the AEs would use structures of the following form:

```
{
        LAEB re-transmit flag
        DAEB0 translation address
        DAEB1 translation address
        VAEB re-transmit flag
        {
                Neuron population number
                Base neuron index
                Projective field type
        } [ 4 ]
}
```

Up to four projective fields are allowed per source AE to specify the addresses to be written to the MUX FIFO.

Neuron populations would be defined over one or more MNC (or other device) as contiguous ranges of neurons on each MNC. These ranges are considered to be adjacent and to form single circular populations. For example an excitatory population might be defined as neurons 3 to 34 inclusive on MNCs 1 and 2 so that neuron 34 on MNC 1 is considered to be adjacent to neuron 3 on MNC 2 and the population 'wraps-round' so that neuron 34 on MNC 2 is considered to be adjacent to neuron 3 on MNC 1. Another, inhibitory, population might be defined as neurons 35 to 38 on MNCs 1 and 2.



Example of neuron populations.

Each projective field type number will be an index into an array of pointers to arbitrary sized arrays of neuron offset and synapse number pairs. There will be a maximum of sixteen projective field types supported by any one SCX board. The neuron offsets will be added to the destination base neuron number in the given circular population. Thus in the example above, if the neuron population number identifies the inhibitory (35,36,37,38,35,36,37,38) population, the base neuron index specifies the 7th neuron

5 Zero is assumed to be not a valid AE.

and neuron offsets specified by the projective field are 0, +1 and +2, the actual neurons to be written to will be neurons 37 and 38 on MNC2 and neuron 35 on MNC1.

Method 1 has the benefits of simplicity, and hence speed of mapping, and of generality. Using this method, division of the MNC neurons into populations, definition of projective fields and projection of neurons onto populations with wrap-around may be handled by the host, resulting in a simple, single stage mapping operation to be carried out as each AE is processed by the DSP. The generality of the method means that any mappings that did not fit the projective field and wrapped-round neuron population model implicit in method 2 could be handled without modification to the algorithm of method 1.

This module will provide the following functions to control the state of the mapping tables:

- Set / delete a complete AE mapping structure
- Get a complete AE mapping structure
- Get a list of currently mapped AEs
- Enable / disable reception of a given AE from a domain bus
- Enable / disable re-transmission of a given AE on a particular bus
- Set the list of destination (MUX) addresses for a given AE
- Add to the list of destination (MUX) addresses for a given AE
- Delete destination (MUX) addresses for a given AE
- Store state to NVRAM.
- Load state from NVRAM.

In addition to the members of the mapping structure shown above, each structure will have two members to specify whether reception of the AE from each domain bus is enabled or disabled. When a complete structure is set or deleted or the 'Enable / disable reception of a given AE from a domain bus' function is called, this module will call the domain AE buses module to control which individual AEs are received from the domain buses.

37 AEB FIFO module

This module will encapsulate the functionality of the bi-directional AE bus FIFOs. It will provide the following functions, each of which will take a parameter indicating which of the AE bus FIFOs it is to operate upon:

- Enable / disable FIFO input
- Get FIFO input enabled / disabled state
- Reset FIFO input side
- Enable / disable FIFO output
- Get FIFO output enabled / disabled state
- Reset FIFO output side

- Set FIFO input side almost full and almost empty stop positions
- Set FIFO output side almost full and almost empty stop positions
- Enable / disable FIFO interrupts
- Get FIFO interrupt enable / disable state
- Enable / disable input side overrun
- Get input side overrun enable / disable state
- Get FIFO input side condition
- Get FIFO input side almost full stop position
- Get FIFO input side almost empty stop position
- Get FIFO output side condition
- Get FIFO output side almost full stop position
- Get FIFO output side almost empty stop position
- Output to FIFO
- Input from FIFO
- Get statistics (number of reads, number of overruns etc.)
- Trigger scope on read of particular AE.

When either side of a FIFO is disabled it will be held in reset. Whilst the output side of a FIFO is held in reset, requests to output to it will be ignored. Whilst the input side of a FIFO is held in reset, requests to input from it will return 0. The output side of a FIFO's default almost full and almost empty stop positions will be used unless the function to set the stop positions has been called. The values set by this function will only take effect when a FIFO's output side is reset or is enabled having previously been disabled. The function to set the positions of the almost full and almost empty stops for the input side of the FIFO will have no effect because the currently planned hardware does not permit these to be changed. Nevertheless, this function and its corresponding 'get' functions will be retained for ease of future implementation.

In addition to the functions listed above, this module will contain interrupt service routines. When interrupts are enabled for a particular FIFO and either the input side or the output side of that FIFO becomes full, the ISR will turn on a front panel LED via the front panel module and alert the host via the VME host interface module.

When input side overrun is disabled for a given FIFO, the attached bus will be halted if the FIFO becomes full (a wait will be generated on that bus). When input side overrun is enabled for a given FIFO, the attached bus will not be halted if the FIFO becomes full (no wait will be generated on the bus).

The scope trigger function will cause one of the front panel scope triggers to be activated via the front panel module whenever the supplied AE is read from one of the FIFOs.

38 Front panel LEDs and scope triggers module

There are six LEDs and two scope trigger outputs on the SCX front panel. This module provides functions to set and get the state of these outputs. The functions will be as follows:

- Set LED pattern
- Set LED state
- Get LED state
- Toggle LED state
- Set scope trigger state
- Get scope trigger state
- Toggle scope trigger state

Each output will be dedicated to some purpose either for all time for a particular build of the SCX software or for some duration under the indirect control of the host software. For instance, the other modules of the SCX software may be built to use the LEDs as follows:

LED number	Function
0	Parameter refresh cycle 'heartbeat'
1	MUX FIFO almost full indicator
2	LAEB output FIFO full indicator
3	DAEB0 output FIFO full indicator
4	DAEB1 output FIFO full indicator
5	Input FIFO full indicator

The scope triggers will typically be 'allocated' by the host software requesting one of the modules in the SCX software to toggle a particular scope output when some event occurs. No resource claiming mechanism will be implemented to prevent the same scope trigger output being manipulated by more than one SCX module at a time.

39 Domain AE buses module

This module will handle the special features of the domain AE buses not covered in the AEB FIFO module, i.e. the programming of the domain bus clock speeds and domain bus SRAM filters. The following functions will be supported:

- Get domain bus controller status
- Get domain bus termination status
- Set domain clock speed
- Get domain clock speed
- Set SRAM filter bits

The 'Set SRAM filter bits' function will be called from the AE mapping module to set the state of both of the domain bus SRAM filter bits for a given AE. This function will set the hardware into the mode required for this operation, set the filter bits as required and return the hardware to the normal operating mode. There will be no 'get' function as the hardware does not support reading back the filter bits, and this module will not maintain copies of the filter bit

states as this would be duplicating information that will be available within the AE mapping module.

The set domain clock speed function will accept, and the get function return, frequencies specified in Hz rather than domain bus oscillator control words in order to isolate the host from the details of the hardware.

40 NVRAM management module

This module will manage the available NVRAM as if it were divided into a set of permanently allocated, fixed size blocks. Each module that uses NVRAM will know the unique handle(s) of the block(s) of NVRAM that it uses and will supply a handle to a block and an offset within the block for each word that it wishes to read or write. Although the NVRAM is structured in 32K pages, the callers of this module will not need to know about the NVRAM pages, and blocks may span more than one page. The following functions will be supported:

- Read word
- Write word
- Get total NVRAM size

The read and write functions will internally convert the supplied handle and handle offset to page and page offset values, map the required page into I/O space and return or set respectively the required word on the page.

The 'Get total NVRAM size' function will enable the host to determine whether the SCX board is fitted with 4, 8, 16 or 32 32K word pages of NVRAM.

NVRAM will be required to store the following:

Program code and static data	Up to 32K
Parameters and synaptic weights	Not more than 4K per current MNC.
AE primary lookup table (sparse array)	64K
AE mapping structures	Up to 64K

The limits on the memory requirements for program code and static data and the AE mapping structures are set by the 64K size of the program and data spaces that these items occupy at run time. The size of the AE primary lookup table is dictated by the 16 bit width of an AE. The estimate of 4K of parameters and synaptic weights per MNC is for the current 36 neuron MNCs with no parameters or weights being held in common for multiple MNCs.

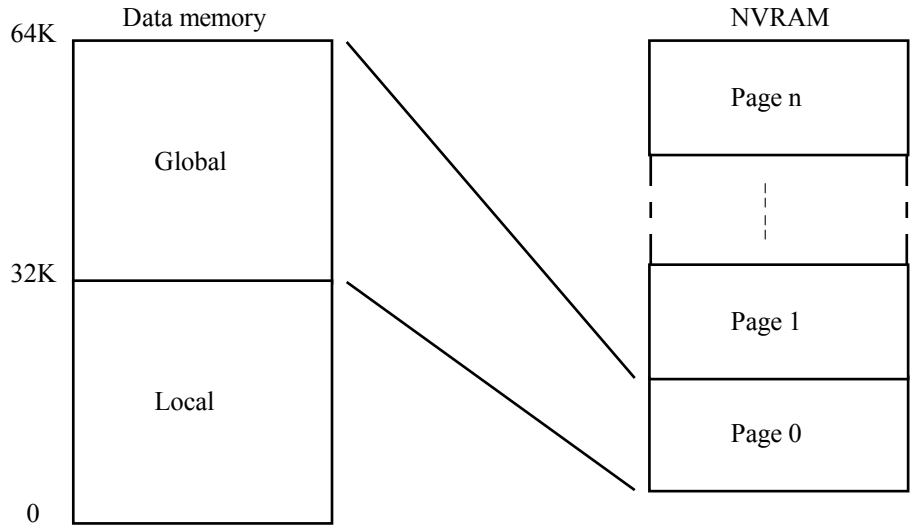
Program code and static data will be copied to program space by the boot loader. AE mapping structures and parameters and synaptic weights will be copied to data space by the 'Load state from NVRAM' functions in the relevant module. The primary lookup table will remain in NVRAM at run time.

41 Boot loader

See appendix A.

APPENDIX A - BOOTING

The MP/~MC pin will be held low so that the 'C50 resets into microcomputer (MC) mode, i.e. with on-chip ROM enabled. At reset, the 'C50 starts to fetch code from address 0 in program memory. The on-chip ROM at this location contains an unconditional branch to TI's Boot Loader. This inspects the boot routine selection word at data memory address 0FFFFh to determine the source from which to boot. At reset, the top 32K of data memory is mapped to global memory (GREG=10000000b) and this will be occupied by page 0 of NVRAM. Therefore the boot routine selection word is read from this NVRAM.



Two of the seven available methods are of interest: 16-Bit Parallel EPROM Mode and 16-Bit Parallel I/O Mode. The Warm Boot method is inappropriate, since this simply causes a jump to an address in program memory specified by the boot routine selection word. A power off-on cycle (or anything else) that destroyed the contents of program memory would leave the boot routine selection word stored in non-volatile RAM pointing to invalid code.

16-Bit Parallel EPROM Mode

16-Bit Parallel EPROM Mode is selected by placing the pattern XXXXXXXXSSSSSS10 in the boot routine selection word where X = don't care and SSSSSS is a 6-bit source page address. This will be the normal boot method, used to copy code not from an EPROM but from the NVRAM in global memory. Given 64K external program RAM, TI's on-chip Boot Loader could in principal be used to transfer up to 32K[6] contiguous code words from NVRAM into program memory anywhere in the range 0800h to FFFFh. (On-chip ROM will still be mapped in below 0800h.) This should be sufficient for our purposes, as we expect our code to be less than 32K, and to use up to the remaining 32K of program memory to store parameters. The first action of our code, once started by TI's boot loader, could be to set the 'C50 into Microprocessor (MP) mode, thus gaining 2K extra program RAM (in the region previously occupied by ROM) and enabling interrupt vectors to be changed without changing IPTR. Our code would then proceed to initialise hardware, load data tables from other pages of NVRAM etc. Note that if MP mode were not selected, interrupt vectors could only be changed by changing IPTR to point into RAM, however this is probably the desired method of working as TI's boot loader can only load one block of code, so this must include the interrupt vectors.

How would the code, interrupt vectors and the boot routine selection word get into NVRAM in the first place? Code would originally be introduced into the system via 16-Bit Parallel I/O Mode boot.

16-Bit Parallel I/O Mode

16-Bit Parallel I/O Mode is selected by placing the pattern XXXXXXXXXXXX1100 (where X = don't care) in the boot routine selection word at the top of the NVRAM page that is mapped into global data memory at reset, or by releasing the front panel Reset button while holding the NMI button depressed.

[6] 32K is the maximum the boot loader can load directly, since it can only load a single contiguous block, and 32K is the maximum amount of NVRAM that can be mapped into memory at any one time.

This forces the four LSBs to be read from the ??? bus to be read as 1100 when the boot routine selection routine is read. Once this has been done, code can be loaded via the 'C50's port 50h under control of TI's boot loader. However, TI's boot loader can only load a single contiguous block of code. Programs written in C consist of multiple, not necessarily contiguous sections. The DSPHEX tool provided by TI can only convert single sections to boot table format. Three options are available to work around this limitation:

1. Write a more sophisticated boot loader for the 'C50 that can load multiple sections. Booting then becomes a two stage process. Firstly the TI boot loader loads our more sophisticated boot loader. This then loads multiple sections via port 50h using an enhanced multi-section form of boot table as follows:

Number of sections S
Entry point address
Section 1 destination address
Section 1 length N_1
N_1+1 code words
Section 2 destination address
Section 2 length N_2
N_2+1 code words
. . .
Section S destination address
Section S length N_S
N_S+1 code words

   Before beginning to load from this multi-section enhanced boot table, our boot loader could set MP mode, and copy itself to the address where on-chip ROM is situated in MC mode. This would leave the same area of program memory free for useful code as is available on reset, and an interrupt vectors section could be loaded at address 0.

   This option would also require a PC program to be written to supply this form of boot table to the 'C50 from one or more hex format files produced by DSPHEX. (Eventually directly from a COFF file?)

2. Write a PC program to convert multi-section (not boot table format) output from DSPHEX (or eventually direct from a COFF file?) to boot table format and supply this to the 'C50. Any gaps between sections would be filled with some fill value (DSPHEX can do this). This option is less flexible than option 1 as an interrupt vector section could not be loaded at address 0.

3. Use the TI linker to re-link the multi-section COFF file to produce a single section COFF file from which DSPHEX can generate a single section boot table to be supplied to the 'C50's boot loader.

Once booted using 16-Bit Parallel I/O Mode, a 'C50 routine will be able to construct a boot table in NVRAM in global data memory simply by copying the whole range of memory occupied by code. The boot routine selection word can then be changed to specify 16-Bit Parallel EPROM mode ready for the system to boot from NVRAM.

Summary

The overall process of booting goes as follows (assuming method 3 above is used):

42 Select 16-Bit Parallel I/O mode by holding the NMI button depressed while releasing the Reset button.
43 'C50 resets into MC mode.
44 'C50 ROM Boot Loader boots SCX application from port 50h.
45 'C50 ROM Boot Loader passes control to SCX application.
46 SCX application initialises hardware, loads data tables etc.
47 SCX application runs.
48 SCX application selects 32K global data memory / NVRAM.
49 SCX application writes boot table header (destination address and length) into NVRAM at address 8000h.
50 SCX application copies an image of all sections of code (e.g. all addresses from 0800h to 7FFFh) to boot table in NVRAM.
51 SCX application sets boot routine selection word to xx82h (16-Bit Parallel EPROM mode, source address = 8000h).
52 SCX application continues to run.
53 'C50 is reset again (into MC mode).
54 'C50 ROM Boot Loader loads image of SCX application from NVRAM.
55 'C50 ROM Boot Loader jumps to address given by the destination address in the boot table in NVRAM (ie. to the SCX application)
56 SCX application initialises hardware, loads data tables etc.
57 SCX application runs.

GLOSSARY

AE	Address-Event.
DAC	Digital to Analog Converter
DAEB0	Domain Address-Event bus 0. One of two address-event buses connecting multiple cortex boards.
DAEB1	Domain Address-Event bus 1. One of two address-event buses connecting multiple cortex boards.
DFD	Data Flow Diagram.
DRAM	Dynamic RAM.
DSP	Digital Signal Processor, a type of microprocessor.
FIFO	First In, First Out.
LAEB	Local Address-Event bus.
MNC	Multi-Neuron Chip.
Neuron	A nerve cell or, in this context, its silicon analog.
NVRAM	Non-volatile RAM.
Parameter	Analog value affecting all neurons on a given chip.
Projective field	The set of synapses on other neurons activated by a given neuron.
Refresh item	Either an analog parameter or a synaptic weight
RPC	Remote Procedure Call.
SCX	Silicon Cortex.
SRAM	Static RAM.
Synapse	Junction between neurons.
Synaptic weight	Analog value representing the strength of the connection at a synapse.
VAEB	VME Address-Event bus.
VME	A microprocessor bus standard (IEC821BUS / IEEE P1014), see reference 12.

- o O o -