# Contents

# 1 Introduction

STEP1 (or simply STEP in this manual) is an indefinitely scalable multiprocessor architecture optimized for the fine-grained simulations of physical systems. From a functional viewpoint, the STEP architecture can be thought of as a spatially-extended, uniform, fine-grained medium of a rather general and abstract nature. Specific behavior is obtained by specifying the medium's connectivity and dynamical properties, and assigning its initial state. In this sense, STEP embodies the metaphor of "programmable matter."

This manual discusses those functional and structural aspects of STEP's hardware that are of interest to the programmer and the sophisticated user. It does not address issues of modeling methodology. Issues of digital design or electronic engineering are dealt with only insofar as they clarify the reasons for certain design choices, and thus have tutorial or mnemonic value.

## 1.1 System overview

Functionally, a STEP machine consists of a three-dimensional array of *modules*. The number of modules along each dimension is finite (though it can be arbitrarily large), with each dimension wrapping around on itself; thus, the module-array topology is that of a three-dimensional torus. A module contains all the resources (state variables and updating machinery) needed to simulate a certain portion (a *sector*) of the system under study. The modules all work in parallel and in lockstep, directly exchanging information only with contiguous modules.

The system being modeled is represented by a collection of discrete *unit cells* arranged in a regular *lattice*. This lattice may have more or fewer than three dimensions, it may have an arbitrary crystallographic structure[1] (in particular, it need not be a cubic lattice), and may or may not be wrapped around.

An equal amount of data storage capacity is allocated to every lattice cell, for representing in a distributed way the system's state variables, its parameters, and other information useful for processing and analyzing the system's state. Depending on the nature of the system, the appropriate cell capacity may range from a few bits to thousands of bits.

Cells within each sector are updated serially: on each updating pass through the sector a single processor is time-shared between the cells in such a way as to simulate the parallel operation of an equivalent number of identical virtual processors. A complete updating *step*, in which all cell state components are appropriately updated, may require several passes. Since the results of each pass are, in ordinary circumstances, independent of the order in which the cells are scanned, the scanning order itself can be chosen so as to accomodate particular display or analysis needs. Moreover, the updating algorithm may change from step to step, so as to implement a *time-dependent* dynamics.

Data communication between cells during each updating pass is not restricted to the immediate neighbors in the lattice. Indeed, one of the features that distinguish the STEP architecture from more "obvious" cellular automata architectures is that essentially instantaneous communication can be achieved between cells that are hundreds or thousands of positions apart. Thus, short- and long-range interactions are efficiently handled at the same time by the updating machinery.

Pipelining of address arithmetic, read, update, and write operations, together with "flash" updating based on lookup tables, yield a high rate of serial updating: for 16-bit cells, 25 Megacells/sec per module in the current implementation.

The STEP machine is controlled by a *host* computer through a *controller* board that sits on the host's bus. The controller communicates with the array of modules via a special tree-shaped bus, called the *TBus*, which distributes commands and data to the modules and collects status reports and data from them.

Physically, the basic building block of a STEP machine is a assembly of eight modules called a *box*. Besides a motherboard with eight slots for the modules to plug in, the box provides mechanical support, shielding, buffering of some signals, power and clock distribution, and TBus routing and fan-out. Boxes are physically stacked in a three-dimensional array, and exchange information only with adjacent boxes.

The smallest viable STEP machine consists of a single box populated by a single module. Conceptually, there is no upper limit to machine size; in practice, a limit is set by economic factors sooner than by engineering constraints. Installations consisting of hundreds or thousands of modules are feasible and, in terms of useful modeling applications, desirable.

Within each module, ordinary dynamic RAM chips (DRAM) are used to store cell data. The sytem's dynamics is defined by lookup tables, which are stored in ordinary static RAM chips (SRAM). The "glue" that turns these

[1] Cf. Neil ASHCROFT and David MERMIN, *Solid State Physics*, Saunders College (1976).

generic commodities into a powerful and flexible modeling environment is represented by a custom VLSI chip—called the STEP*chip*—of which there are a set of sixteeen in each module. This set works as a tightly integrated team, and is indeed called a STEP*chip team*.

A faithful and efficient software simulator for the STEP architecture is available. The current version runs on a SPARCstation, but is easily portable to other platforms. For many applications it is a viable approach to develop an experiment on the software simulator, on a lattice of tens of thousands of sites, and then run the *identical* experiment on a lattice of millions or billions of sites on a STEP machine.

## 1.2 Step's machine language

The STEP architecture gives one great flexibility in specifying the features of the conceptual model that is the object of the simulation. Subject to a single quantitative constraint, i.e., the number of STEP modules available, the user has great latitude in choosing the dimensionality of the lattice, its crystallographic group, and its extension in each of the dimensions; the size and the internal structure of the cells, as well as the interconnectivity of the cells within the lattice; the sequencing of updating steps and algorithmic details of each step; and, finally, the strategies for real-time data analysis and display and the related input/output tactics.

It must be understood that this great variety of high-level features are synthesized from a basic set of low-level, native features of the STEP architecture—a simple, well-structured, highly orthogonal set of machine primitives.

The main goal of this manual is to provide a "programmer's model" of STEP appropriate to this synthesis taks. That is, our chief concern will be to introduce, explain, and document those aspects of the hardware that are meant to visible to the programmer. The approach will be "bottom-up." We shall start with low-level concepts such as *sites* and *layers*, *kicks* and *glue*, *lookup tables* and *events*. Higher-level concepts such as *lattice*, *cells*, and *steps* will not be mentioned again for a while, but will emerge in due course.

Ultimately, STEP programming is reduced to writing to (and occasionally reading from) a set of hardware registers. Only about a dozen of these registers are of direct concern to the user, when programming at the machine-language level. Even if programming is done through a compiler, the user should be generally aware of the existence and function of these registers. The other registers—used for setup, interrupt control, and diagnostics—are handled by system calls and are of concern only to systems programmers.

## 1.3 Organization of this manual

Part I presents STEP's overall theory of operation. The last chapter of Part I is a detailed description of all the STEPchip registers: these registers constitute the functional link between software and hardware.

The STEP design naturally falls into four hierarchical levels, namely *machine*, *box*, *module*, and STEP*chip*. Part II describes each level of this hierarchy in terms of its interface to the rest of the system, and sketches how each level is built up from the lower one. For sake of reference, a systematic listing of all input and output signals ("pinout"), accompanied by a brief description of each signal, is given for each level, using a consistent system-wide nomenclature.

## 1.4 Conventions

Here we list a number of general conventions followed throughout this manual.

### 1.4.1 Number representation

Numerals in typewriter font are in base sixteen; thus `40`=64 and `FF`=255.

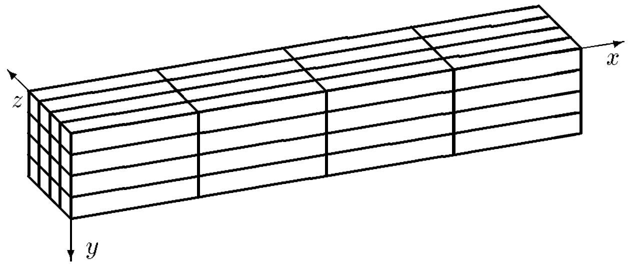
### 1.4.2 Data and variable ordering

In listings, tables, and figures, we had to fight a hard struggle to reconcile conflicting conventions concerning the ordering of items and the orientation of axes.

There are two basic sources of conflict. One conflict is between the "most significant first" approach, typified by the way we write a number such as '1992' starting with the thousands and ending with the units, and the "first causes first" approach, typified by the way the electronic engineer draws the stages of digital counter, starting with the units stage on the left and making the carry propagate left-to-right. The other conflict is between the way a mathematician would write an $i,j$ array, with $j$ increasing left-to-rigth and $i$ top-to-bottom, in agreement with western-style writing, and the way a physicist would draw an $x,y$ or an $x,t$ diagram on the page, with $x$ pointing rightwards but $y$ or $t$ pointing *upwards*. Note that the "mathematician's" approach is also the one used in scanning an ordinary CRT monitor—left-to-right and then top-to-bottom.

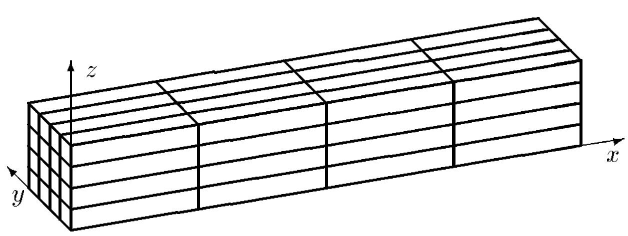
After much soul-searching, we took the responsibility to enforce a consistent, universal convention: in any enumeration, the natural ordering of the items (i.e., increasing values of the enumeration index) will be reflected by the typographical ordering (left-to-right, and, for a second index, top-to-bottom).

Thus, an array indexed by $x$ and $y$ will be displayed with $x$ increasing from left to right and $y$ from top to bottom, as in the following diagram




Here, the diagram's third axis, $z$, is represented as pointing *into* the page).

Note that, with these conventions, the "handedness" of the $(x, y, z)$ frame is the same as that that results from the more conventional ordering of axes, illustrated by the following diagram




Similarly, a 16-bit computer register will invariably be represented as follows

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

and a text description of this register will list individual bits or fields in same order, as in the following example:

| | |
|---|---|
| 0–6 | ASCII character |
| 7 | Parity bit |
| 8–15 | Not used |

When the contents of the register is interpreted as a binary-encoded integer by the computer's arithmetic-logic unit or its address-arithmetic unit, bit 0 will represent the least significant binary digit (units), and in general bit $n$ will represent the coefficient of $2^n$.

The above ordering convention will be broken in *only one circumstance*. In deference to universally ingrained practice, ordinary numerals in the text will be written as usual with the most significant digit first. Thus, the number represented by the following register contents

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | | | | | | | | | | | | | | | 15 |

will be written, in text form, as hexadecimal `00F1` or decimal 237.

### 1.4.3 Signal polarity

Signal names, typeset in sans-serif face, are chosen so as to have appropriate mnemonic value. For instance, the TBus Reset signal has a value of 1 (logical TRUE) when the STEP machine is being reset, and 0 (logical FALSE otherwise.

Physically, on the the TBus wires, this signal travels in complemented form, indicated by Reset*. However, we shall usually mention the signal in its native form, saying, for instance, that "in certain conditions Reset is asserted;" this implies that, in those conditions, the physical signal Reset* is negated.

The complemented form, Reset*, would be used only when tabulating the pinout of a chip or connector, to make it clear that the physical signal present on that pin is the complement of Reset.

# 2 Geometry and scanning

In this chapter we present the basic geometrical and kinematical aspects of STEP's data organization and data movement.

## 2.1 Sites and layers

STEP's main business is the fine-grained simulation of spatially-extended systems; this is achieved by the iterated updating of a large collection of bit-sized state variables according to laws that are, by and large, local and uniform.

One may visualize these bits as arranged in a multi-dimensional array—which we shall call STEP *space*—that plays in the simulated system a role somewhat analogous to that of ordinary physical space. Each element of the array consist of a collection of bits, which play a role somewhat analogous to that of state-components in ordinary physics. This arrangement is illustrated in Fig. 2.1, where dots represent bits. In this figure the array is two-dimensional, and the different state-components are drawn on separate planes for clarity.

We shall formalize the above concepts by saying that each state-variable bit in a STEP unit is uniquely identified by a combination of two labels, namely a *site address* ("which array element") and a *layer number* ("which state-component"). If we fix the site address and let the layer number run through all its possible values, we obtain a data structure called a *site* (the vertical "pile" of bits in Fig. 2.1). If we fix the layer number and let the site address run through all its possible values, we obtain a data structure called a *layer* (the horizontal "sheet" of bits in the same figure). The present implementation of STEP uses sixteen layers, numbered 0 through 15 or, where hexadecimal is more convenient, `0` through `F`.

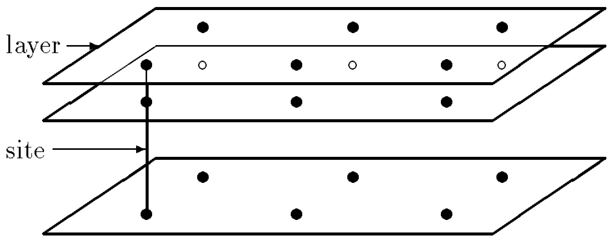



Figure 2.1: State-variable bits in STEP are arranged in *layers* and *sites*.

STEP layers can extend indefinitely in one, two, or three dimensions called *extendible* dimensions. By 'indefinitely' we mean that no limits are set by the architecture: an arbitrary extension along any of these dimensions can be achieved by connecting end-to-end an appropriate number of STEP modules. The last module of the chain will be connected to the first, so as to achieve wraparound (see Chapter 2.5 for more details). Thus, the three extendible dimensions of STEP space have the structure of a three-dimensional torus (*3-torus*).

Additional wrapped-around dimensions, called *internal* dimensions, are possible, but the number and extension of the internal dimensions are subject to certain implementation-dependent tradeoffs, since these internal dimensions are assembled out of the finite resources contained *within* a module.[1]

## 2.2 Updating

The entire array of STEP sites (or a selected subarray if desired) is made to evolve in time by repeated applications of a basic procedure, called an *updating pass*, conceptually consisting of two stages:

- *Data transport.* Each of the sixteen layers is independently shifted as a whole by an arbitrary number of positions in an arbitrary direction. We still end up with a pile of bits at each site, but with a new make-up.

- *Data interaction.* We now take each pile of bits and we send it to a 16-input, 16-output lookup table; the lookup table returns a new pile, which we put in place of the original one.

Two remarks are in order. At the data-transport stage, the shift performed on each layer is a *uniform* and *data-blind* operation (each bit is moved by a fixed offset, independently of its position and its value); in a suitable implementation, this can be achieved by simply shifting the frame of reference (i.e., incrementing a pointer) rather than moving the data themselves. At the data-interaction stage, each site is processed independently: sites do not need to look at one another, and the *order* in which they are processed is irrelevant. Thus, the implementation is free to use any combination of serial an parallel processing.

Most of the rest of this manual can be thought of as a description of how the above conceptual updating scheme is implemented by the STEP architecture.

[1] Basically, the volume of the hyperspace spanned by the internal dimensions cannot exceed the site capacity of a STEP module.

## 2.3 Sectors, modules

Functionally, a STEP unit consists of a number of identical *modules* interconnected as a three-dimensional array (cf. Fig. 2.2). Each dimension is wrapped around on itself (for example, in Fig. 2.2, the side facing in the positive $x$ direction joins the side facing in the negative $x$ direction), yielding the structure of a three-dimensional torus (the canonical way to achieve this wraparound is discussed in Chapter 2.5).

Each module is responsible for storing and updating a portion, called a *sector*, of the entire array of sites. The number of sites in a sector is a fixed power of two ($2^n$, where $n$ equals 22 or 24 in the present implementation, depending on the size of the DRAM chips used), while there is no design limit on the number of sectors (in practice, a limit is set by economic factors sooner than by engineering constraints).

The site address consists of two parts: the first tells where in a sector a site lies, while the second identifies the sector itself (cf. Fig. 2.3). When dealing with an individual module, there will be no danger of confusion if we use the term 'site address' just for the first part, i.e., for the address within the sector served by the module; similarly, the meaning of 'layer' will be restricted to that portion of a layer which is contained within the sector.

## 2.4 Dimensions

For definiteness, let us consider a STEP unit consisting of $2^6$=64 modules physically arranged in a 4×4×4 array, using 16-Mbit DRAM chips ($2^{24}$ bits) to store the layers within each module.

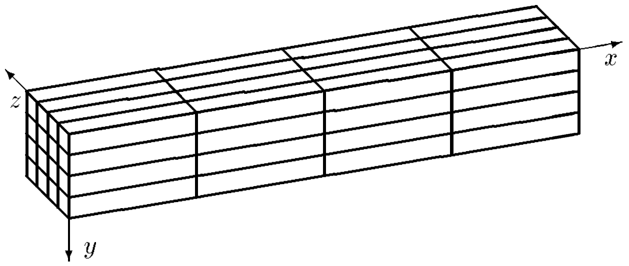



Figure 2.2: Sixty-four sectors (each realized by one STEP module) arranged in a 4×4×4 array. Each sector has been configured as a three-dimensional array of 1024×128×128 sites, with the longer side along the $x$ dimension.

Suppose further that each sector of $2^{24}$ sites has been configured, as will be explained in a moment, as a parallelepiped of 1024×128×128 sites ($2^{10}\times2^7\times2^7$). In this situation, illustrated in Fig. 2.2, a full site address will consist of a string of 24+6 bits having the format of Fig. 2.3, where address bits are grouped and labeled $x$, $y$, and $z$ according to the dimension they refer to—with $x'$, $y'$, and $z'$ denoting those portions of the site address that spill over the sector address. Thus, the total size of the site array will be 4096 (1024×4) in the $x$ dimension and 512 (128×4) in both the $y$ and $z$ dimensions, for a total of 1 Gigasite.

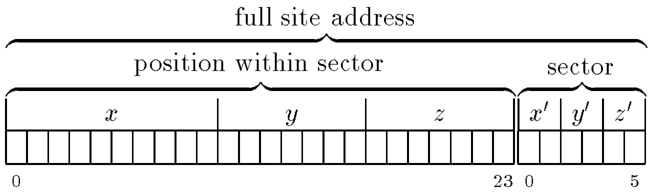



Figure 2.3: Representation of full site address. The 24+6 address bits have been grouped according to whether they encode $x$, $y$, or $z$ coordinates. Primed letters denote bits devoted to the sector address.

How does one establish the number of dimensions and the number of address bits devoted to each dimension?

The structure of the sector address (right portion of Fig. 2.3) is essentially determined by how many modules are present and how they are wired together.[2] Moreover, the typical mode of operation for STEP is for all sectors to operate in parallel carrying out identical orders; clearly, when programming the parameters of a such a mode of operation, the sector address is not a relevant variable.

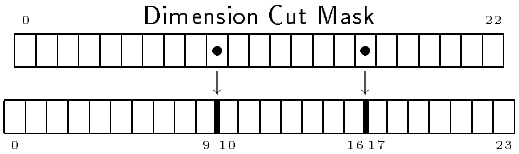



Figure 2.4: Each '•', representing a '1' in the Dimension Cut Mask, introduces a cut (indicated by a thick band) in the corresponding position of the site address. The resulting slices of the site address represent separate components (one for each dimension) of a site's spatial coordinates.

For the above reasons, here we shall restrict our attention to the site address within a sector. Since this is a 24-bit string, there are 23 points where one can "cut" it. By actually cutting the string at two of these places, as in Fig. 2.4, one obtains three separate strings corresponding to the three address slices denoted by $x$, $y$, and $z$ in Fig. 2.3. The cut points are specified by a 23-bit[3] Dimension Cut Mask (§6.19); each '1' in this register specifies a cut at the correponding position. Each of the resulting three slices will be treated by the STEP machinery

[2] Quite a bit of flexibility still remains even at this stage, since the fixed interconnecting wires can be used in different ways to achieve different configurations, as discussed in §??.

[3] This register actually consists of 24 bits, but the last bit is not used.

as a separate component of a site's spatial coordinates, corresponding to a separate dimension.

More generally, by cutting the site-address string (and other registers of the same format associated with address arithmetic) at $d-1$ two places, one obtains $d$ address *slices* of various lengths (the sum of these lengths will, of course, total 24). The actual effect of the "cutting" is that arithmetic carries and borrows are prevented from propagating across the cutpoints, thus making address-arithmetic operations performed on one slice independent of those performed on the adjacent slices.

In this way, the $2^n$ sites of a sector are formally arranged into a $d$-dimensional parallelepiped of sides $2^{n_1}$, $2^{n_2}$, $\ldots$, $2^{n_d}$, where $n_1, \ldots, n_d$ are the lengths of the $d$ site-address slices.

In the following figures, it will not be necessary to explicitly show the Dimension Cut Mask; the position of the cuts will simply be indicated by dark bands as in Fig. 2.4. Shifting the positions of the cuts from (a) to (b) to (c) in Fig. 2.5 changes the aspect ratio of a sector from $4{:}\frac{1}{2}{:}\frac{1}{2}$ to 1:1:1 to $1{:}2{:}\frac{1}{2}$, as in (a), (b), and (c) of Fig. 2.6.

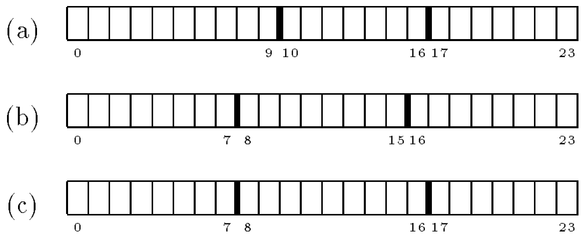



Figure 2.5: Different sets of cuts in the site-address string, as in (a), (b), and (c), yield different aspect ratios for a sector, as in Fig. 2.6.

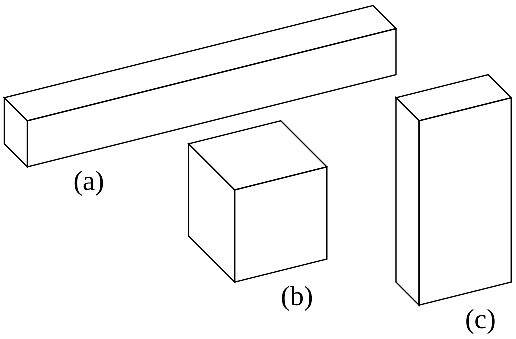



Figure 2.6: Different sector aspect ratios, namely $4{:}\frac{1}{2}{:}\frac{1}{2}$ in (a), 1:1:1 in (b), and $1{:}2{:}\frac{1}{2}$ in (c), arise from different sets of cuts in the site-address string (cf. Fig. 2.5).

As explained above, the Dimension Cut Mask determines the number of dimensions of a sector and their aspect ratio, but does not specify which (if any) of these dimensions should be associated with the $x$, $y$, and $z$ axes along which the sectors themselves are arranged in STEP space. This is done by the three Dimension Cut Pointers (XDCP, YDCP, and ZDCP; §6.19), each of which points to the high end (i.e., the carry position out of the most significant bit) of one of the above dimension slices—as indicated in Fig. 2.7.

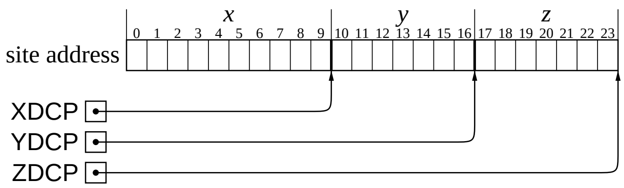



Figure 2.7: The three Dimension Cut Pointers tell which dimension slices are to be associated with the array axes $x$, $y$, or $z$. Any dimension thus marked becomes an *extendible* dimension: the dimension marked $x$, say, is continued from one sector to the next along the $x$ axis instead of wrapping around within each sector. The present arrangement of pointers corresponds to the situation depicted in Fig. 2.2 and Fig. 2.3).

In Fig. 2.8 we illustrate a more general situation. The site-address string is cut into five slices by the Dimension Cut Mask. Of these, three are identified as extendible dimensions $x$, $y$, and $z$ by the Dimension Cut Pointers (note that the order is different from Fig. 2.7). The remaining two slices, labeled $u$ and $v$, represent the address components for additional *internal* dimensions—which wrap around within each sector rather than extend from one sector to the next.

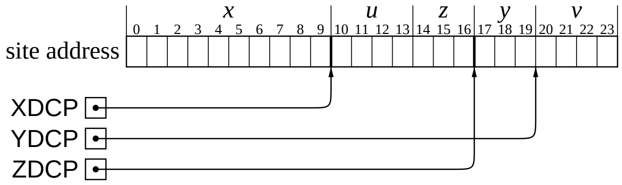



Figure 2.8: Here the site address is partitioned into five slices by the Dimension Cut Mask. Of these, three have been associated with the $x$, $y$, and $z$ axes by the Dimension Cut Pointers; the remaining two slices, labeled $u$ (address bits 10–13) and $v$ (address bits 20–23) correspond to nonextendible or *internal* dimensions.

With reference to Fig. 2.2, where all three extendible dimensions are used, each sector is in contact with the six adjacent sectors through its six faces, which will be labeled in an obvious way $x_-$, $x_+$, $\ldots$, $z_-$, $z_+$, as in Fig. 2.9. (Note that sectors wrap around in each dimension, so that no faces remain exposed.)

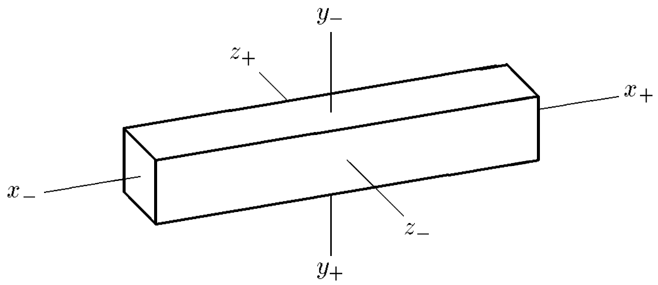



Figure 2.9: Labeling of a sector's faces, with respect to the orientation of the coordinate axes $x$, $y$, and $z$ of STEP space (cf. Fig. 2.2).

## 2.5 Block structure

In the STEP architecture, the minimal functional unit is the individual module: realizations of arbitrary size can be built by simple iteration of this basic unit.

However, for reasons of concrete interconnection geometry it is desirable to introduce a larger "prefabricated" structure, called a *block*, consisting of eight sectors that stand in a distinguished geometrical relationship with one another within STEP space. This logical grouping of sectors is reflected in the implementation, where the eight modules that realize the eight sectors of a block are housed in the same physical *box* (cf. §1.1 and §7.8).

We have seen (§2.1) that the topology of the three extendible dimensions of STEP space is that of a 3-torus. The block structure discussed here provides a systematic way to embed the 3-torus into ordinary *3-space* (three-dimensional space) in such a way that

- Boxes can be stacked in a straightforward three-dimensional array (no circular arrangement of boxes to make opposite ends of the array meet).
- No matter what size the array, modules that realize logically contiguous sectors are found in adjacent boxes. This arrangement avoids size-dependent propagation delays.

### 2.5.1 Glue ports

As mentioned in §2.3, each STEP module is responsible for storing and updating the dynamical variables associated with a *sector*, i.e., a parallelepiped-shaped portion of simulation space. A sector exchanges information with contiguous sectors, along the three extendible dimensions $x$, $y$, and $z$, through its six faces—labeled $x_-$, $x_+$, ..., $z_-$, $z_+$ as in Fig. 2.9.

Since a module scans the sites of its sector *sequentially* (cf. §2.7), data communication between contiguous modules is handled serially by serial *glue ports*, using a modest number of pins. For the purposes of the present chapter, a glue port's detailed structure and the direction of data flow (in or out of a module) on a particular port line at a given moment are irrelevant; all that matters is that the glue ports handle the flow of information in and out of a sector's faces.

The assignment of a specific glue port to a specific face is software-reconfigurable. Here, by 'glue port $x_-$', say, we shall mean whatever port has been assigned to face $x_-$.

### 2.5.2 Wraparound

In some applications, the simulation volume represented by an array of STEP sectors arranged as in Fig. 2.2 will correspond to a similarly shaped volume of the system under study. The exposed faces which delimit this volume represent the interface between the simulated portion of the system and the rest of the system which surrounds it; the latter is not explicitly modeled within the STEP machine. In these circumstances, it is up to the experimenter to feed appropriate boundary conditions (e.g., constant, random) to the exposed faces so as to approximate the influence of the surroundings.[4]

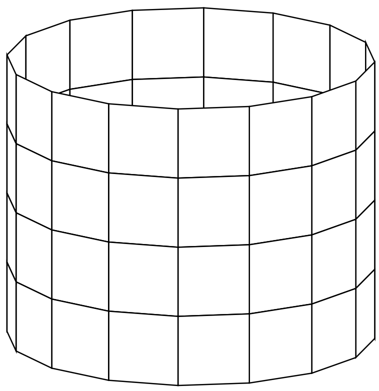


Figure 2.10: Two-dimensional array wrapped around in one dimension (*cylinder*).

However, in most applications (and chiefly in those of a more conceptual nature) the experimenter will prefer to model a self-contained system having no distinguished boundaries. To this purpose, one can bend the STEP-space volume back onto itself, directly connecting each exposed face to the corresponding one on the opposite side.[5] Fig. 2.10 shows a two-dimensional array wrapped around in one dimension (cylinder), while Fig. 2.11 shows a two-dimensional array wrapped around in both dimensions (torus).

[4] This can be done in a variety of ways, as explained elsewhere.
[5] Physicist often use the term 'periodic boundary conditions' for such an arrangement.

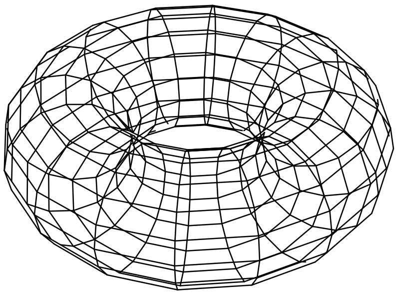


Figure 2.11: Two-dimensional array wrapped around in two dimensions (*torus*).

With STEP, the most demanding situation is when a three-dimensional array as that of Fig. 2.2 is to be wrapped around in all three dimensions, yielding a *3-torus*. That is, how is this functional wraparound going to be achieved physically?

A simple way is to arrange the modules in straight rows as in Fig. 2.2, and join ports at opposite ends with long "return" cables, as shown in Fig. 2.12 for one dimension.

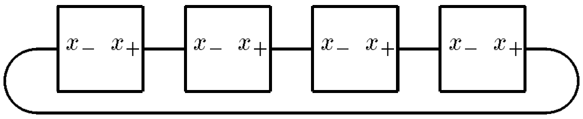



Figure 2.12: Wraparound of modules, in one dimension.

However, as the number of modules in a row increases, such an arrangement would lead to proportionally longer propagation delays, and the clock rate would have to be slowed down correspondingly.

## 2.6 Blocks and strands

Let us consider now the *interleaved* arrangement of Fig. 2.13, where both legs of a round trip are evenly strewn with modules. In this arrangement the physical distance between logically contiguous sectors is somewhat greater than in Fig. 2.12, but the need for a return wire of unbounded length is eliminated.

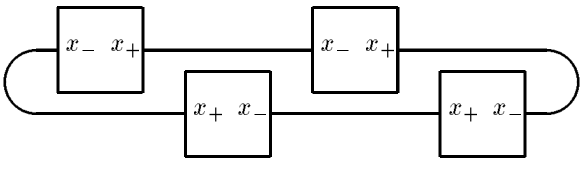



Figure 2.13: Interleaved wraparound, in one dimension.

Fig. 2.13 suggests that in the interleaved arrangements individual modules may be left in essentially the same physical position as in Fig. 2.12 (and, in three dimensions, as in Fig. 2.2); only their interconnection is modified. While the new interconnection may entail a longer propagation delay, the extra delay is a *fixed* one, independent of the number of modules in the chain.[6]

The interleaved arrangement of Fig. 2.13 consists of two parallel *strands* of modules, one with the $x_+$ ports facing rightward and one leftward (and conversely for $x_-$). With this arrangement, the minimal repetitive unit is no longer the individual module, but a *block* of two modules—one from each strand—as shown in Fig. 2.14. Though in close physical proximity, the two modules of a block are not connected to one another within the block itself.

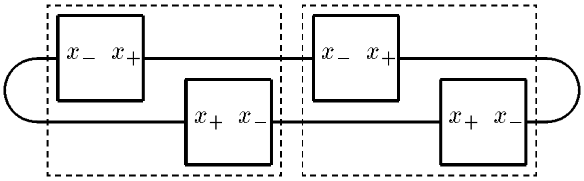



Figure 2.14: With interleaved wraparound, the minimal repetitive unit is a two-module *block*.

In STEP, the above doubling up and interleaving of module chains is carried out for all three dimensions, leading to a minimal repetitive unit consisting of an eight-module block. A block can be thought of as a sector-size "window" on the eight three-dimensional strands that run parallel to one another when the 3-torus is "squashed" onto ordinary 3-space. Figures 2.15 and 2.16 illustrate the analogous situation in two dimensions.

Note that, just as in the one-dimensional case there are two separate strands of one-dimensional space that run parallel to one another but with *opposite orientations* within a block (cf. Fig. 2.14), and in the two-dimensional case there are four separate strands of two-dimensional space (Fig. 2.16), thus in three dimensions each block is traversed by *eight* separate strands of 3-space, one for each of the eight possible orientations.

These eight strand orientations (and thus the eight sectors within a block) will be labeled by three binary parameters, $s_x$, $s_y$, and $s_z$, defined as follows. For a given strand, we'll let $s_x$=0 if the $x_+$ ports of that strand face in the positive $x$ direction of physical space, and $s_x$=1 if they face in the opposite direction (cf. Fig. 2.14); and similarly for $s_y$ and $s_z$. To each of the eight strands there will correspond a different value of the triplet $\langle s_x, s_y, s_z\rangle$, which can also be interpreted in an obvious way as an

[6] A closer packing of modules in the given dimension is of course possible, but that only at the cost of building up thickness—and thus propagation delays—in another dimension.

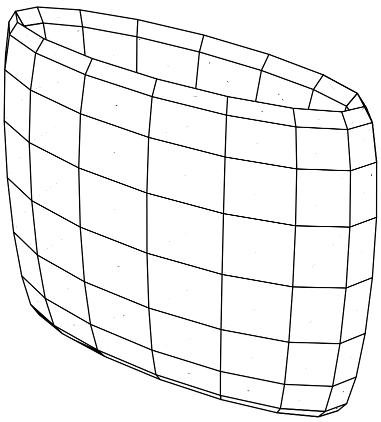


Figure 2.15: A two-dimensional torus, which is usually visualized as embedded in three-dimensions, here is shown in the process of being "squashed" onto two-dimensions. Like a folded handkerchief, the resulting structure has a "thickness" of four two-dimensional layers, or *strands*.

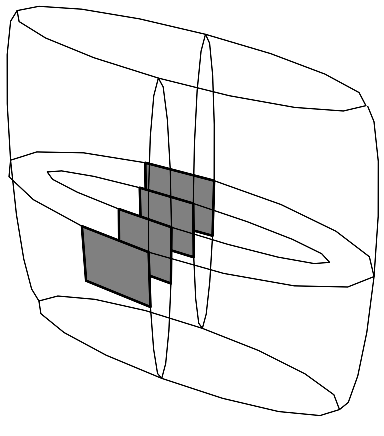


Figure 2.16: In the squashed torus of Fig. 2.15, a *block* is a sector-size window cutting through all four two-dimensional strands.

integer $S$ in the range 0–7, as in the following table

| $s_x s_y s_z$ | $S$ |
|---|---|
| 000 | 0 |
| 100 | 1 |
| 010 | 2 |
| 110 | 3 |
| 001 | 4 |
| 101 | 5 |
| 011 | 6 |
| 111 | 7 |

. (2.1)

To each of the eight strands the block contributes a six-ported module, for a total of 8 modules and 48 ports in each block, as detailed in the following table (2.2)

| *strand* | | *ports* | | |
|---|---|---|---|---|
| $s_x s_y s_z$ | $S$ | $x_- x_+$ | $y_- y_+$ | $z_- z_+$ |
| 000 | 0 | • • | • • | • • |
| 100 | 1 | • • | • • | • • |
| 010 | 2 | • • | • • | • • |
| 110 | 3 | • • | • • | • • |
| 001 | 4 | • • | • • | • • |
| 101 | 5 | • • | • • | • • |
| 011 | 6 | • • | • • | • • |
| 111 | 7 | • • | • • | • • |

(2.2)

Each of a blocks's 48 glue ports, represented here by black dots, is uniquely identified by a face label ($x_-$, $x_+$, ..., $z_+$) and a strand label ($S = 0, \ldots, 7$). It is possible to physically arrange a block's glue ports so that adjacent blocks directly mate both mechanically and electrically.

Block wiring (between blocks and within blocks) is discussed in §7.8.

## 2.7 Scanning

While the STEP modules operate in parallel (and it is in this sense that STEP is a parallel processor), sites within each sector are accessed *serially*. Here we are concerned with the scanning mechanism that determines which sites are accessed and in what order.

Scanning is concerned with a site's *address*—rather than its contents and what is done with it. Thus, though the following considerations are coached in terms of site-updating—where the mode of access is *read-modify-write*, they apply equally well to the *read-only* and *write-only* access modes, which are used in a variety of situations involving display, analysis, and input/output of site data.

In the most basic mode of operation of STEP, an identical updating procedure is used for each sector, so that there is complete symmetry between the sectors. The procedure itself is applied in turn to every site of the sector.[7] (or of whatever portion of the sector one wishes to restrict one's attention; cf. below). Thus, within each sector certain sites are updated before others. However, the functional symmetry between the sites of a sector is not destroyed by this ordering, because the computational model underlying the STEP architecture (§??) is such that the results of updating are *independent* of the actual sequential order of site access. This invariance leaves one free to manipulate the order of access for other purposes, such as producing a display of a given format (e.g., quadruple-interlaced) or accessing sites in the most convenient order for a certain type of real-time analysis.

Many variations of the above basic mode are possible. In some situations, one may want to restrict the updating activity within each module to just a fraction of a sector, so as to have more frequent updates. In other situations, a sector is divided into a number of subsectors, and the parameters for the updating procedure are changed from one subsector to the next (e.g., one may use different lookup tables).

From the host's viewpoint, the primitive hardware function used by STEP for sequentially accessing sites is called a *scan*; this is an indivisible, uninterruptible sequence of site-accesses, autonomously performed and timed by the module.[8] After setting the desired scan parameters within a given module the host *schedules* a scan; this scan will be started by the module at the earliest appropriate moment and will proceed to completion without host assistance (§6.1). The module will then notify the host that the scan machinery is available for a new scan.

Fig. 2.17 gives an overall view of the scan generator. Details will be expanded in the following sections.

The prime mover of the entire address-generation chain is the Scan Index Register (§6.14)—a straightforward 24-bit binary counter which during a scan is normally incremented at the system clock rate, thus converting the steady passage of time into an ascending sequence of integers.

Let us for a moment identify the contents of this register, or *scan index*, with the *site address*: as the scan index increments during a scan, the space of all possible $2^{24}$ site addresses within a sector is traversed in the most straightforward fashion. For instance, if the site address has been partitioned into three slices corresponding to dimensions $x$, $y$, and $z$, as in Fig. 2.7, regular incrementing of the scan index would scan this space as if by three nested loops, the $x$ dimension corresponding to the inner loop and being scanned fastest.

In reality, the STEP scan generator provides a programmable conversion stage, called the *scan index permuter*, between the scan index and the site address (Fig. 2.17); by this means, the regular ascending order of the scan index can be transformed into a variety of more complex access patterns at the site-address level (§2.9). Another programmable stage, the *offset adder*, converts the site address into a DRAM *address*, taking into account "kick" information (§2.10)—which is in general different from layer to layer. The DRAM address is the ultimate offspring of the scan generator, and points directly at a physical bit in the DRAM chip.

Besides controlling the DRAM, the scan generator also controls the glue circuitry that regulates the exchange of scanned data between adjacent sectors; in a sense, this section of the scan generator interfaces the "site-within-sector" portion of the site address with the "sector" portion (cf. Fig. 2.3).

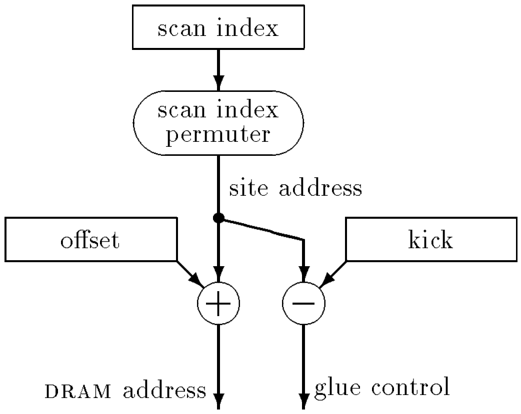



Figure 2.17: Schematic view of the STEP scan generator. Square boxes indicate registers (for instance, 'kick' denotes the contents of the Kick Register); round boxes, combinational logic. Though the scan index increments steadily during a scan, the corresponding site-address sequence may be more complex, since through the permuter the twenty-four bits of the scan index may undergo an arbitrary permutation. By adding a constant to the site address, the $\oplus$ box shifts the spatial origin of a layer, resulting in a virtual shift of the layer's contents. The offset specifies the current value of this constant, while the kick specifies by how much the offset should be modified between one scan and the next. (Not shown in the figure, the kick is subtracted from the offset before the new scan; during the scan itself, both kick and offset remain constant.) Data carry-over from one sector to the next resulting from this offset change is handled by the glue-control machinery, to which the $\ominus$ box supplies the relevant information.

[7] This is done in parallel on all sectors, so that homologous sites in different sectors are affected at the same moment.

[8] Within each module, each of the sixteen layers is served by a separate STEPchip, which contains a full implementation of the scanning mechanism. In principle, while accessing in a given layer the bit located at site address $i$, one may access in another layer the bit located at a different site address $j$. Unless explicitly stated, we shall assume that the same *site* address is generated by all sixteen STEPchips of a module.

## 2.8 Scan index

The scan index steadily increments by one unit at a time during a scan, cycling back to 0 after reaching its maximum value, $2^{24}-1$, or, in hexadecimal, FFFFFF.

In the most basic situation, the scan index will start from 0 and traverse the entire range in a single scan. However, one can request shorter scans, spanning a submultiple (by a power of 2, and boundary-aligned on the same power of 2) of this range. In any case, the next scan will start where the previous one left off (unless, of course, in the intervening time the host writes a different value in the Scan Index Register).

For example, if the starting value is 000100 and the scan length (as determined by the End-of-Scan Pointer, §6.17) is 40, on the next four scans the scan index will step through the following values (columnwise)

$$\begin{array}{cccc}
\texttt{000100} & \texttt{000140} & \texttt{000180} & \texttt{0001C0} \\
\texttt{000101} & \texttt{000141} & \texttt{000181} & \texttt{0001C1} \\
\texttt{000102} & \texttt{000142} & \texttt{000182} & \texttt{0001C2} \\
\vdots & \vdots & \vdots & \vdots \\
\texttt{00013E} & \texttt{00017E} & \texttt{0001BE} & \texttt{0001FE} \\
\texttt{00013F} & \texttt{00017F} & \texttt{0001BF} & \texttt{0001FF}
\end{array}, \tag{2.3}$$

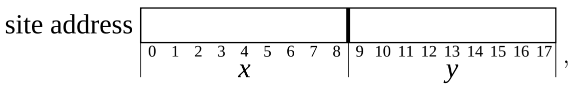
coming to rest on the value of 000200.

## 2.9 Scan permuter

In the most basic situation, the scan index is used directly as a site address, so that the site-address space itself is traversed in a steadily ascending fashion (cf. §2.8). In many circumstances, however, it is desirable to access sites in a more complex order. The *scan index permuter* (§6.15) provides a certain class of scan-index-to-site-address transformations; these include arbitrary permutations of the scan-index bits.

We shall introduce the *scan index permuter* by giving a simple example of its use.

Let us first consider the basic situation. Suppose we have a system of 256K sites arranged as a two-dimensional array of 512×512, or $2^9 \times 2^9$. As shown in the following diagram




the low-order 9 bits of the site address span the $x$ dimension; the following 9 bits span the $y$ dimension; the remaining 6 bits are not used (and are not shown in the diagram).

A STEP module is able to update an array of this size in less than 1/60 sec. We shall arrange for the updating scan to match the format and timing requirements of a VGA-like CRT monitor (namely, a horizontal rate rate of about 31 KHz and a vertical rate of about 60 Hz), so that the evolution of the system can be displayed in real time on the monitor, formatted as in Fig. 2.18, without the need for a frame buffer.

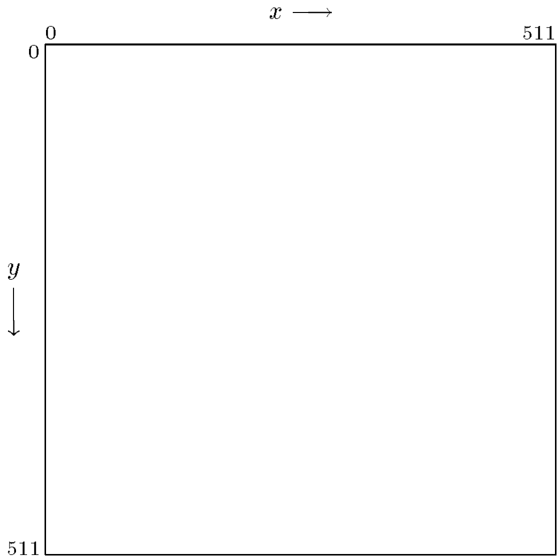



Figure 2.18: A 512×512 array can be updated at 60 frames/sec and displayed in real time on a VGA-like CRT monitor. The corresponding arrangement of the scan permuter is given in Fig. 2.19.

Since the scan index is incremented sequentially, so that higher order bits correspond to coarser time divisions, the low-order 9-bit slice of the scan index, labeled $h$ in Fig. 2.19, will be associated with the monitor's horizontal sweep, which takes place at the faster rate. In other words, as $h$ increments from 0 to 511, the CRT beam will sweep a horizontal line, displaying 512 pixels corresponding to the contents of whatever sites are traversed by the site address during that time. Since we want a horizontal sweep to correspond to the array's $x$ dimension, the $x$ slice of the site address will be derived directly from the $h$ slice of the scan index, as shown in Fig. 2.19.

Similarly, the $y$ slice of the site address will be derived directly from the $v$ slice of the scan index, so that the $y$ address will steadily increase with time as the CRT beam performs a vertical sweep.

Of course, in order to display a complete picture, incrementing of the scan index will be suspended during horizontal and vertical retraces (cf. Scan Sync Mode in §6.1).

Let us consider now a more complex scanning situation. We have a system of 256K sites as in the previous example, but consisting of a three-dimensional array of 64×64×64, or $2^6 \times 2^6 \times 2^6$. As shown in the following

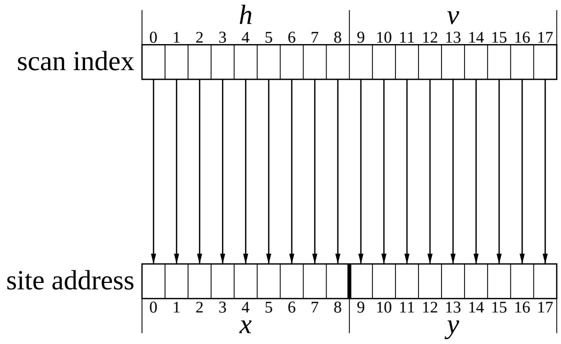
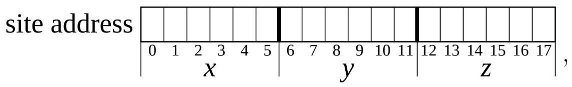



Figure 2.19: When the bits that make up the scan index are used in their natural order to make up the site address, the least significant bits of the site address (those associated with the $x$ dimension, in this case) are incremented at the fastest rate. The label $h$ denotes that section of the scan index that cycles at a rate compatible with a CRT's horizontal sweep, while $v$ corresponds to a vertical sweep. The resulting CRT display format is shown in Fig. 2.18.

diagram




the site address consists of three 6-bit slices, corresponding to the $x$, $y$, and $z$ dimensions (the 6 unused bits are not shown in the diagram).

One way to display this system on a CRT monitor is to show 64 two-dimensional slices, arranged side-by-side on an 8×8 array as in Fig. 2.20.

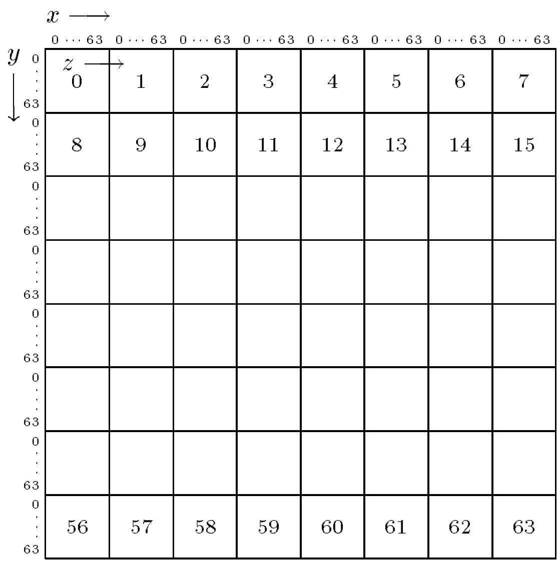



Figure 2.20: A 64×64×64 array can be updated at 60 frames/sec and displayed in real time on a VGA-like CRT monitor by showing 64 two-dimensional slices arranged side-by-side on an 8×8 array. The corresponding arrangement of the scan permuter is given in Fig. 2.21.

In order to produce this arrangement on the CRT screen,[9] sites have to be scanned in nested loops as follows (starting with the innermost loop)

| loop length | slice width | site-address slice |
|---|---|---|
| 64 | 6 | $x$ |
| 8 | 3 | $z$ (lower half) |
| 64 | 6 | $y$ |
| 8 | 3 | $z$ (upper half) |

. (2.4)

This scanning order is achieved by permuting the scan index lines so that the low three bits of the site-address $z$ slice are driven by the high three bits of the scan-index $h$ slice, and the high three bits of the $z$ slice by the high three bits of the $v$ slice, as shown in Fig. 2.21.

[9] Note that the least significant half of the $z$ address increments at a faster rate than $y$, while the most significant half increments at a slower rate than $y$.

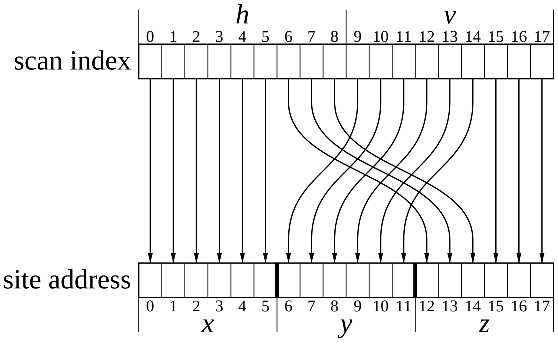



Figure 2.21: Here some of the scan-index bits appear in the site address in a permuted order. Namely, the coarser portions of the $h$ and $v$ sections control the scanning of the $z$ dimension. The resulting CRT display format is shown in Fig. 2.20.

In general, the scan index permuter is programmed by specifying, as a source for each of the 24 site-address bits, any of the 24 scan-index bits or the contants 0 and 1 (cf. §??). The transformations thus obtainable include all the *combinations with repetition* of the 24 scan-index bits—of which the permutations form a subset. In practice, however, permutations are all that we are generally interested in.

The use of constants as sources of site-address bits during a scan is illustrated by the following example. Starting from the situation of Fig. 2.18 and Fig. 2.19, suppose we want to display on the CRT screen a *magnified* view of just a portion of the system. By modifying the scan index permuter arrangement as in Fig. 2.22, each site of the array will be displayed four times, on four consecutive pixels (since the two least significant bits of the scan index are ignored), and similarly each row will be scanned four times, thus giving a linear magnification of 4×.

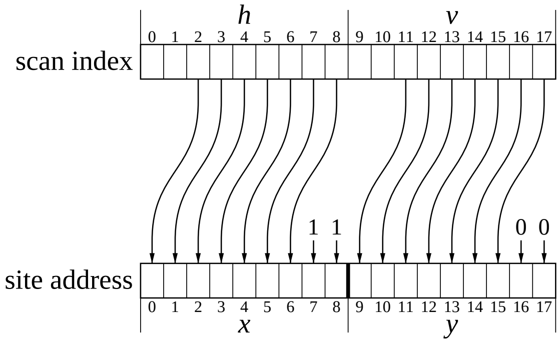



Figure 2.22: By ignoring the two least significant bits of the scan index, the same site will reappear on four consecutive pixels, yielding a linear magnification of 4×; similarly, by ignoring two higher bits, the same row of the array will reappear on four consecutive scan lines, yielding a vertical magnification of 4×. The constants on positions 7–8 and 16–17 select which portion of the array will be displayed in this magnified fashion.

The values of the constant site-address bits determine *which* columns and rows are repeatedly scanned. With the values shown in Fig. 2.22 on positions 7–8 and 16–17, the magnified display will show the upper-right portion (128×128 sites) of the frame of Fig. 2.18. Note that no updating must be taking places during the magnified display.

## 2.10 Kicking

*Kicking* is the technique by which the data-transport stage (§2.2) is realized in STEP. In this section we shall restrict our discussion to a simplified situation involving a single module. The *gluing* of modules in a multi-module machine is discussed in §2.11.

As mentioned at the end of §2.7 (also cf. Fig. 2.17), the DRAM address of a state-variable bit (§2.1) is obtained by adding to its site address an *offset* specified by the Offset Register (§6.18); both site address and offset are $d$-dimensional vectors, with the number $d$ of dimensions and the number of bits devoted to each dimension determined by the Dimension Cut Mask, as explained in §2.4.

The offset specifies where the origin of the bit array (site address $\langle 0, 0, \ldots \rangle$) is located in DRAM. The operation of modifying this offset, and thus shifting the origin of the array, is called a *kick*. Kicking, of course, does not by itself physically shift any data; however, it modifies the relative timing with which data bits from different layers are fetched from DRAM during a scan. Bits that on the previous scan were fetched from DRAM at the same time—i.e., bits that belonged to the same "bit pile" (cf. §2.2)—will, because of the change of offset, be fetched at different times on the next scan; on the other hand, certain bits that used to belong to different piles will now be part of the same pile: they will be fetched from DRAM together and will interact with one another by going together go through the lookup table.

Kicking is done by subtracting from the Offset Register (§6.18) a $d$-dimensional vector specified by the Kick Amount field of the Kick Register (§6.2). Further aspects of kicking can only be appreciated in the context of intermodule *gluing*, discussed in the nest section. In particular, the other fields of the Kick Register are only relevant in that context. For more technical details, see also §6.2.

## 2.11 Gluing

By modifying the offset, a kick influences the order in which the bits of a layer are fetched from DRAM. If we restricted our attention to a single, isolated sector, the net effect of a kick would be just a virtual "circular shift" of layer data *within* the sector. Because of address wraparaound, any data that shift out of, say, the right edge of the sector are shifted back into the left edge of the same sector (note that all this happens along each of the $d$ dimensions separately, and typically with a different value of the kick for each layer).

However, sectors are not isolated. After a bit has been shifted out of one edge and before it is injected into the opposite edge, a process called *gluing* switches the bit in question from the control of the given module to that of an adjacent one, so that, in the above example, the bit will be injected into the left edge of the sector *immediately to the right* of the given sector. This happens in a coordinated way on all right/left edges,[10] effectively gluing homologous layers from all the sectors into a single, machine-wide "conveyor belt."

With gluing, some data actually move from one physical place to another, i.e., from a certain DRAM address in one module to the identical DRAM address in an adjacent module. Let us examine the process in more detail, using, for sake of illustration, the case of a one-dimensional array of length 8; the site address within each sector will consist of three bits and will run from 0 through 7.

We shall assume that the lookup table contains the identity function (i.e., the bit coming out of layer $n$ is returned unaltered to layer $n$), so that no interaction between layers takes place. In this way, it will be sufficient to analyze the kicking and gluing of just one layer.[11]

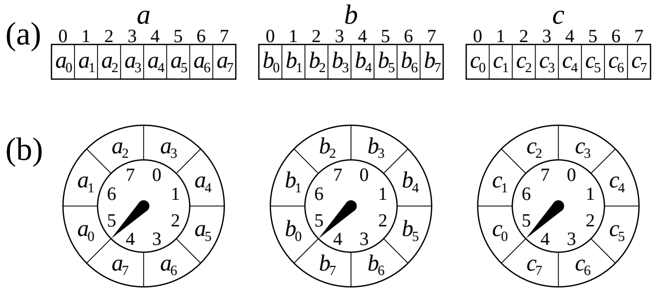


Figure 2.23: (a) Initial contents of three adjacent sectors $a$, $b$, and $c$. (b) Actual layout of the data in DRAM, assuming a value of 5 for the Offset Register; this value is indicated by the position of the dial pointer.

Restricting our consideration to this layer, let us fix our attention on three adjacent sectors, $a$, $b$, and $c$. In Fig. 2.23a, the sites are labeled by the site address (0 through 7); the initial values of the bits contained in the three sectors are denoted by $a_0, \ldots, a_7$, $b_0, \ldots, b_7$, and $c_0, \ldots, c_7$. We want to shift the layer data by 3 positions to the right, with the results shown in Fig. 2.26a.

Now, let us look "under the hood" to see how this is actually done in STEP. In the DRAM that implements sector $b$, the eight bits $b_0, \ldots, b_7$ are found stored consecutively, but not necessarily starting from DRAM address 0. If the current offset is, say, 5, that means that bit $b_0$ happens to be in DRAM address 5, as shown in Fig. 2.23b. Similar considerations apply to sectors $a$ and $c$.

We shall assume that a complete scan of the site array will always run through site addresses 0–7 in that order.[12] In the absence of a kick, the next scan will pick up bits $b_0, \ldots, b_7$ in that order, that is, in the same order as they

[10] Using, as we shall see in a moment, the information contained in the Kick Sign and Kick Max Flag fields of the Kick Register (§6.2).

[11] This data movement in one layer is significant, of course, only relative to the other layers with which it will eventually interact. For simplicity, let us view the movement as taking place relative to an unkicked layer.

[12] If the scan index permuter (§2.9) specifies other than the identity permutation, site addresses may be accessed in a different order, but the overall results would be identical tho those described here.

had been deposited by the previous scan. In order to shift data by 3 positions to the right, we'll give a kick of +3; this kick will be subtracted from the offset, resulting in an offset of 2 (=5−3). In moving from the value of 5 to the value of 2, the offset pointer will "sweep" over a certain fraction of the DRAM circle, as indicated in Fig. 2.24.

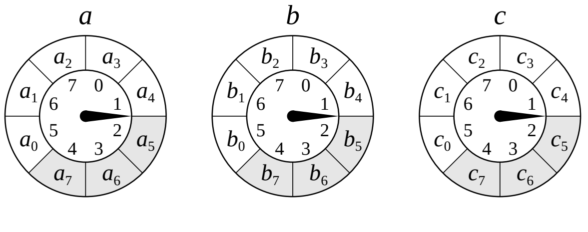



Figure 2.24: In moving from the value 5 to the value 2, the offset pointer swept over the shaded DRAM locations 4, 3, and 2.

The next scan will start picking up data bits from where the offset is pointing now, i.e., DRAM address 2 (Fig. 2.24), and thus will successively get ata bits

$$b_5\ b_6\ b_7\ b_0\ b_1\ b_2\ b_3\ b_4$$

from sector $b$ (cf. Fig. 2.25a), and similarly for $a$ and $c$). During the scan, each bit is eventually deposited back to *some* DRAM in the same order as it was picked up.

*No gluing.* In the absence of gluing, the scan would deposit each bit in the same DRAM, $a$, $b$, or $c$, where it found it, leaving the situation as in Fig. 2.25. Thus, with no gluing all that physically changes is the offset; however the interpretation in terms of site contents is now that of Fig. 2.25a: i.e., the data have been rotated *within* each sector.

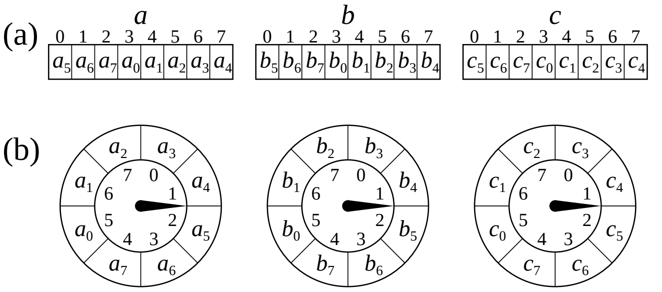



Figure 2.25: Changing the offset without doing any gluing is equivalent to a rotation of data independently within each sector. In this case, a kick of +3 has moved the offset (sector origin in DRAM) three places counterclockwise, equivalent to moving the data three places clockwise.

*Gluing.* Let us go back to the situation of Fig. 2.23, and this time apply the same kick of +3 but *with* gluing. The new scan starts with DRAM and offset as in Fig. 2.25b, and picks up data bits as before, in parallel for the three sectors $a$, $b$, and $c$, in the order

| $a$ | $b$ | $c$ |
|---|---|---|
| $a_5$ | $b_5$ | $c_5$ |
| $a_6$ | $b_6$ | $c_6$ |
| $a_7$ | $b_7$ | $c_7$ |
| $a_0$ | $b_0$ | $c_0$ |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_3$ | $b_3$ | $c_3$ |
| $a_4$ | $b_4$ | $c_4$ |

And now the critical step. Before depositing a bit back to DRAM (in the same DRAM address where it came from), if the bit came from the *shaded* area in Fig. 2.24 we will hand it to the sector just to the *right*, and we will use in its place the homologous bit that is handed to us at the same moment by the sector just to the left, as follows

| $a$ | $b$ | $c$ | |
|---|---|---|---|
| | $a_5$ | $b_5$ | $c_5$ |
| | $a_6$ | $b_6$ | $c_6$ |
| | $a_7$ | $b_7$ | $c_7$ |
| $a_0$ | $b_0$ | $c_0$ | |
| $a_1$ | $b_1$ | $c_1$ | |
| $a_2$ | $b_2$ | $c_2$ | |
| $a_3$ | $b_3$ | $c_3$ | |
| $a_4$ | $b_4$ | $c_4$ | |

For example, just as module $b$ hands bit $b_5$ to module $c$, it receives bits $a_5$ from module $a$, and that is the bit that gets deposited in DRAM address 2 in place of bit $b_5$. Thus, while bits $b_0$–$b_4$ go back to their original places, bits $b_5$–$b_7$ will end up in sector $c$, where they will take the place of bits $c_5$–$c_7$, while their place will be taken by bits $a_5$–$a_7$ from sector $a$, as shown in Fig. ??

In conclusion, bits coming from the shaded area of Fig. 2.24 move one sector to the right before going back to DRAM, while bits coming from the clear area remain in their sector; in either case, the DRAM address of each bit remains unchanged.

*Negative kick.* Let us consider now a kick of −5 rather than +3. Since the sector is of size 8, arithmetic is done modulo-8. The Kick Amount (cf. §6.2) is the same in both cases, nad the offset pointer will land on the same position as in Fig. 2.24. However, the Kick Sign filed of the Kick Register keeps track of the fact that the pointer swept clockwise over DRAM locations 5, 6, 7, 0, 1 (Fig. 2.27), rather than counterclockwise over 4, 3, 2.

As long as no gluing takes place, the results are the same as those of Fig. 2.25. However, with gluing turned on, it is the data in the new shaded areas (Fig. 2.27) that hop from sector to sector during the scan; moreover, as

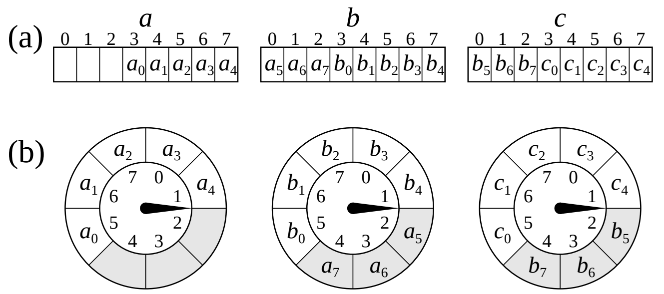



Figure 2.26: (a) Same sector contents as in Fig. 2.23, but shifted three places to the right. (b) This has been achieved by moving the offset pointer three places to the left, sweeping over DRAM locations 4, 3, 2 (shaded areas). During the subsequent scan, the gluing process moved the contents of these three locations in DRAM $a$ to the corresponding locations in DRAM $b$ (and similarly between $b$ and $c$).

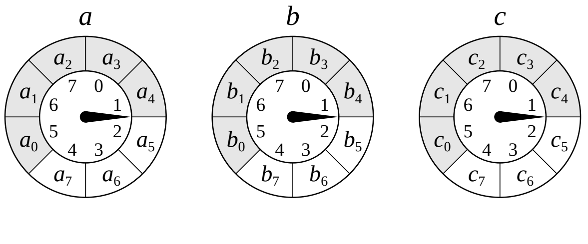



Figure 2.27: The shaded area represents the fraction of DRAM swept over by the offset pointer for a kick of $-5$ (cf. Fig. 2.24).

directed by the Kick Sign, they will be handed by each sector to the sector immediately to the *left*, with the results of Fig. 2.28.

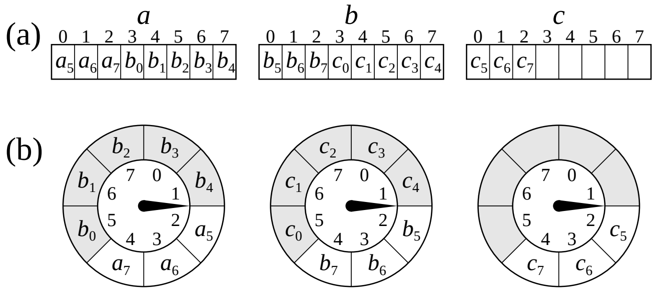



Figure 2.28: The result of a $-5$ kick with gluing, starting from the situation of Fig. 2.23. The shaded area in (b) is complementary to that of Fig. 2.26; bits in the shaded areas move to the sector immediately to the *left* of their sector, and the results in terms of sector contents (b) are a wholesale shift of 5 places to the left.

If $n$ is the number of site address bits assigned to a certain dimension by the Dimension Cut Mask (cf. Fig. 2.4), then any kick value from $2^{-n}$ to $2^{+n}$ is admissible.

The two end-points of this range are only relevant for an extendible dimension, and when gluing is in force. A kick of $2^{-n}$ corresponds to copying the entire sector to the one before it along that dimension; similarly a kick of $2^{+n}$ copies each sector to the one after it. If the dimension is not extendible, or if gluing is turned off, then both these extreme values for the kick coincide with a kick of 0.

The gluing process explained above takes place independently for each of the extendible dimensions.

# 3 The TBus

## 3.1 SBus and TBus; the step controller

The activity of the assembly of modules that make up a STEP machine is governed and monitored by a general-purpose computer called the *host*. The host we are currently using is a SUN SPARCstation—which features a high-performance external bus called the *SBus*.

All exchanges between the host and the STEP machine are mediated by the STEP *controller* (Fig. 3.1), which communicates with the host on the SBus, and drives the modules via the *TBus*—a tree-like bus having the controller at its root and the individual modules at its leaves.

Physically, the STEP controller consists of a small board plugged into one of the SBus slots.

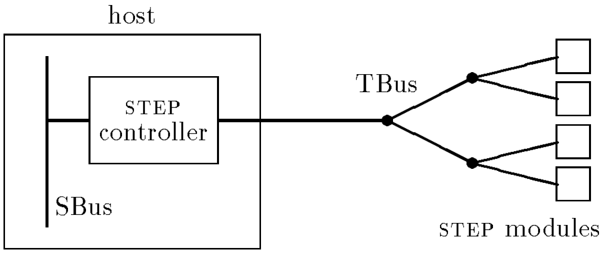



Figure 3.1: The STEP controller drives the assembly of STEP modules via the tree-like TBus; it communicates with the host via the SBus.

## 3.2 TBus usage

Before discussing in more detail the structure of the TBus, it will be useful to give some representative examples of its usage.

*Trigger commands.* The steady-state situation in which STEP requires least attention from the host is when all the structural parameters (number of dimensions, neighbor connectivity, transition function, etc.) remain constant from one updating scan to the next. In this case, all that is needed to keep STEP running at full speed is simply a stream of "Go!" commands (cf. §6.1)—typically at a rate of a few per second—to request a new scan every time that the current one is finished.

Note that, even though new data are generated at a fairly high rate (500 Mbits/sec per module), one does not have to make use of the TBus to display them. Each module's data appear on the module's Video Out lines in the same site-by-site order as they are accessed for updating (§??). If the current experiment uses an appropriate scan format, these data can be directly displayed on a dedicated CRT monitor.

DRAM *readback.* Though direct visual monitoring may be invaluable at an experiment's development stage, in most applications one would eventually want to set up some form of automated data-collection and analysis. Given STEP's unprecedented throughput, the choice is between looking at a lot of data once in a while and looking at a little very often.

By "a lot of data" we mean the entire current state of the system being simulated (or a substantial fraction of it). Reading into the host (via the TBus and the SBus) the state of a single STEP sector takes a time comparable to that needed by STEP to update the entire array.

A much longer time, of course, will be needed for the host to perform even the most trivial form of analysis on the data thus collected. However, while this analysis is taking place within the host, the updating of the STEP array can progress at full speed.

In some circumstances, the monitoring might entail reading back only a small area of DRAM (for example, to check the status of a flag or "indicator light" within the simulated system). *Localized* readback of this kind requires little TBus bandwidth, can be performed between scans without substantially affecting the updating rate, and can be digested by the host in real time. Analogous considerations apply to the *writing* of small amounts of array data, for instance as a way of applying localized stimuli or forcing certain boundary conditions.

*Event-counter readback.* The STEP event counters (§5.2) represent a powerful tool for performing a fairly wide range of data analysis tasks *on line*, within the STEP modules themselves. Suitable tasks for the event counters are detecting the occurrence of specific events, or counting and averaging local quantities. If desired, this counting can be done separately over small "patches" (*coarse-grain averaging*) so as to yield the spatial distribution of the quantities being measured instead of just a system-wide average.

The information abstracted by the event counters about several megabytes of system state reduces to just a few bytes per module. Moreover, the event counters are double-buffered, and their readout is streamlined by a special sequential-read mode (§3.15). Thus, the entire complement of counters can be read out while a new scan is in progress, and in a time much shorter than the scan itself.

*Lookup-table downloading.* Much of the flexibility of STEP derives from the ease with which one can set up time-dependent dynamics, and especially *microprogrammed* dynamics—in which one updating pass actually consists of several scans each associated with a different neighborhood interconnection pattern and governed by a different transition function. This mode of operation leads to the most intensive usage of the TBus: while a scan is in progress, a new set of LUT tables is downloaded via the TBus in preparation for the next scan. Though the amount of data in a module's SRAM lookup table is not much smaller than that in the module's DRAM, typically one and the same table is broadcast to all modules, so that LUT downloading time is independent of system size.

Other uses of the TBus—such as setting up the geometry of the array at the beginning of an experiment, verifying the system's functionality, performing parity or similar checks for the purpose of error detection, and so on—are likely to be of an occasional nature.

## 3.3 TBus structure

The TBus is a tree-like, synchronous, pipelined, bidirectional communication channel having the STEP controller at its root and STEP's individual modules at its leaves.

The special structure of the TBus is dictated by the nature of the STEP architecture. In fact, a STEP machine can in principle consist of an arbitrarily large number of modules; as a consequence, the physical distance between the host and a module—and consequently the transit time of both data and control signals—can be unboundedly large. In order to achieve a modular bus design whose technological parameters (clock rate, drive capability of line drivers, etc.) are independent of the number of STEP modules, an appropriate number of stages of fan-out and of signal regeneration must be provided for.

In the TBus, control and data states are *distributed*, and are propagated by *local* means. This design permits maximum bus throughput at a constant clock rate when both source and destination are capable of sustaining this rate. At the same time, this design makes it possible for the bus to automatically adjust to situations where either source or destination become incapable of sustaining the maximum rate, or where the data direction is changed (*bus turnaround*, §3.9).

What, of course, cannot be avoided is bus *latency*, which is proportional to bus *length* (distance from node to leaves); in turn, bus length is roughly proportional to the third root of the number of modules—since modules will be physically arranged in a three-dimensional array. TBus latency has no effect on STEP throughput when the modules are driven by the host in a "blind reckoning" mode—the most common situation during a simulation run. However, latency becomes important when feedback from the modules is desired. In particular, latency determines the bus turnaround time—during which the bus's effective thoughput drops to zero.

In practice, owing to the third-root dependency on size, TBus latency is negligible (a few clock cycles) up to sizes of a few hundred modules. Even in an extremely large STEP machine (say, 64K modules) the TBus latency will not be greater than a few dozen clock cycles. Moreover, special hardware optimization is provided to avoid frequent bus turnaround in the most common situation requiring feedback from the modules, i.e., readout of the event counters (see §3.15).

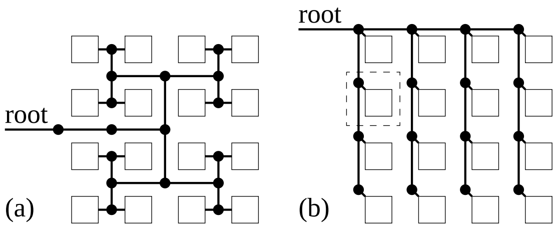



Figure 3.2: (a) Balanced and (b) unbalanced realizations of the TBus. TBus nodes are denoted by dots, STEP modules by solid square. In the current implementation, a module board (dashed box) contains, in addition to a module, also a full TBus node.

The TBus specifications allow much latitude in the actual shape of the TBus tree. The tree sketched in Fig. 3.1 is *balanced*, i.e., all of the branches that start from a given node have identical structure. Indeed, such a balanced tree could be embedded in a regular way in a three-dimensional array of modules, as suggested by Fig. 3.2a (where only two dimensions are shown). On the other hand, the arrangement of Fig. 3.2b, though unbalanced, is based on a one-to-one correspondence between node machinery and module machinery: nodes can be mounted "piggyback," as it were, on the modules themselves (dashed box in Fig. 3.2b). The STEPchip communication protocol is compatible with either type of tree arrangement. What is more, in addition to the circuitry implementing a STEP module, a STEPchip team also contains all the circuitry for optionally implementing a TBus node having the module itself as one leaf. The current prototype model of STEP makes use of this circuitry at the box level: each box acts as TBus node which, in addition to directly serving the eight modules in the box, provides potential TBus branches for up to eight more boxes. Up to three of these branches (i.e., one per dimension) are actually used, as explained in §7.4 (specifically, Fig. 7.4).

One disadvantage of the unbalanced arrangement is that the same command will reach different modules at different moments. If the command is one that requests a scan, where all modules are to operate in lockstep, the modules that are closer to the root will have to insert a calibrated delay (cf. Tree Balancing Delay, §6.20) before starting the scan, so as to start at the same time as the farthest module.

## 3.4 Signal direction

Each segment of the TBus tree consists of sixteen *data* lines and a number of *control* lines (see §3.6 for details). Control lines are unidirectional. Data lines are bidirectional, in the sense that they can carry information in one direction or the other, but not both at the same time (*half-duplex* communication).

For sake of clarity, it will be convenient to introduce the following terminology.

The terms *in* and *out*—as well as *input* and *output*—refer to signals that respectively enter or leave a node. Thus, an output pin is one that drives the associated line, while an input pin is one that senses it. (The same pin may be used as input or output at different moments.)

The *inner* side of a node is the one facing towards the host; the *outer*, towards the modules. The STEP controller acts as a node having only a outer side, and each module as a node having only an inner side.[1]

The terms *inbound* and *outbound* refer to the absolute direction of travel: outbound signals travel from the host towards the modules, and vice versa for inbound signals. Thus, an output pin on the inner side of a node will be driving an inbound signal.

Finally, the terms *upstream* and *downstream* refer to the current direction of travel; upstream is where a signal comes from, downstream, where it goes.

## 3.5 Fan-out and fan-in

The bus nodes (cf. Fig. 3.1) act as repeaters, allowing a signal to propagate over an arbitrary distance; wherever the tree branches out, they also act as *fan-out* or *fan-in* elements—according to the direction of the signal. The following discussion applies to each of the data lines individually.

Outbound data travel along the branches of the tree towards the modules, with a delay of one clock cycle between one node and the next. Fan-out occurs by each node copying its inner data to all its outer data lines, as in Fig. 3.3a. Thus, outbound data eventually reach all the modules; it is the responsibility of each module, as instructed by previous commands, to accept or ignore the broadcast data (§3.13).

[1] As noted in the previous section, each module board also contains the machinery for an additional full node.

Inbound data proceed along tree branches toward the root, with a delay of one clock cycle between one node and the next. Inbound data originate from only one module at a time; previous commands will have specified which module is selected for this purpose at any particular time (§3.13). The unselected modules send 0s as their data.[2] Fan-in of inbound data occurs by each node ORing the data from all outer lines and putting the result on the inner line; since the 0s contributed by unselected modules are ignored by the OR operation, in effect the data coming from the selected module—denoted by $x$ in Fig. 3.3b—are routed unchanged through the node no matter which branch they come from.

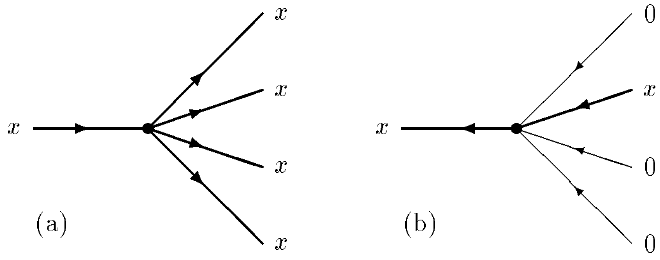



Figure 3.3: Fan-out of outbound data (a), and fan-in of inbound data (b) on the TBus.

Unless otherwise specified, fan-out and fan-in of control signals occur in analogous ways.

## 3.6 TBus signals

Each TBus *segment*—terminated by a node at each end—consists of sixteen *data* lines and six *control* lines, as in Fig. 3.4.
At any node where the TBus branches out into two or more segments, the node's outer side will provide terminations as in Fig. 3.4 for each of these segments. The details of this branching will be discussed in §??; until then, the issue of bus branching can be ignored.

All TBus signals are sensed at the rising edge of a common STEP *clock* which is made available to all STEP modules and TBus nodes, as well as to the STEP controller (§7.3, §7.5).

The direction of the data lines is not directly governed by a "Read/Write" control signal, but is established by a protocol discussed in §3.8 and §??.

The two *handhsake* lines (Outbound Handshake and Inbound Handshake) regulate the rate of flow of data on the data lines, as explained in §3.7.

The Reset signal is originated by the STEP controller, and is used to reset STEP modules and TBus nodes to a

[2] This sending of valid, though dummy, data is done in order to retain data synchronization between all modules; cf. §??.

known state after the occurrence of errors or other unanticipated conditions.

The Sync signal carries short binary-encoded messages ("sync pulses") providing horizontal and vertical sync information for a CRT monitor and related information for the STEP machine itself (see §??).

The Frame signal is used to delimit a TBus frame, specifying which part of it contains address, data, or parity information—as explained in §3.8.

By the Status line, modules or nodes send to the STEP controller real-time status information, namely interrupt requests and "scan in progress" reports (see §?? and §??).

In the current implementation, in which TBus fanout is managed by a module (cf. Fig. 3.2b), the module hitches a ride on the TBus for an additional outbound signal (not indicated in Fig. 3.4), called Scan, which is asserted precisely when the module itself is scanning. Since in most circumstances all STEP modules are supposed to scan in lockstep in spite of any TBus tree unbalance (cf. §3.3), this signal allows modules to verify (by local comparison) that they are indeed scanning in lockstep.

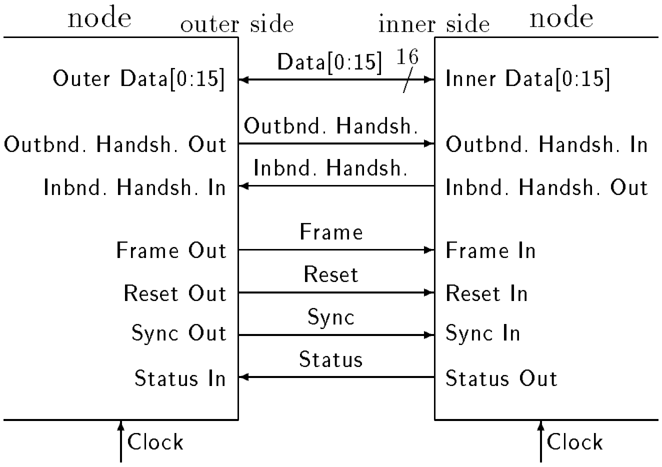



Figure 3.4: Makeup of a TBus segment.

We mentioned that a STEP module acts as a TBus node having only a inner side (since it is located at the end of a branch). Besides one bidirectional data pin, each STEPchip has one pin for each of the TBus control signals (§10.2.6). When sixteen STEP chips are assembled together in a module, their sixteen Tbus Data pins (§10.2.6) provide the terminations Inner Data(0:15) of Fig. 3.4. The sixteen chips' Reset inputs are tied together, so as to present a single Reset In termination to the TBus Reset line—and similarly for the other outbound control signals. The inbound control signals, Inbound Handshake Out and Status Out are supplied to the TBus by just one of the sixteen chips, since all sixteen chips act in lockstep.

All of the TBus signals are *pipelined*, i.e., each node samples a signal coming from upstream at the rising edge of the clock, it logically combines the sampled signal with other sampled signals and internal states, and puts out the result downstream early enough during the clock cycle for this result to reach the next node downstream in time to be sampled by the next rising edge of the clock.

In addition, the data signals are *double-buffered*, as explained in §3.7.

## 3.7 Data flow control

In each TBus segment, the bundle of sixteen data lines is accompanied by two *handshake* lines. The one pointing upstream signals whether a node is *full*—and thus cannot accept the incoming data bit. The one pointing downstream signals whether the node is providing a *valid* data bit or just a dummy bit (whose value is of no relevance); details will be given in a moment. Which line has which role depends on the current data direction; that is, when the data are outbound the Outbound Handshake line (which in this case points downstream, i.e., in the same direction as the data) plays the role of Valid signal, while the Inbound Handshake line (which points upstream) plays the role of Full signal, as in Fig. 3.5. The roles are reversed when the data are inbound, as in Fig. 3.6.

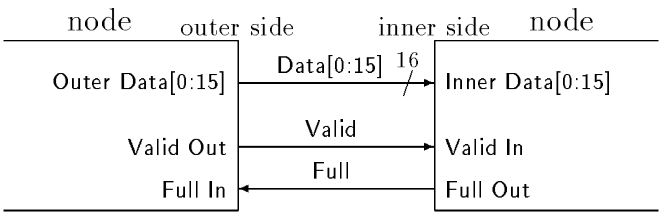



Figure 3.5: When data are outbound, the Outbound Handshake, which flows with the data, plays the role of Valid signal, while the Inbound Handshake, which flows against the data, plays the role of Full signal.

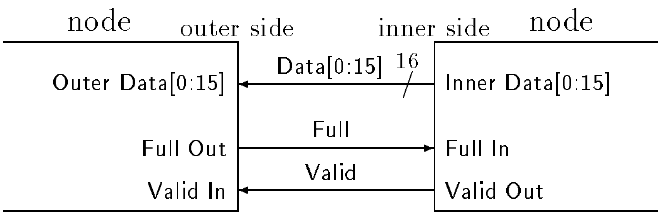



Figure 3.6: When data are inbound, the Inbound Handshake, which flows with the data, plays the role of Valid signal, while the Outbound Handshake, which flows against the data, plays the role of Full signal.

Each of the data lines of the TBus is double-buffered as it passes through a node.[3] The double-buffering mechanism, which is essentially a 2-bit FIFO, allows a node to use the Full line as an "XON/XOFF" signal to regulate the rate of data flow. Let the flow proceed from node $A$ to $B$ and from $B$ to $C$, as in the following diagram

$$A \longrightarrow B \longrightarrow C \quad .$$

Normally node $B$ holds only one bit,[4] $x$, which at the next clock pulse is passed on to $C$ and replaced by the one supplied by $A$. If node $C$ asserts its Full line, then the $x$ bit is retained within $B$; $A$'s bit is still accepted by $B$, which now contains two bits and consequently asserts its Full line. As soon as $C$ is no longer full, bit $x$ is passed on to it and $B$ itself is no longer full.

[3] Traffic on a pipeline has some "inertia," i.e., it cannot be stopped all at once by action-at-a-distance. Since there is one clock cycle of delay from the time a node realizes that traffic downstream is backed up to the time this information is communicated to the node immediately upstream, room must be available to accomodate the bit passed on during that cycle.

[4] This may be a valid bit or a dummy bit: the Valid tag accompanies a data bit throughout its life.

Unlike other FIFO schemes, a TBus node will never contain less that one bit. In fact, the ultimate sources of data bits (the STEP controller for outbound data flow, and the STEP modules for inbound data flow) keep the data pipeline always primed: if at a given clock cycle data could be accepted (i.e., the node immediately downstream is not full) but the source does not have any valid data, a dummy bit will be interpolated. Once thus created, a dummy bit will only be consumed at the ultimate destination.[5] This "incompressibility" of the data stream was deliberately introduced in order to maintain a satisfactory timing relationship beween streams flowing towards the root from different branches, even in the case of an unbalanced tree (cf. §3.3, §??).

## 3.8 TBus frames

Data on the TBus are transmitted in the form of self-contained *frames*, delimited by the Frame signal as in Fig. 3.7. A frame consists of three phases:

- An *address* phase of variable length (possibly of zero length). The string transmitted during this phase specifies the address of one of the STEPchip's registers (see §3.10) and the direction of the requested data transfer (see §?? for details). The address phase is terminated by negating Frame.
- A *data* phase, during which the STEP controller either writes data to that register or reads data from it; the length of this phase depends on the register that is being accessed (cf. §3.10). The data phase is terminated by re-asserting Frame.
- A *parity* phase, of exactly one bit, where the STEP controller transmits the parity of the frame.

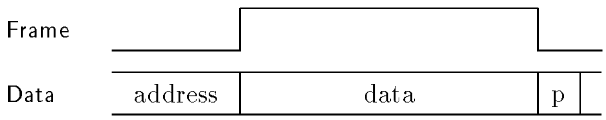



Figure 3.7: Framing of address, data, and parity on the Data lines.

Fig. 3.7 represents the *sequence* of valid bits on a data line, not the actual *timing*; depending on the status of the Valid line (§3.7), dummy bits may appear anywhere between valid bits. Typically, several idle cycles, represented by dummy bits, will elapse between the parity bit at the end of one frame and the first address bit at the beginning of the next frame.

[5] The alternative would have been to allow dummy-bit gaps that may have developed between valid bits to be "reabsorbed" whenever earlier valid data were slowed down.

Note that writing or reading are done in parallel, via the sixteen data lines, to or from all sixteen STEPchips that belong to a module; thus, the data flowing on the data lines during the data phase may (and typically will) be different for each of the sixteen chips. Similarly, sixteen different parity bits, one for each of the sixteen STEPchips of a module, are transmitted in parallel on the data lines.

On the other hand, during the address phase the *same* address and data-direction information is sent on all sixteen data lines (even though in principle it would be possible to send different values), so that homologous registers will be accessed (all read or all written) on all sixteen chips of a module simultaneously.

The entire frame will be called a *write* or a *read* frame according to the direction of data movement (outbound or inbound) for the data phase—which, as we have seen, is specified during the address phase.

## 3.9 Bus turnaround

The default data direction for the TBus is 'write'. The direction turns to 'read' only during the data phase of a read frame.

Data direction changes propagate from the STEP controller toward the modules at the rate of one bus segment per clock cycle, with one exception. If a change from write to read reaches a segment where outbound data are being held up waiting for the Full signal (= Inbound Handshake; cf. §3.7) to be asserted, the propagation of this change will be held up too, so as not to overcome the stretch of write data still in the pipeline.

On the other hand, a change from read to write will propagate at full speed regardless of whether there are inbound data still in the pipeline. These data, together with the information carried by the associated handshake lines (cf. §3.7), will be lost. The handshake lines themselves will immediately adapt to their new role of accompanying outgoing data.

TBus data direction and bus turnaround are discussed in more detail in §??.

## 3.10 stepchip registers

As mentioned in §3.8, the TBus represents the medium through which the STEP controller writes data to or reads data from a number of registers located on the STEPchips (these registers are detailed in §6). During the address phase of a TBus frame, a particular register is addressed and the data direction for the ensuing tranfer is established; during the data phase, data are serially written to this register from a TBus data line, or serially read from this register to the same data line. This process goes on in parallel for the sixteen chips of a module and the corresponding sixteen data lines.

The STEPchip registers come in different lengths. Most registers have a fixed length, ranging from 1 to 120 bits. Two registers, however, namely the LUT I/O Register (§6.13) and the Scan I/O Register (§6.16), are actually serial data ports used to download or upload selected regions of, respectively, the DRAM and SRAM areas of a module. For these two registers, the length of the desired data transfer—which plays the role of register length—must be explicitly programmed by the STEP controller into the chip before the transfer itself takes place.

Many of the STEPchip's registers are logically collections of shorter registers that have been put together under a single register address for convenience and efficiency. For example, the six-bit Event Counter Source Register (sectECSR) consists of (a) a two-bit field selecting one of four possible data sources for the event counter and (b) a four-bit field specifying how this source is to be combined with another fixed source.

For I/O purposes, the STEPchip registers are organized as circular shift-registers[6] with an input/output tap at a distinguished position. A register is written by serially shifting new bits into it via the input tap, while the register's entire contents rotates one bit-position at a time; similarly, a register is read by rotating it entire contents one bit-position at a time, letting its bits serially appear at the output tap.

With this mechanism, it is clear that a read operation must clock out of a register a number of bits equal to the register's length, in order to leave the register in the original alignment at the end of the operation; an equal number of bits must be clocked into the register by a write operation, in order for each bit to have reached its proper destination in the register at the end of the operation.

The length of each register is explicitly known to both the STEPchip and the host. This length is one of the parameters that is passed by the host to the STEP controller in order to set up the transfer of a frame of data, and is used by the STEP controller to know when to terminate the data phase (cf. §3.8) and proceed to the transmission of the parity bit (cf. §3.8 and §3.11).

The *register address space* of the STEPchip consists of 32 locations. Each register is uniquely identified by a five-bit address. In the address phase of a TBus frame, leading zeros of the register address need not be transmitted (that's why this phase has variable length, as noted in §3.8); thus, low addresses require less time to send. Of course, the lowest addresses have been assigned to the most frequently used registers (cf. Chapter 6).

[6] These remarks do not apply to the data ports.

## 3.11 Frame parity

The *parity* of a TBus frame is a single bit obtained by XORing together the frame's address and data bits and complementing the result ("odd parity"). This parity is computed independently by the STEP controller and by the individual STEPchips. At the end of the frame, the STEP controller broadcast its version of the parity bit (Fig. 3.7); it is the responsibility of each STEPchip that was selected for that frame to compare this bit with its own parity bit, and generate a parity interrupt in case of mismatch (§??).

**[Pointer to details of when the parity flags in the step controller and the chips are cleared in preparation for a new frame.]**

The parity bit is handled in a similar way but with a somewhat different meaning in sequential-read transactions (§3.15).

## 3.12 Interrupts

Interrupts generated by a STEP module propagate inwards along the TBus on the Status line, which also carries Scan-in-Progress information (§??). On each module, only one of the sixteen STEPchips senses the incoming Status line, and only one of the STEPchips drives the outgoing Status line. A wire-AND rail, Int, ties together the sixteen chips, allowing them to exchange interrupt information (§??, §??).

Interrupts that are enabled (see IER) will generate a host interrupt when the corresponding error condition is signaled. Once an interrupt reaches the host interface, further interface operation is suspended and the interface waits for the host to restart it.

Errors that are traceable to a specific TBus frame are signaled at the end of that frame; thus Bus Parity (§3.11) (signaled by setting the BPIF) and Unknown Register[7] (URIF) errors are signaled at the moment when the STEPchip checks the parity bit for the frame just completed (see also VWE) **[what is this?]**. Thus, by the time the error is signaled the next instruction is just being fetched by the STEP controller. The erroneous instruction is therefore always the *previous* instruction; a pointer to this instruction is available to the interrupt service routine.

Because of the pipelined nature of the TBus, interrupts generated by STEP take some number of clocks to reach the controller. It is therefore necessary that the STEP controller guarantee a sufficient delay so that a second data transfer over the TBus will not take place before an interrupt from the previous transfer can reach the STEP controller.

Contention errors (two drivers both trying to write onto the same line) are not normally synchronized with a particular data transfer, and so they are signaled as soon as they occur. These are fatal errors: the interrupt routines will not try to recover (as they would from a Bus Parity error, for example). While a given Contention Interrupt Flag is set, the STEPchip should probably turn off all tri-state drivers on the affected wire, in order to avoid hardware damage (at least if the associated enable flag is set). Contention on the TBus is of course a special case, since it could prevent one from reading the IFR. This is not really a problem: if an interrupt occurs but the IFR is read as all zeros, then it must be that the TBus drivers are disabled, and so the interrupt signals STEP Bus Contention.

In this implementation, the STEP controller treats STEP interrupts as being edge-triggered: once a STEP interrupt has been detected, the current instruction decode is aborted, a host interrupt is generated, and the STEP controller goes into a halt ("waiting for host") mode. *No new host interrupt will be generated until the* STEP *interrupt(s) have been cleared.* Thus the host interrupt service routine will be free to interrogate the IFRs, even though the interrupt is still being asserted by STEP.

## 3.13 Module selection

During the data phase of a write frame, data are physically fanned-out by the TBus to *all* STEP modules. In most cases, these data are intended for use by all modules; in certain cases, however, it is desirable that only certain modules be enabled as active listeners, either on an individual basis or as members of a certain group of modules.

During the data phase of a read frame, it is essential for the proper functioning of the fan-in mechanism (cf. §3.5) that only *one* module be enabled as a talker on the TBus.

STEP's module selection mechanism, described here, allows each module to decide whether or not it should take active part in a given write or read transaction. The unit of selection handled by this mechanism is an entire module. Finer-grained selection, such as write-enabling of individual layers within a module, may be achieved by other methods (cf. §6.27).

We have seen that a STEP machine consists of an arbitrary number of modules, and each module consists of sixteen layers each served by a separate STEPchip. In most respects, the sixteen chips of a set act independently

[7] This error occurs if an attempt is made to send data to (or read data from) a non-existent register. Note that data bits written to a nonexistent register are always accepted (and simply discarded) by the STEPchip, and are simply discarded; bits read are all zeros.

of one another; they manage to achieve coordinated action not because they exchange coordinating information between one another, but because they are separately instructed by the host to play in lockstep appropriately correlated roles.

However, the sixteen chips of a module act as a unit for the purpose of deciding whether that module is *selected*, i.e., will accept the data broadcast by the host on the TBus during a write frame, or will put its own data on the TBus during a read frame.

The decision of whether or not a module is currently selected is based upon a match between an *internal name* and a *target name*. Both of these names are sixteen bits long (one bit per chip). For each of the sixteen positions, internal- and target-name bits are (directly or indirectly) specified by a three-bit *selection code*, which is broadcast by the host and written to the Module Select Register (§6.0) of all modules.[8]

There are two primary sources available for the internal-name bit. The *module ID*, stored in the Module ID Register (MIDR) is used to uniquely identify a module, and is assigned once and for all at initialization time (§3.14); the *group ID*, stored in the Group ID Register (GIDR) is used for grouping modules as needed for the convenience of particular operations, and may be reassigned by the host from time to time. Both IDs are sixteen bits long, with one bit contributed by each chip. A third source is the Glue-AND bit, used only at initialization time (§3.14).

For the target-name bit, the selection code may supply directly a 0, a 1, or a 'dont care'; or it may specify that the bit should be taken from the LUT Index Register (§6.11). This fourth option is used during a special *sequential-read* mode (§3.15).

The following table lists what sources the different selection codes specify for the internal bit and the target bit[9]

| selection code (MSR) | bits to be matched | |
|---|---|---|
| | internal | target |
| 000 | MIDR | 0 |
| 010 | MIDR | 1 |
| 100 | MIDR | LIR |
| 110 | Glue-AND | 0 |
| 001 | GIDR | 0 |
| 011 | GIDR | 1 |
| 101 | GIDR | LIR |
| 111 | don't care | |

. (3.1)

[8] This register is exceptional in the sense that data addressed to it are accepted by *every* module regardless of the module's current selection status.

[9] The values 000, 010, and 111, which are probably the most commonly used, happen to be of a form that can be packed as *immediate* data in a step-list frame (cf. §??).

Of course, the individual STEPchip will be able to detect a match between internal name and target name only in one of the sixteeen bit positions. The overall name match is obtained by ANDing together the sixteen partial matches. This logical operation is physically performed by a wired-AND line that ties together all sixteen chips' ModSel pins (§9.5), collecting the arguments of the AND from the individual chips and returning the result to all chips. In this way, the individual chip not only gives its contribution to the selection decision, but is also made aware of the collective decision's outcome.

The above comparison between internal name and target name is made at the end of the data phase of each frame (Fig. 3.7), and the result ('selected' or 'not selected') remains valid for the entire duration of the following frame.

## 3.14 Module ID initialization

As mentioned in §3.13, before a STEP machine can function properly each module must be given a distinct module ID. This initial naming task presents a challenge, because there are no intrinsic features in the architecture that distinguish one module from another. In fact, (a) the modules are structurally identical and are physically interconnected in a uniform way (i.e., they form a three-dimensional torus), and (b) at system reset all modules are set to an identical internal state.

To break this symmetry between modules, it is sufficient to "tag" one module; this is done at power up, as explained in §??. Once the symmetry is broken, every module is in principle uniquely identifiable by its $x$, $y$, and $z$ displacements (in the torus coordinates) relative to the tagged one. This fact is exploited by the module ID initialization procedure described below (which the host will run immediately after power up): starting from the tagged module and taking advantage of the glue lines that connect modules in the $x$, $y$, and $z$ directions, this procedure successively steps from module to module, selecting and uniquely naming each one in turn.

What distinguishes the tagged module at power up is that it it is accessible on the TBus(only the modules in the corner box—cf. §7.1—are initially accessible) and has a 1 rather than a 0 as the least-significant bit of its module ID.

**[Update the following procedure]**

By appropriate use of the 000, 010, and 111 selection codes (table (3.1)), the host will select just this module and give it a name consisting of all 0s (`0000` in hexadecimal notation), and then select all the other modules at once and give them, for the moment, a name consisting of all 1s (hexadecimal **`FFFF`**).

We shall now describe how naming information is propagated through the array of modules by the initialization procedure.

Let us recall that, for each of the sixteen data layers, each module has six glue pins which connect it, via as many glue lines, to the six adjacent modules (Fig. 3.8a).

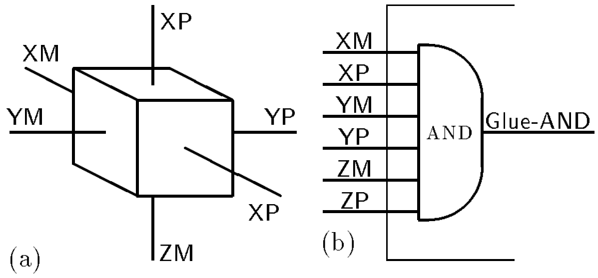



Figure 3.8: (a) The six glue lines that connect one module to its neighboring modules. (b) The module ID initialization sequence makes use of the AND of the six glue lines.

At power-up, the six glue pins of all modules are programmed as inputs; these inputs are internally ANDed together into a signal call Glue-AND (Fig. 3.8b). Since the glue inputs are internally pulled up by resistors to logical 1, Glue-AND will have a value of 1.

We'll now turn the XP pin of module 0000, layer 0, into an output[10] and put a 0 on it. This 0 will travel on a glue line and reach pin XM of an adjacent module, forcing its Glue-AND to 0. Making appropriate use of selection code 110, corresponding to Glue-AND=0 (cf. table (3.1)), we can select just this adjacent module,[11] and write 0001 as its module ID. At this point we'll ask module 0000 to turn pin XP of layer 0 back into an input, and module 0001 to put a 0 on one of its glue pins, so as to "touch" another module. We will name the new module 0002, and so on until all modules of the STEP machine have been given unique IDs.

## 3.15 Sequential read

An optimized, sequential mode of data access is provided for use in reading back event counts. Normally, event-count readback is done using literal data in the selection code to explicitly address specific modules in turn (§3.13).

[10] By appropriately setting the sign of the kick register; cf. §??.

[11] With the present naming procedure, the only two modules having Glue-AND=0 are the module that emits 0 and the one that senses it. But the former will already have been named, while the latter still has the default name FFFF. Since selection code 110 is only used with one of the sixteen chips, the remaining fifteen name positions can use code 010 to discriminate between the emitting module and the receiving module. (This scheme does not allow one to discriminate between the last two modules in a 64K-module machine, but in this case the last module does not have to be named, since it already has the correct name FFFF.

The sequential-read mode makes use of the LUT Index Register (LIR; cf. §6.11). All LIRs initially have the same value, and all are incremented simultaneously to maintain this property. In the sequential read mode, this common value of the LIRs is used to specify the target module for the readback operation.

Within each module, the 16-bit LIR—used as a target name[12]—is compared with the module ID—used as an internal name (cf. §3.14). The comparison is carried out cooperatively by each module's STEPchip team: each STEPchip has a complete copy of the LIR and performs one bit of the comparison. As soon as target name and internal name match, the selected module sequentially places on the data lines event-counter data (least significant bits first) for as many clock cycles as specified by the Event Counter Length (§6.17). After that, all modules increment their LIR in lockstep, and the next selection is performed.

[12] Entries 100 or 101 of (3.1).

# 4 Host and controller

Even though the current implementation of STEP uses a SPARCstation as a host, hardware and software were designed so as to be essentially independent of the nature of the host. The only host-specific component of the system is the host side of the STEP controller, which is matched to the host's bus. The current STEP controller interfaces with the SPARCstation's SBus.

Basically, the STEP controller sequentially interprets a list of intructions found in the host's memory, and translates them into commands to be broadcast on the TBus. Most of these instructions entail moving a block of data from a memory buffer on the host to the STEP array or viceversa.

## 4.1 Host/controller communication

The STEP controller has two separate modes of communication with the SBus, one as a bus *slave* and one as a bus *master*.

The STEP controller contains a number of hardware registers, called *control registers*, which are use to initialize, direct, and monitor its operation. These registers, some parts of which are reod-only, some write-only, and others read/write, are memory mapped on the SBus as four consecutive 32-bit words which are accessed as SBus slaves.

Under certain conditions, writing to the first of these registers (the Next List Pointer) initiates the execution of a step list (§4.3)—an activity that the STEP controller performs as an SBus master.

## 4.2 Shared memory

The step lists and the associated data buffers reside in an area of host memory that is shared beween the host and the controller. Several step lists (with the associated sets of data buffers) may be present in shared memory at the same time, ready to be executed in turn. In the SPARCstation, this shared memory is mapped simultaneously into the kernel's virtual address space (where it is seen by the STEP controller) and into the user's address space (where it is seen by the user's program). All shared memory is allocated in multiples of four 32-bit words (16 bytes).

## 4.3 The step list

The "program" that the STEP controller executes consists of a sequence of instructions, called *step list*, residing in the host's memory.[1] Typically, the step list and the associated data buffers will have been constructed at leisure by a user program running under UNIX, without concern for real-time performance. On the other hand, dependable real-time performance is expected of the STEP controller; to this end, the STEP controller is given *bus master* access to the SBus when interpreting the step list (cf. §4.1).

The data buffers provide designated areas for transfering data to and from the STEP array. A typical data buffer would a LUT buffer, i.e., a memory image of a STEP lookup table (§??).

Each step-list instruction consists of exactly four 32-bit words named as follows

$$\boxed{\begin{array}{l}\text{opcode}\\ \text{buffer pointer}\\ \text{register length}\\ \text{link to next instruction}\end{array}}\,. \tag{4.1}$$

Typically, an instruction makes reference to some associated data. These data can be in an *inline buffer*, immediately following the instruction, or in a separately allocated buffer. In either case, the buffer pointer field in (4.1) points to the origin of this buffer. Small amounts of data can also be encoded in *immediate* form, i.e., within the instruction itself; in this case no data buffer is used.

Scanning of a step list proceeds sequentially from the beginning to the end of the list—there are no flow-control instructions such as IFs, JUMPs, or GOTOs.

Writing a memory address to the Next List Pointer tells the STEP controller that a step list is present in memory at the given address and that the STEP controller should start executing it as soon as it is done with the list (if any) which it is currently executing. The exact sequence of events will depend on the current state of the STEP controller. For the purposes of writes to the NLP, the STEP controller can be in one of three states: *execute*, *wait-for-host*, and *error-wait-for-host*.

In the execute state, the STEP controller is still executing the previous step list. Assuming that no errors are encountered, the STEP controller will start executing the new list as soon as the previous one is finished.

[1] The step list is analogous to the "display list" routinely used for interfacing a conventional processor to a specialized, high-performance graphics processor.

In the wait-for-host state, the STEP controller is idle. As soon as the NLP is written to, the STEP controller immediately starts execution of the given list.

♠ In the error-wait-for-host state, writing to the ... [clarify this] this register can be written, but is is additionally necessary for the user (through the driver) to indicate to CAM that it is acceptable to resume execution, through the "enable-dvma" option – simply writing to the NLP is insufficient. The reason for this is to prevent execution from continuing when the interface receives an error during the write to the NLP, in which case execution of the new step list should not commence.

## 4.4 The step controller's control registers

The STEP controller's control registers comprise one read/write register (the Next List Pointer), three read-only registers, and three write-only registers. Read-only and write-only registers share the same three memory locations. Thus, when read from, the STEP controller's control registers have the following description

| STEP controller's registers (read) | |
|---|---|
| NLP | Next List Pointer |
| ISR | Interrupt Status Register |
| CIP | Current Instruction Pointer |
| PIP | Previous Instruction Pointer |

, (4.2)

and when written to they have the following description

| STEP controller's registers (write) | |
|---|---|
| NLP | Next List Pointer |
| ICR | Interrupt Control Register |
| DSL | Display Scan Length |
| DBL | Display Blank Length |

. (4.3)

The three pointers above (NLP, CIP, and PIP), the low four bits have a special use. The addresses to which the pointers refer are always 16-byte aligned (cf. §4.2), so that their low four bits are predictably 0s and need not be explicitly given. In their stead, those four bit positions are used as read-only windows on certain status bits (writes to the same bit positions are ignored.)

### 4.4.1 Next List Pointer

The Next List Pointer is a read/write register. Bits 4–31 (`next_list`) represent the starting address (without the low four bits, which are understood to be 0s) of the step list to be executed next. Bits 0–3 are read-only status bits (cf. §4.4) with the following meaning

| Next List Pointer status bits | | |
|---|---|---|
| bit | name | function |
| 0 | `nlp_written` | the STEP controller is still executing the previous display list, and has not yet made use of the NLP. |
| 1 | `exception_status` | the STEP controller has seen an exception (timeout or STEP interrupt). |
| 2 | `wait_for_host` | the STEP controller is in "wait-for-host" mode. |
| 3 | `interface_halted` | the STEP controller is currently halted. |

. (4.4)

[revise this paragraph] Bit 3 is a status bit which is set if CAM is currently scanning. Bit 2 is set if the interface is in the "DMA Active" mode, meaning that it is performing, or ready to perform the execution of step list. Bit 1 is set if the interface is in "wait-for-host" mode, and bit 0 is set if the current NLP has not yet been processed by the interface (meaning that the interface is still working on the previous display list.) ♠

### 4.4.2 Interrupt Status/Control Registers

The Interrupt Status Register (Fig. 4.1) and the Interrupt Control Register (Fig. 4.2) are used to monitor, enable, or disable interrupts generated by the STEP controller. The latter register is is also used to reset, halt, and restart the STEP controller, and to reset the STEP array.

The STEP controller can place on the SBus five kinds of interrupts, namely,

| STEP controller interrupts | |
|---|---|
| name | description |
| `soft_int` | Software interrupt. |
| `cam_int` | STEP-originated interrupt. |
| `sbus_int` | The STEP controller detected an SBus error. |
| `timeout_int` | Timeout in communicating with the STEP array. |
| `nlp_int` | Next List Pointer interrupt: the controller has started executing the step list indicated by the Next List Pointer; a new step list can be scheduled by writing a new adress to the Next List Pointer. |

. (4.5)

To each of these interrupts the STEP controller associates an *interrupt enable* bit and an *interrupt flag* bit, whose functions will be explained in a moment. The five enable bits and the five flag bits are directly readable respectively in positions 0–4 and 8–12 of the Interrupt Status Register.

The five flag bits cannot be directly written by the host. Each of these bits is set by the occurrence of the corresponding interrupt condition (e.g., timeout interrupt); they are collectively cleared by any read operation performed on the Interrupt Status Register.

Whether a flagged interrupt actually generates an interrupt on the SBus depends on whether that particular interrrupt is enabled. The five enable bits of the Interrupt Status Register are not writable in a conventional way. They are *set* by writing a 1 to respectively bits 8, 9, 10, 11, and 12 of the Interrupt Control Register, and *cleared* by writing a 1 to, respectively, bits 0, 1, 2, 3, and 4 of the same register.

| Interrupt Status Register | |
|---|---|
| bit | name |
| 0 | soft_int_flag |
| 1 | cam_int_flag |
| 2 | sbus_int_flag |
| 3 | timeout_int_flag |
| 4 | nlp_int_flag |
| 5–7 | — |
| 8 | soft_int_enable |
| 9 | cam_int_enable |
| 10 | sbus_int_enable |
| 11 | timeout_int_enable |
| 12 | nlp_int_enable |
| 13–31 | — |

Figure 4.1: Structure of the Interrupt Status Register. Bits 0–4 (interrupt flags) are set by the occurrence of the appropriate interrupt conditions, and are automatically cleared by any read of this register. Bits 8–11 are set by writing 1s to bits 8–11 of the Interrupt Control Register, and cleared by writing 1s to bits 0–5 of the same register.

The occurence of a `cam_int` or a `timeout_int` suspends the list scanning activity of the STEP controller, regardless of whether these interrupts are enabled (and thus reported to the host as SBus interrupts). However, these interrupts can be masked at an earlier stage, so that they will not be seen at all by the STEP controller. This is done by two more enable bits, namely `cam_exception` and `timeout_exception`, which can be set or cleared in a way similar to the other enable bits, by writing a 1 to, respectively, bits 5 and 6 or bits 13 and 14 of the Interrupt Control Register.

Finally, as shown in Fig. 4.2, four more control functions are performed by writing a 1 to bits 12, 13, 16, and 31 of the Interrupt Control Register.[2]

| Interrupt Control Register | |
|---|---|
| bit | effect of writing a 1 |
| 0 | `soft_int` disable |
| 1 | `cam_int` disable |
| 2 | `sbus_int` disable |
| 3 | `timeout_int` disable |
| 4 | `nlp_int` disable |
| 5 | `cam_exception` disable |
| 6 | `timeout_exception` disable |
| 7 | — |
| 8 | `soft_int` enable |
| 9 | `cam_int` enable |
| 10 | `sbus_int` enable |
| 11 | `timeout_int` enable |
| 12 | `nlp_int` enable |
| | also: Clear fault |
| 13 | `cam_exception` enable |
| | also: Clear exception |
| 14 | `timeout_exception` enable |
| 16 | Halt the STEP controller. |
| 17–30 | — |
| 31 | Reset the STEP controller. |

Figure 4.2: Structure of the Interrupt Control Register.

### 4.4.3 Current Instruction Pointer

The Current Instruction Pointer is a read-only register (writes to this location affect the Display Scan Length; §4.4.5). Bits 4–31 (`curr_inst`) represent the address (without the low four bits, which are understood to be 0s) of the step-list instruction currently being executed. Bits 0–3 are read-only status bits (cf. §4.4) with the following meaning

[2]Bits 12 and 13 happen to affect also exception control (see above); this is an oversight (fortunately a harmless one), and will be fixed with the next release of the STEP controller chip set.

| Current Instruction Pointer status bits | | |
|---|---|---|
| bit | name | function |
| 0 | `sbus_error` | The STEP controller is suspended because of SBus error. |
| 1 | `partial_write` | Owing to an SBus error, only a partial write was performed by the STEP controller. |
| 2 | `done_status` | STEP controller is not relaying to STEP. |
| 3 | — | Reserved. |

(4.6)

[Edit bit 2.]

### 4.4.4 Previous Instruction Pointer

The Previous Instruction Pointer is a read-only register (writes to this location affect the Display Blank Length; §4.4.5). Bits 4–31 (`prev_inst`) represent the address (without the low four bits, which are understood to be 0s) of the step-list instruction executed before the current one. Bits 0–3 are read-only status bits (cf. §4.4) with the following meaning

| Previous Instruction Pointer status bits | | |
|---|---|---|
| bit | name | function |
| 0 | `cam_int_active` | This bit reflects the status of the Interrupt signal of the TBus, resynchronized to the SBus clock. It allows the host to directly see whether there is an interrupt on the TBus. |
| 1 | `cam_scanning` | STEP is scanning |
| 2 | | (Reserved for SPARCstation-2 bit.) |
| 3 | | (Reserved.) |

(4.7)

Bit 2 (?) of the previous instruction register indicates if the long-burst transfer mode supported on the SPARCstation-2 2 is being used by the interface board.

### 4.4.5 Display Scan/Blank Length

### 4.4.6 Remarks

[understand and drop in place the following remarks.]

wait-for-host is fixed by writing to the NLP

all error interrupts except new-list (NLP) and soft cause dvma-inactive and wait-for-host mode.

cam-error: parity and verify-write errors are not signalled until end of transfer, this interrupt will always be in the "next" instruction.

current-instruction: generally this is the current instruction if "stop" is received. This instruction should have been attempted, the previous instruction has generally been completed.

Check the state that interface and cam are in when a hardware interrupt is received. do not assume that interface is in some "clean" state.

## 4.5 Step device driver

All interaction between a program running on the host under UNIX and the STEP controller is mediated by an *installable device driver*[], called the driver, which acts as a custom extension of the UNIX kernel.

UNIX treats devices as a special files, to which certain special-purpose operations (`ioctls`) are applicable in addition to the standard open, close, read, and write calls. Devices appear in the `/dev` directory; the driver is listed as `/dev/cam`.

### 4.5.1 Using the driver

In order for the driver's entry points to be accessible by a user program, the `fcntl.h` and `errno.h` system header files as well as the custom header file `camio.h` must be included. The latter declares the entry points to the driver, defines the structure `sl_element` of a step list instruction, and also defines the arguments for the different `ioctl` calls permitted on the driver.

The `cam` device is opened with the standard `open` call. The cam device must be opened for read/write access (`O_RDWR`) in order to function correctly. Similarly, `cam` is closed with the standard `close` call. All meaningful interactions with the device are done through `ioctl` calls. The standard `read` and `write` calls are not used; though entry points for these calls are provided, they result simply in an error being returned, with no effect on the state of STEP or its device driver.

The macros for the `ioctl` calls, described in the following subsections, are defined in the `camio.h` header file. The call names are identical to the subsection headings. (Eventually, both call names and subsection headings should be updated to reflect the actual register names.)

`CIOSTEP`

This call is used to schedule a step list. Basically, the call receives a pointer to the beginning of a step list, ideally in both the user and kernel address spaces. The call prepends a no-op instruction with a soft interrupt to the beginning of the step list, sets the Next List Pointer on the STEP controller, and then waits for the hardware to perform the interrupt. The call returns as soon as the soft interrupt is received. (This is the typical behavior expected from a read/write device driver call.)

`CIOSTOP`

This call is essentially the scheduling of a stop step list. The stop call generates a no-op instruction with a soft interrupt and a wait for host. The host next pointer on the interface is set appropriately, and the call waits for the hardware to perform the interrupt. The call returns as soon as the soft interrupt is received.

`RESET-INTF`

This call is used to reset the STEP controller. The associated bit of **cam_rstintr** automatically clears itself when the STEP controller reset sequence is complete.

`RESET-CAM`

This call is used to reset the STEP hardware. Unlike the STEP controller reset, this reset is not self-clearingd; the associated bit of **cam_rstintr** needs to be explicitly cleared. This bit should not be cleared until the driver has waited a sufficient time for the STEP hardware to have completed the reset. (For the actual timing of the STEP reset sequence, consult Tom D.) ♠

`SPARC-2?`

Returns the type of SPARCstation in which the board is installed. This call exists so that the hardware may take advantage of the ability of newer SPARCstations to use 16-word bus transfers in addition to 4-word transfers. (Actually, it may not be necessary to make this a call, as it is probably not necessary for the higher-level software to be concerned with this distinction. Basically, the purpose of this is to permit 16-word transfers on the SPARC-2 and limit transfers to 4-words on the SPARC-1.) ♠

`Get-status-interrupt`

Senses the interrupt status of the device. It returns a **u_int** which encodes the interrupt status. This is the mechanism for passing interrupt information to the user program, since the **cam_intr** routine, being a "bottom half" routine[], cannot directly pass information to the user program.

`Clear-status-interrupt`

Clears the interrupt status of the device as seen by a user program.

`MEM` ♠

This kernel call is supposed to allocate memory for sharing between the user program and the device. This needs to allocate memory that is mapped in the kernel context (as well as the user context.)

`GET-HNP`

Returns the value of the Next List Pointer. ♠

`GET-CURR-INST`

Returns the value of the Current Instruction Pointer..

`GET-PREV-INST`

Returns the value of the Previous Instruction Pointer.

# 5 The data stream

## 5.1 Sources and destinations

Central to the STEP updating and analysis process are five internal inputs within each STEPchip which we shall call *destinations*:

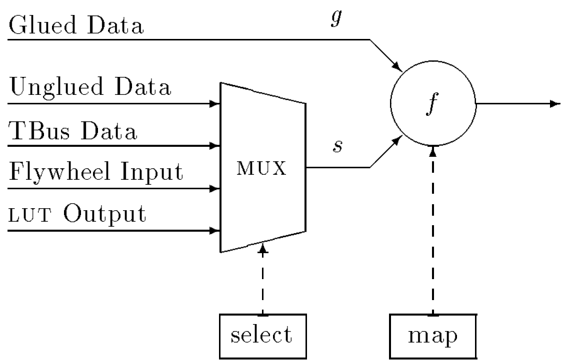
- *LUT Address* (§6.4). The LUT Address bit contributed by the STEPchip. Collectively, the sixteen LUT Address bits contributed by a STEPchip set make up the address used, at every site of a scan, for lookup in the active LUT.

- *Flywheel Output* (§6.5). The bit to be placed on the STEPchip's Flywheel Data Output pin (§10.2.5)—perhaps to be modified and returned through the Flywheel Data Input pin.

- *Site Data* (§6.6). The Site Data bit contributed by the STEPchip. Collectively, the sixteen Site Data bits contributed by a STEPchip set make up the new value to be written in the site being scanned (cf. Fig. 2.1).

- *Event Counter* (§6.7). The bit to be fed to the STEPchip's Event Counter (see §5.2). Each of the sixteen STEPchips of a module has a separate event counter.

- *Display Data* (§6.8) The data to be displayed on the dedicated monitor, or read back by the host during scan I/O (§6.16).



$x$ Source Register

Figure 5.1: Source selection and combination network. Each of the five destinations has its own network, driven by the corresponding source register ($x$ stands for one of the possible five destinations). The last input to the multiplexer is LUT Output (as shown in the figure) for the three destinations Site Data, Event Counter, and Display. For the other two destinations, i.e., LUT Address and Flywheel Output, it is the bit specified by the Side Address Bit Select Register. For $g$ and $s$, cf. table (5.2).

Each destination independently can be fed, under software control, with data derived from a number of sources. Separate registers (the Site Address Bit Select Register mentioned above and the registers described in the next few sections) determine the data sources that are used for each of these destinations. In all cases, source selection and combination is made in two stages, using a two-bit *select* register and a four-bit *map* register, as sketched in Fig. 5.1.

The select register selects one source out of four, namely,

| select | source |
|---|---|
| 0 | Unglued Data bit (§??). |
| 1 | TBus Data bit (§6.14, §6.14). |
| 2 | Flywheel Input bit (§??). |
| 3 | *When the destination is either the* LUT *address bit or the flywheel output bit*: one bit of the site address, as specified by the Site Address Bit Select Register. *For all other destinations:* the LUT output. |

(5.1)

The selected source is combined with another source, namely, Glued Data, by a two-input, one-output Boolean function $f$. The map register (Fig. 5.1) specifies which of the sixteen possible Boolean functions is to be used for $f$ according to the following table, where $g$ denotes the Glued Data source and $s$ denotes the source chosen by the select register.

| map | Boolean function |
|---|---|
| 0 | $0$ |
| 1 | $\overline{g\ nor\ s}$ |
| 2 | $g\ and\ \overline{s}$ |
| 3 | $\overline{s}$ |
| 4 | $\overline{g}\ and\ s$ |
| 5 | $\overline{g}$ |
| 6 | $g\ xor\ s$ |
| 7 | $\overline{g\ and\ s}$ |
| 8 | $g\ and\ s$ |
| 9 | $g = s$ |
| 10 | $g$ |
| 11 | $g\ or\ \overline{s}$ |
| 12 | $s$ |
| 13 | $\overline{g}\ or\ s$ |
| 14 | $g\ or\ s$ |
| 15 | $1$ |

(5.2)

## 5.2 Event Counter

With a maximum space size of $2^{24}$ sites in a module, the event counter must be able to store the $2^{24}+1$ possible values for the count, from 0 to $2^{24}$. Thus, we need a 25-bit counter and a 25-bit buffer.

## 5.3 Data flywheel

are taps on the data streams that circulate within each STEPchip. For each layer, all the data accessed during a scan are streamed past an output tap and an input tap. Thus, when provided with appropriate synchronization, external hardware can, through these taps, selectively extract any of the data involved in the simulation, and can optionally replace them with arbitrary new data. The sixteen Flywheel Output and the sixteen Flywheel Input lines of each module, together with the control lines that decide whether external injection of data is going to take place, are simply brought out to connectors on the motherboard; they are mainly intended for experimental use of the machine. (It is unlikely that even the most advanced user will connect external equipment to more than one box.)

# 6 stepchip Registers

This chapter systematically presents (see cf. §3.10 for an overview) the registers that make up the STEPchip *register address space*. This address space is directly accessible by the TBus. Other on-chip data structures are accessed only in an indirect way (for instance, the Event Counter is read by the user via a register called the Event Count Register), or are of no immediate relevance to the STEP user.

Each register is presented in a separate section; the section number itself (e.g., the '23' in §6.23) coincides with the register's address (§3.10). The makeup of each register is summarized in tabular form immediately below the section title. The register's length in bits is given, followed by the register's mnemonic and its full name. Often, a register is actually a collection of logically separate components; in such cases the components themselves are individually listed, starting from the one at the least-significant-digit end of the register. For example, the 30 bits of the Kick Register (§6.2) comprise a 24-bit Kick Amount (whose least significant bit is the least significant bit of the entire Kick Register), three one-bit Kick Signs, and three one-bit Kick Max Flags. This listing order corresponds to the order in which bits are transmitted to or received from a register, i.e., least-significant bit first.

All registers are both readable and writable, although most would only be read for error recovery or test purposes. At power-on reset, all registers except the Module ID Register are cleared to zero (see §6.23); a host-initiated, *soft*, reset is similar, but leaves the DRAM contents and several registers (OSR, IR and DOCR) unchanged.

As mentioned in §3.10, the two I/O registers, namely the Scan I/O Register and the LUT I/O Register, follow their own conventions concerning length of transfers. Moreover, the Event Count Register, whose full length is 25 bits, may be read or written for only part of its length, as specified by the Event Counter Length component of the Scan Format Register.

For reference, a summary description of STEP registers is given immediately below; detailed descriptions follow.

| Address | Bits | Mnemonic | Name |
|---|---|---|---|
| 0 | 3 | MSR | Module Select Register |
| 1 | 6 | RMR | Run Mode Register |
| | 2 | SSM | Scan Sync Mode |
| | 1 | RT | Run Type |
| | 1 | ECT | Event Count Transfer |
| | 1 | RPK | Repeat Previous Kick |
| | 1 | ALT | Active LUT Toggle |
| 2 | 30 | KR | Kick Register |
| | 24 | KA | Kick Amount |
| | 1 | XKS | X-Kick Sign |
| | 1 | YKS | Y-Kick Sign |
| | 1 | ZKS | Z-Kick Sign |
| | 1 | XKMF | X-Kick Max Flag |
| | 1 | YKMF | Y-Kick Max Flag |
| | 1 | ZKMF | Z-Kick Max Flag |
| 3 | 5 | SABSR | Site Address Bit Select Register |
| 4 | 6 | LASR | LUT Address Source Register |
| | 2 | LAS | LUT Address Select |
| | 4 | LAM | LUT Address Map |
| 5 | 6 | FOSR | Flywheel Output Source Register |
| | 2 | FOS | Flywheel Output Select |
| | 4 | FOM | Flywheel Output Map |
| 6 | 6 | SDSR | Site Data Source Register |
| | 2 | SDS | Site Data Select |
| | 4 | SDM | Site Data Map |
| 7 | 6 | ECSR | Event Counter Source Register |
| | 2 | ECS | Event Counter Select |
| | 4 | ECM | Event Counter Map |
| 8 | 6 | DSR | Display Source Register |
| | 2 | DDS | Display Data Select |
| | 4 | DDM | Display Data Map |
| 9 | 1 | SSR | Show Scan Register |
| 10 | --- | ECR | Event Count Register |
| 11 | 16 | LIR | LUT Index Register |

**12**

| | | |
|---|---|---|
| 5 | LIPR | LUT Index Permutation Register |

**13**

| | | |
|---|---|---|
| --- | LIOR | LUT I/O Register |

**14**

| | | |
|---|---|---|
| 24 | SIR | Scan Index Register |

**15**

| | | |
|---|---|---|
| 120 | SIPR | Scan Index Permutation Register |
| 5 | SSA0 | Select Site Address 0 |
| 5 | SSA1 | Select Site Address 1 |
| | . | |
| | . | |
| | . | |
| 5 | SSA23 | Select Site Address 23 |

**16**

| | | |
|---|---|---|
| --- | SIOR | Scan I/O Register |

**17**

| | | |
|---|---|---|
| 35 | SFR | Scan Format Register |
| 2 | SM | Scan Mode |
| 5 | ESC | End-of-Scan |
| 4 | ESW | End-of-Sweep |
| 4 | EST | End-of-Stretch |
| 5 | SBRC | Sweeps Between Refresh Cycles |
| 8 | RCL | Refresh Cycle Length |
| 5 | ECL | Event Counter Length |
| 2 | STM | Stretch Magnification |

**18**

| | | |
|---|---|---|
| 24 | OSR | Offset Register |

**19**

| | | |
|---|---|---|
| 38 | DR | Dimension Register |
| 23 | DCM | Dimension Cut Mask |
| 5 | XDCP | X-Dimension Cut Pointer |
| 5 | YDCP | Y-Dimension Cut Pointer |
| 5 | ZDCP | Z-Dimension Cut Pointer |

**20**

| | | |
|---|---|---|
| 16 | HER | Hardware Environment Register |
| 1 | LPL | LUT Pipeline Length |
| 1 | FPL | Flywheel Pipeline Length |
| 1 | DCS | DRAM Column Size |
| 6 | TBD | Tree Balancing Delay |
| 4 | TMS | Test Mode Select |
| 1 | NBF | Node Bypass Flag |
| 1 | SRE | Select Read Enable |
| 1 | ALS | Active LUT Select |

**21**

| | | |
|---|---|---|
| 10 | MPCR | Multipurpose Pin Control Register |
| 5 | MAFS | Multipurpose A Function Select |
| 5 | MBFS | Multipurpose B Function Select |

**22**

| | | |
|---|---|---|
| 18 | GPCR | Glue Pin Connectivity Register |
| 3 | XMPC | XM-Pin Connect |
| 3 | XPPC | XP-Pin Connect |
| 3 | YMPC | YM-Pin Connect |
| 3 | YPPC | YP-Pin Connect |
| 3 | ZMPC | ZM-Pin Connect |
| 3 | ZPPC | ZP-Pin Connect |

**23**

| | | |
|---|---|---|
| 1 | MIDR | Module ID Register |

**24**

| | | |
|---|---|---|
| 1 | GIDR | Group ID Register |

**25**

| | | |
|---|---|---|
| 10 | IER | Interrupt Enable Register |
| 1 | BPIE | Bus Parity Interrupt Enable |
| 1 | BCIE | Bus Contention Interrupt Enable |
| 1 | GCIE | Glue Contention Interrupt Enable |
| 1 | MAIE | Multi-A Contention Interrupt Enable |
| 1 | MBIE | Multi-B Contention Interrupt Enable |
| 1 | SSIE | Skewed Scan Interrupt Enable |
| 1 | XHIE | External Hardware Interrupt Enable |
| 1 | RLIE | Register Length Interrupt Enable |
| 1 | URIE | Unknown Register Interrupt Enable |
| 1 | ISIE | Invalid SYNC Interrupt Enable |

**26**

| | | |
|---|---|---|
| 10 | IFR | Interrupt Flags Register |
| 1 | BPIF | Bus Parity Interrupt Flag |
| 1 | BCIF | Bus Contention Interrupt Flag |
| 1 | GCIF | Glue Contention Interrupt Flag |
| 1 | MAIF | Multi-A Contention Interrupt Flag |
| 1 | MBIF | Multi-B Contention Interrupt Flag |
| 1 | SSIF | Skewed Scan Interrupt Flag |
| 1 | XHIF | External Hardware Interrupt Flag |
| 1 | RLIF | Register Length Interrupt Flag |
| 1 | URIF | Unknown Register Interrupt Flag |
| 1 | ISIF | Invalid SYNC Interrupt Flag |

**27**

| | | |
|---|---|---|
| 3 | VWR | Verify Write Register |
| 1 | VWE | Verify Write Enable |
| 1 | VWIE | Verify Write Interrupt Enable |
| 1 | VWIF | Verify Write Interrupt Flag |

**28**

| | | |
|---|---|---|
| 16 | DOCR | DRAM Ones Count Register |
| 8 | LDOC | Low DRAM Ones Count |
| 8 | HDOC | High DRAM Ones Count |

## 6.0 Module Select Register

| | | |
|---|---|---|
| 3 | MSR | Module Select Register |

In order for a STEP module to be enabled to accept the data broadcast by the host on the TBus during a write frame or put its own data on the TBus during a read frame, the module must be *selected*, as explained in §3.13. The parameters upon which this selection depends are specified by a three-bit *selection code* written to the Module Select Register. Selection codes (see table (6.1)) specify which internal register should match which target value. For the selection to be successful. all STEPchips in the module must simultaneously indicate a match between their selected registed and their target.

| selection code (MSR) | bits to be matched: internal | target |
|---|---|---|
| 000 | MIDR | 0 |
| 010 | MIDR | 1 |
| 100 | MIDR | LIR |
| 110 | Glue-AND | 0 |
| 001 | GIDR | 0 |
| 011 | GIDR | 1 |
| 101 | GIDR | LIR |
| 111 | don't care | |

(6.1)

Owing to its special role in the selection mechanism, the Module Select Register "escapes selection," in the sense that data *written* to it by the TBus are accepted by *every* module regardless of the module's current selection status. The selection status of a module (i.e., selected or not) is latched when this register is written, and remains in force until this register is written again.

The Module Select Register also escapes normal selection on reads: read operations require a single module to be selected, and so the normal selection mechanism wouldn't allow us to read back the current selection status of STEP if more than one module happened to be currently selected! Since the Module Select Registers of all modules are always written simultaneously, it doesn't matter *which* module responds to a read of the MSR, as long as only one does: a bit in the Hardware Environment Register (§6.20) is used to single out one particular module for this "honor."

## 6.1 Run Mode Register

| | | |
|---|---|---|
| 6 | RMR | Run Mode Register |
| 2 | SSM | Scan Sync Mode |
| 1 | RT | Run Type |
| 1 | ECT | Event Count Transfer |
| 1 | RPK | Repeat Previous Kick |
| 1 | ALT | Active LUT Toggle |

Writing a nonzero value to the Scan Sync Mode *schedules* a scan (cf. §2.7), i.e., instructs the STEPchip to start a new scan as soon as possible; the specific value written in the register determines to what extent the scan itself and the individual sweeps that make it up must be synchronized—as detailed in table (6.2)—with the VCAM and HCAM pulses that travel on the TBus Sync line (§??). At the end of the scan, this register is automatically cleared.

| SSM | Scan Sync Mode |
|---|---|
| 0 | *Standby.* No scan scheduled. |
| 1 | *Frame sync.* The starting of individual sweeps of the scan will wait for a horizontal sync pulse, HCAM; the first sweep of the scan (and thus the entire scan) will wait for a vertical sync pulse, VCAM (which also acts as an HCAM pulse). |
| 2 | *Line sync.* Individual sweeps of the scan will wait for HCAM. VCAM is treated as equivalent to HCAM [**check this**]. |
| 3 | *Free run.* The scan will start immediately, and sweeps will follow one another without any gaps. |

(6.2)

The HCAM and VCAM pulses appear on the same TBus Sync line (§3.6) on which the HMON and VMON pulses appear, and are in a definite phase relationship with them. The latter are sent to the CRT monitor to which video data are fed (§6.8). Thus, in a scan of sync mode 1 a sector may be scanned in lockstep with the scanning of the CRT monitor. A sync mode 2 scan typically consists of a single sweep; this is used when one intends to change some of the registers (e.g., the Kick Register) from one sweep to the next. A sync mode 3 scan, in which the timing of the scanning is not constrained by real-time display considerations, yields, of course, the fastest scanning rate.

In all cases, the actual start of each sweep is shifted in time by the amount specified by the Tree Balancing Delay (§6.20), to insure that modules placed at different distances from the TBus root will all scan in lockstep (cf. §??).

The Scan-in-Progress signal is defined as the logical OR of the two bits of the Run Mode Register, and has a value of 1 from the moment a scan is scheduled to the moment the scan itself ends. This signal is made available to the STEP interface via the Status line (cf. §3.6, §6.17), to allow the interface to determine when a scan is finished, and thus when operations that must wait for end-of-scan (such as the scheduling of a new scan) can proceed.

Although the Run Mode Register isn't double buffered, the delay between the end of one scan and the beginning of the next need not be longer than a few cycles. In fact, the data to be sent to this register to schedule a new scan can be loaded into the STEP interface during the current scan, ready to be broadcast to the STEP modules as soon as the Scan-in-Progress signal is no longer asserted (cf. §??). (As explained in §6.10, the event counts for the current scan are double-buffered and can be read at leisure *after* the new scan has started.)

Several other actions associated with scheduling a scan are controlled by the Run Mode Register, and occur (if indicated) when this register is written.

The Run Type is used to label a scan with some extra "type" information (one bit per STEPchip). This information is made visible to external hardware during the scan via multipurpose pins (see §6.21, Run Type). The Run Type is then reset to zero at the end of the scan. If, for example, a Frame Buffer is attached to the CAM display, and a multipurpose pin is configured appropriately, we could signal which scans should be captured for display by appropriately setting a Run Type bit whenever we write the Run Mode Register.

The Event Count Transfer bit controls whether to copy the value accumulated in the Event Counter (over perhaps several scans) into the Event Count Register. Whenever a transfer occurs, the Event Counter itself is automatically cleared. Thus, as we start a new scan, the count from the previous scan can be made available to be read back by the host during the new scan (§6.10).

The Repeat Previous Kick bit is an optimization. Normally, the Offset Register is decremented by the Kick Amount whenever the Kick Register is written. If the kick for the next scan is the same as that just used, we can avoid rewriting the Kick Register by simply reusing its current value, and again decrementing the offset. Note that if we want a kick of zero, we must explicitly change the kick to zero (by writing the Kick Register); it is not enough to simply fail to set the Repeat Previous Kick bit.

Special care must be taken when the scanning of a sector is broken up into several subscans. Kicks sent (or repeated) on each subscan would accumulate, which usually is not what we want. On the other hand, sending a kick of zero for all subscans except the first would produce the right results as far as the offset is concerned, but would deprive the gluing machinery of the relevant information. The correct approach when gluing is involved is, before each subscan after the first, to "undo" the previous kick (by sending its complement) and then send the desired kick afresh.

The Active LUT Toggle switches the roles of *active* and *inactive* LUT between LUT 0 and LUT 1. The active LUT is the one that will be used as the main processing element in the data stream (cf. Fig. ??). The other, or inactive, LUT is accessible to the TBus via the LUT I/O Register (cf. §6.11); specifically, it can be loaded with new contents even while a scan is in progress (cf. §3.2).

One can find out which of the two LUTs is the active one by reading the Active LUT Select bit in the Hardware Environment Register (§6.20). By writing that bit, one can force a specific LUT (rather than just "the other one") to be the active one.

## 6.2 Kick Register

| | | |
|---|---|---|
| 30 | KR | Kick Register |
| 24 | KA | Kick Amount |
| 1 | XKS | X-Kick Sign |
| 1 | YKS | Y-Kick Sign |
| 1 | ZKS | Z-Kick Sign |
| 1 | XKMF | X-Kick Max Flag |
| 1 | YKMF | Y-Kick Max Flag |
| 1 | ZKMF | Z-Kick Max Flag |

The basic philosophy of "kicking" is explained in §2.10.

The bits of the Kick Amount constitute a $d$-component vector, $\vec{k}$, representing a displacement in a $d$-dimensional space. The structure of this vector (number $d$ of components and number of bits devoted to each component) is determined by the Dimension Register, as explained in §2.4. As soon as it is sent, this vector is subtracted from the Offset Register (§6.18)—another vector with the same structure which keeps track of the current position of the origin of coordinates of STEP space.

Subtraction of the Kick Amount from the Offset Register, and, in general, all site-address arithmetic is performed independently on the separate site-address slices determined by Dimension Cut Mask in the Dimension Register. Each slice of the Kick Amount specifies the size of the kick for the corresponding dimension. For the three extendible dimensions $x$, $y$, and $z$ the Kick Amount is further qualified by a Kick Sign and a Kick Max Flag; the information contained in all three registers is used *during a scan* by the glue machinery.

If $n$ is the number of site address bits assigned to a certain dimension by the Dimension Cut Mask (cf. Fig. 2.4), then the Kick Amount for that dimension ranges from 0 thru $2^n - 1$ ($2^n$ distinct choices).

For each of the three extendible dimensions, the corresponding Kick Amount component is extended by a Kick Sign bit and a Kick Max Flag bit. With the help of these two bits, the range of the kick is extended to the interval $[2^{-n}, 2^{+n}]$ (overall, $2^{n+1}+1$ distinct choices). Jointly, Kick Amount and Kick Sign express the value of the kick in two's complement notation, from $2^{-n}$ through $2^{+n}-1$ (i.e., $2^{n+1}$ choices). The last value of the interval, namely $2^{+n}$, is distinguished from the value of 0 by writing a 1 in the Kick Max Flag.

In the following discussion, we'll use, for definiteness, $n = 8$. Then the Kick Amount component ranges from `00` to `FF` (in hex). A kick of `01` will circularly shift (within the sector boundaries) the layer's data by one position in the positive direction; a kick of `FF` will shift the data by 255 positions in the positive direction, or, which is equivalent, by one position in the negative direction.

For each of the three extendible dimensions, the shift specified by the Kick Amount affects the layer as a whole (rather than each sector being independently circularly shifted), and the Kick Sign is used as the most significant bit in the two's complement notation. One obtains in this way a kick range from −256 to +255; the value +256 is also possible using only data from nearest neighboring modules, and is indicated by setting the corresponding Kick Max Flag. The entire range of cases, for an 8-bit slice, is summarized by the following table (using the $x$ dimension for definiteness)

| XKMF Max flag | XKS sign | KA amount | Actual shift |
|---|---|---|---|
| 1 | 0 | 00 | +256 |
| 0 | 0 | FF | +255 |
| 0 | 0 | FE | +254 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 0 | 0 | 02 | +2 |
| 0 | 0 | 01 | +1 |
| 0 | 0 | 00 | 0 |
| 0 | 1 | FF | −1 |
| 0 | 1 | FE | −2 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 0 | 1 | 02 | −254 |
| 0 | 1 | 01 | −255 |
| 0 | 1 | 00 | −256 |

(6.3)

Note that if the Kick Sign and Kick Max Flags are both fixed at zero, and the Dimension Cut Pointers (§6.19) are set appropriately (a value of 31), then $x$, $y$, and $z$ can be used as internal dimensions: they will be internally wrapped around and no use will be made of glue.

## 6.3 Site Address Bit Select Register

| | | |
|---|---|---|
| 5 | SABSR | Site Address Bit Select Register |

This register selects one of the Site Address's 24 bits as part of the data to be sent to a given destination, when source 3 is selected (cf. §5.1). Specifically, routing site-address information to the lookup table is one way to make the dynamics *space*-dependent.

## 6.4 LUT Address Source Register

| | | |
|---|---|---|
| 6 | LASR | LUT Address Source Register |
| 2 | LAS | LUT Address Select |
| 4 | LAM | LUT Address Map |

LUT Address Select and LUT Address Map are used to construct the source for the LUT Address destination signal (cf. §5.1).

## 6.5 Flywheel Output Source Register

| | | |
|---|---|---|
| 6 | FOSR | Flywheel Output Source Register |
| 2 | FOS | Flywheel Output Select |
| 4 | FOM | Flywheel Output Map |

Flywheel Output Select and Flywheel Output Map are used to construct the source for the Flywheel Output destination signal (cf. §5.1).

## 6.6 Site Data Source Register

| | | |
|---|---|---|
| 6 | SDSR | Site Data Source Register |
| 2 | SDS | Site Data Select |
| 4 | SDM | Site Data Map |

Site Data Select and Site Data Map are used to construct the source for the Site Data destination signal (cf. §5.1).

## 6.7 Event Counter Source Register

| | | |
|---|---|---|
| 6 | ECSR | Event Counter Source Register |
| 2 | ECS | Event Counter Select |
| 4 | ECM | Event Counter Map |

Event Counter Select and Event Counter Map are used to construct the source for the Event Counter destination signal (cf. §5.1).

## 6.8 Display Source Register

| | | |
|---|---|---|
| 6 | DSR | Display Source Register |
| 2 | DDS | Display Data Select |
| 4 | DDM | Display Data Map |

The Display Source Register controls the relationship of the STEP unit to a dedicated video display, and also controls which data is read back during a scan I/O operation (cf. §6.16).

The Display Data Select and the Display Data Map are used to construct the source for the Display Data destination signal (cf. §5.1).

Note: Display Data signals output a value of zero during any clock when there is no valid display bit available.

## 6.9 Show Scan Register

| | | |
|---|---|---|
| 1 | SSR | Show Scan Register |

The STEP interface must wait for an end-of-scan condition before it proceeds to read or write a register involved in the scan. Since it is possible that not all modules are performing identical scans (e.g., one distinguished module may be acting as a frame buffer), we explicitly choose which module is to provide feedback on the state of the scan. Note that, on the selected module, this information is normally combined with the Interrupt signal coming from outer modules to produce the Status signal.

More specifically, when Show Scan is set, the Scan-in-Progress signal generated by the STEPchip (§6.1) is multiplexed onto the TBus Status line, which also carries the TBus Interrupt* signal: the Scan-in-Progress signal is chopped (ANDed with a signal that toggles at the clock rate), inverted, and ANDed with TBus Interrupt*.

## 6.10 Event Count Register

| | | |
|---|---|---|
| --- | ECR | Event Count Register |

The Event Count Register is used as a buffer for the STEPchip's event counter (§5.2), to permit reading the count from the previous scan even as a new count is accumulating during the current scan. It has the same physical length of 25 bits as the event counter, and is loaded with the contents of the latter at the moment that the Run Mode Register's Event Count Transfer bit is set (§6.1).

The event count is the most likely quantity that will be read from all modules at possibly frequent intervals during a simulation. Two provisions are made to optimize this readback.

With short scans, new event counts will be available with great frequency;[1] on the other hand, in this situation the maximum possible value recorded by the event counter will be small, so that the effectively used length of this counter may reduce to just a few bits. The Event Counter Length (§6.17) specifies this effective length, instructing the STEPchip to transmit only that many bits during the read-out of the Event Count Register. Note that registers are read by circularly shifting their contents past an output tap; when less than 25 bits are read from this register, the data will be left in the register with a different alignment than before the read operation. If a reread of the same data is desired, one must be aware of the new alignment.

By making use of the sequential-read mode (§3.15), it is possible to read all event counters in a single TBus frame, with automatic switchover of bus control from one module to the next. This allows all event counters to be read without repeatedly changing bus direction to explicitly select each module in turn.

Although the Event Count Register is normally only read, like all other registers it can also be written. To facilitate testing, when this register is written, the same value is also written into the event-counter itself. Reading this register or performing a verify write on it leaves the counter unaffected.

## 6.11 LUT Index Register

| | | |
|---|---|---|
| 16 | LIR | LUT Index Register |

The LUT Index Register indicates which entry of the inactive LUT (§6.1) is to be accessed. The pointer autoincrements each time an entry is read or written, thus pointing to the next entry to be accessed. A data transfer to this register is presumed to be finished when the pointer increments to zero.

The same register does double duty in a different context. When event counters are read back in sequential-read mode (§6.10, §3.15), the LUT Index Register provides the target name specifying which module is to be selected in turn. The pointer autoincrements each time Event Count Length number of bits have been read, thus pointing to the next module to be selected.

In principle, each STEPchip is responsible for only one bit of the pointer. For both hardware convenience and increased flexibility, the entire counter is repeated within each STEPchip, and each chip of a set is instructed by its LUT Index Permutation Register (§6.12) to use a different bit of this counter.

## 6.12 LUT Index Permutation Register

| | | |
|---|---|---|
| 5 | LIPR | LUT Index Permutation Register |

As mentioned in §6.11, each STEPchip has its own copy of the entire LUT Index Register but uses only one bit of it. The LUT Index Permutation Register specifies which of the LUT Index Register's 16 bits is to be used by a given STEPchip during table downloading.

Normally, STEPchip 0 will use the bit in position 0, STEPchip 1 that in position 1, etc., to compose the load

[1] In fact, short scans are usually set up in order to measure totals or averages separately over small patches of space rather than over an entire sector.

address. However, it is possible to specify a different correspondence between STEPchip numbers and bit positions. This feature can be used to download a given table with the roles of its inputs permuted. One can also bypass the LUT Index Register and send instead a constant of 0 or 1. This feature is used to download a specified portion of the LUT.

The available choices are summarized in the following table

| LIPR | LUT address bit source |
|---|---|
| 0 | bit 0 of LUT Index Register |
| 1 | bit 1 of LUT Index Register |
| 2 | bit 2 of LUT Index Register |
| ⋮ | ⋮ |
| 15 | bit 15 of LUT Index Register |
| 16–29 | not used |
| 30 | 0 |
| 31 | 1 |

(6.4)

## 6.13 LUT I/O Register

| --- | LIOR | LUT I/O Register |
|---|---|---|

As mentioned in §3.10, the LUT I/O Register is used for writing data to the inactive LUT (§6.1) or reading data from it, and does not have an intrinsic length.

The data transfer starts from the location initially specified by the LUT Index Register (§6.11), which will have been set up previously, and continues upwards (this pointer is autoincrementing). The transfer ends when the LUT Index Register reaches its terminal count; whereupon, on a read frame the STEPchip quits sending bits, while on a write frame it signals an error by setting the Register Length Interrupt Flag (cf. §??) if additional bits are received. After an error-free transfer the LUT Index Register contains a value of zero.

If the Verify Write Flag (§6.27) is set, any portion of the inactive LUT may of course be checked instead of written. Note that, unlike all other registers, it is not possible to use the verify write mode to perform *write-per-bit* (see §6.27) to the LUT I/O Register.

## 6.14 Scan Index Register

| 24 | SIR | Scan Index Register |
|---|---|---|

The Scan Index Register is an autoincrementing pointer which specifies the sequential place, within a scan, of the currently accessed site. At the beginning of an experiment this pointer will normally be initialized by writing a value of 0 to it. Since the length of a scan is always a power of two (see End-of-Scan Pointer, §6.17)—say, $2^n$—the $n$ low-order bits of the SIR will return to zero at the end of each scan. Thus, this register does not ordinarily need to be cleared between scans.

Since the SIR ultimately points to a DRAM bit (cf. Fig. 2.17), its clocking rate is geared to the accessibility of that bit as a source and/or a destination; on data transfers between the host and STEP, it is further constrained by the accessibility of the corresponding host bit. In the fastest scanning modes (modes 2 and 3 in §6.17), the SIR peak clock rate equals the system clock rate; in modes 0 and 1, it drops to one quarter that rate (since only one bit per DRAM nybble is effectively usable). During host I/O via the Scan I/O Register, operation of the scan circuitry (including the Scan Index Register) is enabled by the TBus handshake signals (cf. §3.7).

## 6.15 Scan Index Permutation Register

| 120 | SIPR | Scan Index Permutation Register |
|---|---|---|
| 5 | SSA0 | Select Site Address 0 |
| 5 | SSA1 | Select Site Address 1 |
| . | | |
| . | | |
| . | | |
| 5 | SSA23 | Select Site Address 23 |

As explained in §2.9, the order in which sites are accessed during a scan is derived from the Scan Index Register, which is incremented by one for each scanned site. When the 24 scan index bits are directly used in their natural order as site address bits, then the entire module is scanned as if by series of nested loops in which the first dimension corresponds to the innermost loops and the last to the outermost loop (cf. §2.4, §6.19).

However, this order can be significantly altered by permuting the scan index bits before using them as site address bits. The source of each of the 24 site address bits is specified by the value of a five-bit Select Site Address $n$

pointer[2] according to the following table

| SSA | site-address bit source |
|---|---|
| 0 | bit 0 of the Scan Index Register |
| 1 | bit 1 of the Scan Index Register |
| 2 | bit 2 of the Scan Index Register |
| ⋮ | ⋮ |
| 23 | bit 23 of the Scan Index Register |
| 24–29 | not used |
| 30 | 0 |
| 31 | 1 |

. (6.5)

Note that the constant values 0 and 1 are among the possible sources. This is important for scanning less than the whole space: in this case the unused address bits are set to a constant that specifies which part of the space is being scanned (cf. §2.7).

## 6.16 Scan I/O Register

| --- | SIOR | Scan I/O Register |
|---|---|---|

As mentioned in §3.10, the Scan I/O Register is used for accessing all or part of the DRAM chip connected to a given STEPchip, and does not have an intrinsic length. By setting up a scan of a certain size and shape (see §6.17), data may be read from or written to the scanned area through the Scan I/O Register.

Addressing the Scan I/O Register starts a scan[3] that is clocked by data transfer across the TBus (cf. §6.14). By appropriately setting the Source Registers (§6.4–§6.7), host-generated data may be integrated into this TBus-driven scan. This not only allows host data to be combined with current DRAM data, but, depending on the options selected, it also allows the host to do read/write operations that are not associated with scanned sites. For example, the host can write to the event counter (for testing), or put data on the video bus so as to download a new color palette to the CRT interface (cf. §??). In a similar way, a TBus read transaction with the Scan I/O Register makes a module scan (and possibly update) its data with a timing that takes into account the SBus's moment-by-moment availability to accept data. All Scan I/O read operations get their data from what is fed to the Display Data destination (cf. §6.8).

If the Verify Write Flag (§6.27) is set, any portion of the DRAM may of course be checked instead of written.

## 6.17 Scan Format Register

| 35 | SFR | Scan Format Register |
|---|---|---|
| 2 | SM | Scan Mode |
| 5 | ESC | End-of-Scan |
| 4 | ESW | End-of-Sweep |
| 4 | EST | End-of-Stretch |
| 5 | SBRC | Sweeps Between Refresh Cycles |
| 8 | RCL | Refresh Cycle Length |
| 5 | ECL | Event Counter Length |
| 2 | STM | Stretch Magnification |

The Scan Format Register is used, in conjunction with the Scan Index Permutation Register, to set the scan format.

The Scan Mode selects one of the four basic scan modes: mode 0 (open sweep, no row break), mode 1 (closed sweep, no row break), mode 2 (open sweep, 1 row break allowed), mode 3 (closed sweep, 1 row break allowed). These modes are explained in detail in §??–??. The software should select the fastest of these four modes of scan that is compatible with the selected permutation (allowing a row break adds overhead to the scan, even if no row break actually occurs). All chips operating in the same mode with the same format parameters are guaranteed to have the same scan timing.

The End-of-Scan holds the log (to base two) of the length of the scan: the number of updates that each module will perform without waiting for further instructions from the host (cf. §??). The hardware derives its end-of-scan signal by using this value as a pointer to a carry position in the Scan Index Register.

Similarly, End-of-Sweep determines the length of accesses that the hardware should assume lie within a single DRAM row, and End-of-Stretch determines the number of bits that the hardware can assume will be accessed from consecutive locations. As with End-of-Scan, the actual *End* signal is derived by interpreting these values as pointers to carry positions within the Scan Index Register.

Sweeps Between Refresh Cycles is the number of sweeps to perform between each refresh cycle; the Refresh Cycle Length indicates the number of refreshes to perform during each refresh cycle (refreshes are continuously performed when scanning isn't taking place).

The Event Counter Length specifies the effective length of the event counter, which may be less than 25 bits (see §6.10). This length will depend on the number of sites to be counted between readings, and typically is set whenever the scan size is changed.

The Stretch Magnification is used when the Scan Index Permutation Register is configured to achieve a readback or display with magnification (i.e., zoom). Up to a factor

[2] These registers actually allow one to have all possible *combinations with repetition*, of which permutations form a subset. In practice, however, permutations are all that we are generally interested in.

[3] At the moment of initiating a DRAM I/O operation via this register, the Scan Sync Mode (§6.1) will normally be found to be 0, indicating that no internally-clocked scan is in progress or scheduled; nevertheless, typically one will explicitly write to it before performing DRAM I/O, in order to transfer the event count from the previous scan into the Event Count Register.

of 4, this value is the log of the magnification – above 4 it should have a value of 3. The Stretch Magnification applies only to the component of the magnification that affects the stretch, and is needed by the DRAM address control circuitry to set up the nibble access pattern.

## 6.18 Offset Register

| | | |
|---|---|---|
| 24 | OSR | Offset Register |

As explained in §??, the Offset Register of a STEPchip records (modulo the size of each dimension) by how much the origin of the associated layer has shifted with respect to its initial reference position under the cumulative action of previous kicks.

The layer offset is added (separately for each dimension, as specified by the Dimension Cut Mask, §6.19) to the site address (which normally is the same for all layers) to yield the Physical Address (in DRAM) of the bit belonging to the currently scanned site and to the layer in question.

To implement the gluing algorithm (§??), this DRAM bit is made available to the data stream (§??) together with the homologous bits coming from the six adjacent modules. Which of these six bits is actually used depends on the $x$, $y$, and $z$ carries from the above addition, in conjunction with the corresponding kick signs (§6.2).

The Offset Register is not cleared by a soft (interface-initiated) STEPchip-reset. This is done to facilitate error recovery (see IR and DOCR).

## 6.19 Dimension Register

| 38 | DR | Dimension Register |
|---|---|---|
| 23 | DCM | Dimension Cut Mask |
| 5 | XDCP | X-Dimension Cut Pointer |
| 5 | YDCP | Y-Dimension Cut Pointer |
| 5 | ZDCP | Z-Dimension Cut Pointer |

The Dimension Register is used to indicate how the simulation space is shaped into a multidimensional array.

Specifically, the 23-bit Dimension Cut Mask is used to cut the site-address arithmetic chain into up to 24 slices, each corresponding to a distinct dimension (cf. §2.4).

The X-Dimension Cut Pointer specifies which of those dimensions is to be identified with the extendible dimension $x$. As indicated in the following table

| XDCP | X-Wrap source |
|---|---|
| 0 | carry from bit 0 |
| 1 | carry from bit 1 |
| 2 | carry from bit 2 |
| ⋮ | ⋮ |
| 23 | carry from bit 23 |
| 24–29 | not used |
| 30 | 0 |
| 31 | 1 |

, (6.6)

this pointer selects, as the source for the X-Wrap signal, the borrow (out of the corresponding slice) resulting from subtracting the $x$-component of the kick from the $x$-component of the site address. Values 30 and 31 of the Dimension Cut Pointers substitute for this borrow a constant value of 0 or 1; this is useful for testing, or for suppressing external gluing and allowing an extendible dimension to be wrapped around within the module (cf. §6.2).

Analogous considerations apply to the $y$ and $z$ dimensions and the associated Dimension Cut Pointers. Note that external gluing is controlled by the Kick Signs and Kick Max Flags, as well as by the Cut pointers.

## 6.20 Hardware Environment Register

| 16 | HER | Hardware Environment Register |
|---|---|---|
| 1 | LPL | LUT Pipeline Length |
| 1 | FPL | Flywheel Pipeline Length |
| 1 | DCS | DRAM Column Size |
| 6 | TBD | Tree Balancing Delay |
| 4 | TMS | Test Mode Select |
| 1 | NBF | Node Bypass Flag |
| 1 | SRE | Select Read Enable |
| 1 | ALS | Active LUT Select |

The Hardware Environment Register contains information about the hardware connected to the STEPchip: the size of the DRAM, the configuration of the TBus, pipelining information about the updating circuitry, etc.

If the LUT Pipeline Length is set, five bits of pipeline delay are expected between the SRAM address pins and the SRAM data pins (§10.2.2); otherwise, only one bit of delay is expected. What is normally connected between these two sets of pins is a SRAM lookup table which operates in a single clock cycle.

Similarly, if the Flywheel Pipeline Length is set, five bits of pipeline delay (rather than one) are expected in any hardware connected between the flywheel data output pins and the flywheel data input pins (§10.2.5).

The STEPchip can work with DRAMs with two column address sizes: 10 bits or 11 bits. 4 Mbit, nibble-wide DRAMs have a 10 bit column address and a 10 bit row address (10+10); 16 Meg DRAMs come in both 10+12 and 11+11. The DCS bit is used to tell the chip the width of the column address, according to the following table

| DCS | column address width |
|---|---|
| 0 | 10 |
| 1 | 11 |

. (6.7)

In the 10-bit mode, for chip test purposes, the two unused column-address lines carry the two least significant bit of the DRAM address, i.e., the address bits that select a bit within a nibble.

If the TBus tree is not implemented as a balanced tree (cf. §3.3), bus data will reach certain modules earlier than others. The Tree Balancing Delay specifies how much closer the given module is to the root than the farthest module in the tree. Since modules that are closer to the root receive their commands earlier (by one clock cycle per tree segment), a corresponding delay is inserted by the STEPchip hardware before the start of a sweep (cf. §6.1) so as to allow all modules to sweep in lockstep. Note that this delay is not inserted during scan-io (§6.16), and so no use of glue should occur during this mode, since modules will not be scanned in lockstep.

Test Mode Select is used to put the STEPchip into special test modes. Each of the 16 TMS values selects not only a test mode, but also a single output value that can be made available through Multipurpose Pins (cf. §6.21). The choices are **[is this table final?]**

| TMS | test mode | signal name |
|---|---|---|
| 0 | none | |
| 1 | none | End Of Stretch Pulse |
| 2 | none | End Of Sweep Pulse |
| 3 | none | End Of Scan Pulse |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |
| 11 | | |
| 12 | | |
| 13 | | |
| 14 | | |
| 15 | | |

(6.8)

A value of 1 for the Node Bypass Flag indicates that TBus signals and data entering the node on this chip from the inner (host) side should be used directly by the I/O circuitry, *without* first passing through the node. A value of 0 for this flag indicates that these bus signals are first reclocked by the node, and only then used (by the rest of the chip) *after* passing through the node. Use of this flag (provided bus timing constraints allow it) makes it easier to construct a balanced version of the TBus.

The Select Read Enable bit is used when register 0, the Module Select Register, is read. On such a read, the current selection state of the STEPchip is ignored, and only STEPchips that have the Select Read Enable bit set will respond (see §6.0).

According to its value, 0 or 1, the Active LUT Select bit tells which of the two LUTs (0 or 1) is currently active (cf. Active LUT Toggle in the Run Mode Register, §6.1). Writing this bit forces a specific LUT to become the active one.

## 6.21 Multipurpose Pin Control Register

| 10 | MPCR | Multipurpose Pin Control Register |
|---|---|---|
| 5 | MAFS | Multipurpose A Function Select |
| 5 | MBFS | Multipurpose B Function Select |

The Multipurpose pins (§10.2.7) are used for input or output signals that need appear only once per module rather than once per STEPchip. Each chip of a set will handle two different signals through its two Multipurpose pins, as instructed by the Multipurpose Pin Control Register. We provide two of these pins per chip so that even a module with a small number of STEPchips will have enough multipurpose pins to provide all essential signals. This is convenient for constructing low-budget STEP machines, and also for reducing the number of STEPchips needed to perform a complete simulation or test of a working system.

There are 32 choices for the function of each of the Multipurpose Pins. For simplicity, we have exactly the same choices available on both pins: the first four choices use the pin for input, the rest of the choices are outputs. The four input choices are:

| MFS | Signal |
|---|---|
| 0 | External Input |
| 1 | External Interrupt |
| 2 | Scan Input |
| 3 | Status Input |

(6.9)

The External Input allows an external signal to be delayed or inverted and come out on the other Multipurpose Pin, if it is suitably configured as an output; the External Interrupt allows external hardware connected to the module to signal an interrupt (see IER); the Scan Input makes it possible to check for intermodule scan-synchronization errors (see IER); and the Status Input (coming from a module farther out on the bus tree) is combined with the internally generated status signal to produce the Status output (see below).

The rest of the Multipurpose Function Select values indicate that the corresponding Multipurpose Pin is configured as an output. Each STEPchip generates all of the module signals that might be needed; which signal is ouput to the pin is established by the Multipurpose Function Select according to the following table.

| MFS | Signal |
|---|---|
| 4 | Status (Int/Scan) |
| 5 | Node LSOUT |
| 6 | Node IOSEL |
| 7 | Node RSOUT |
| 8 | Box Data Enable |
| 9 | SYNC (delayed) |
| 10 | Node Data Direction |
| 11 | Node Data Enable |
| 12 | Scan Is Active |
| 13 | Latch Glue Direction |
| 14 | LUT0 Chip Select |
| 15 | LUT0 Write Enable |
| 16 | LUT1 Chip Select |
| 17 | LUT1 Write Enable |
| 18 | Site Output Data Valid |
| 19 | Unglued Data Bit |
| 20 | Host Supplied Bit |
| 21 | SABSR Selected Bit |
| 22 | Display Output Data Valid |
| 23 | Run Type |
| 24 | Active LUT Select |
| 25 | Active LUT Output |
| 26 | CAM Interrupt |
| 27 | Scan In Progress |
| 28 | External Input Inverted |
| 29 | External Input Delayed |
| 30 | Constant of 0 |
| 31 | Constant of 1 |

(6.10)

Choices 4–11 are TBus signals that are needed only once per module; they are documented in §3. Scan Is Active (choice 12) can be used in conjunction with Scan Input (choice 2) from another module to test that periods of active scanning occur at the same time in both modules (for this test, the scan is *not* considered to be active during Scan I/O, cf. §6.16). Latch Glue Direction (choice 13) is a pulse (intended for use by external glue signal buffers) that occurs just before active scanning begins; at this time, glue lines are either pulled up to indicate inputs, or actively driven low to indicate outputs. Choices 14–17 control the SRAMs that constitute the two LUTs in a module. Data Valid (choice 18) is used in conjunction with choices 19–21, and also the LUT Address and the Flywheel Output to mark valid data bits. Display Valid (choice 22) has a slightly later timing, and is used to indicate when the Display Output has valid data. This timing is the same as that of the active LUT's outputs, and the Flywheel input (except when used as an input to LUT and FLY). Run Type (choice 23) reflects the value of the third bit of the Run Mode Register; it is used in conjunction with choice 22 to indicate valid display output that occurs during a run of a particular type – for example, one that is meant to be sampled by a frame buffer. Similarly, Active LUT Select (choice 24) reflects the value of the last bit of the Hardware Environment Register, which indicates which of the two available LUTs is currently active for updating. Active LUT Output (choice 25) is a reclocked version of the actual output bit from the currently active LUT. CAM Interrupt and Scan Requested (choices 26 and 27) repeat the same information encoded in the Status signal (choice 4), but as two separate signals. External Input Inverted (choice 28) reflects the value (inverted) of an External Input (choice 0) if such an input was selected at the other Multipurpose Pin of this STEPchip; similarly, External Input Delayed (choice 29) reflects an External Input that has been clocked through a single delay element.The Test MUX Output (choice 29) is an output that is determined by the current test mode (see Test Mode Select in §6.20). Finally, constants of 0 and 1 can be selected as choices 30 and 31 respectively.

At reset time, the Multipurpose pins are always configured as inputs (choice 0), and the Module ID Register is set from Multipurpose Pin A. This is used to distinguish one STEP module from all of the others in a given box (§7.7), using pullups and pulldowns. After reset, in an unbalanced tree, all boxes except the one at the root (corner) of the tree are cut off from the TBus until the root's Multipurpose Pins are configured.[4] This distinguishes the root box from all other boxes, and is useful in establishing a distinguished module to use as the origin for numbering the rest (i.e., the distinguished module in the corner box).

[4] The point in a TBus frame at which the new value of the Multipurpose Pin selections become enabled is chosen to make enabling the TBus work properly.

## 6.22 Glue Pin Connectivity Register

| 18 | GPCR | Glue Pin Connectivity Register |
|---|---|---|
| 3 | XMPC | XM-Pin Connect |
| 3 | XPPC | XP-Pin Connect |
| 3 | YMPC | YM-Pin Connect |
| 3 | YPPC | YP-Pin Connect |
| 3 | ZMPC | ZM-Pin Connect |
| 3 | ZPPC | ZP-Pin Connect |

The Glue Pin Connectivity Register is written at setup time in order to determine what logical interconnection should be associated with the physical interconnection provided by each of the six glue pins (§10.2.4).

One use of this configuration flexibility is described in §7.8.2. More drastic reconfigurations are possible. Normally the glue pins define a three-dimensional array of modules, interconnected as described in Chapter 2.5. However, by appropriately arranging the connectivity between modules, one can connect them as a two-dimensional array (physically folded back-and-forth over itself, with the folds piling up in the third dimension), or even as a one-dimensional chain (§??).

**[make sure the terminology agrees]** Each of the six connectivity registers corresponds to a *physical* glue port (STEPchip pins XM, XP, YM, YP, ZM, ZP) and associates with it a *logical* glue port ($x_-$, $x_+$, $y_-$, $y_+$, $z_-$, $z_+$, as explained in §??) or one of the constants 0 and 1.

Whenever the kick sign for dimension $x$ is positive, thus indicating that logical port $x_+$ should be an *output* (cf. §??), then any physical port that has been associated with $x_+$ by the Glue Pin Connectivity Register will be configured as an output and will display the value of $x_+$; if more than one physical port is associated with the same logical port, they will all behave alike.

When the kick sign is negative, thus indicating that logical port $x_+$ should be an *input*, then any physical port associated with $x_+$ will be configured as an input and will contribute a source of data for $x_+$: if more than one physical port is associated with $x_+$, their values will be ORed before being passed on to $x_+$.

Finally, if a physical port is associated with a constant, then it will always be an output and display that constant. The constants are used during initialization for determining the physical connectivity of the machine and initializing each module's ID (§3.14).

**[Read and understand; maybe add a diagram]** The value of XM-Pin Connect determines the logical glue ports (or the constant) associated with the corresponding to physical port XM according to the following table (and similarly for the other five ports)

| XMPC octal | XMPC binary | logical port or constant |
|---|---|---|
| 0 | 00 0 | $x_-$ |
| 1 | 00 1 | $x_+$ |
| 2 | 01 0 | $y_-$ |
| 3 | 01 1 | $y_+$ |
| 4 | 10 0 | $z_-$ |
| 5 | 10 1 | $z_+$ |
| 6 | 11 0 | 0 |
| 7 | 11 1 | 1 |

(6.11)

Each glue pin is internally pulled up by a resistor to a logical 1, so that, when configured as an input and not otherwise driven, it will sense a logical 1; this feature is used during module ID initialization (§3.14).

Any glue pin which is not configured as a constant will act as an input whenever the scan is not active.

## 6.23 Module ID Register

| 1 | MIDR | Module ID Register |
|---|---|---|

Each STEP module has a unique *module ID*, which is primarily used for module selection (§3.13); this ID is assigned during hardware initialization (§3.14). Each of the 16 STEPchips of a set holds one bit of the module ID, in the Module ID Register (MIDR).

At reset, the MIDR is set to the value of the Multipurpose Pin A (see §3.14 for details).

## 6.24 Group ID Register

| 1 | GIDR | Group ID Register |
|---|---|---|

Besides the module ID, which is unique for each chip (§6.23), each STEP module has also a *group ID*, which is primarily used for module selection (see §3.13) when one wants to address a whole group of modules rather than a single one (see §3.13). Each of the 16 STEPchips of a set holds one bit of the group ID, in the Group ID Register (GIDR).

Note that by using the *don't care* selection code (see table (6.1)) in some positions of the Module Select Register (§??), one can select different groups of modules based on data contained in different portions of the GIDR.

## 6.25 Interrupt Enable Register

| 10 | IER | Interrupt Enable Register |
|---|---|---|
| 1 | BPIE | Bus Parity Interrupt Enable |
| 1 | BCIE | Bus Contention Interrupt Enable |
| 1 | GCIE | Glue Contention Interrupt Enable |
| 1 | MAIE | Multi-A Contention Interrupt Enable |
| 1 | MBIE | Multi-B Contention Interrupt Enable |
| 1 | SSIE | Skewed Scan Interrupt Enable |
| 1 | XHIE | External Hardware Interrupt Enable |
| 1 | RLIE | Register Length Interrupt Enable |
| 1 | URIE | Unknown Register Interrupt Enable |
| 1 | ISIE | Invalid SYNC Interrupt Enable |

The Interrupt Enable Register is a collection of bits that individually enable the interrupts described in the following section, devoted to the Interrupt Flags Register. An interrupt will be recognized only if the corresponding IER bit is set. Since these interrupts all signal error conditions, their enables should normally all be on. Even in chips which don't have External Hardware or Scan Input attached, the corresponding interrupts should not signal an error, and so can be left enabled.

An additional interrupt enable flag, associated with the verify-write mode, is discussed in §6.27.

## 6.26 Interrupt Flags Register

| 10 | IFR | Interrupt Flags Register |
|---|---|---|
| 1 | BPIF | Bus Parity Interrupt Flag |
| 1 | BCIF | Bus Contention Interrupt Flag |
| 1 | GCIF | Glue Contention Interrupt Flag |
| 1 | MAIF | Multi-A Contention Interrupt Flag |
| 1 | MBIF | Multi-B Contention Interrupt Flag |
| 1 | SSIF | Skewed Scan Interrupt Flag |
| 1 | XHIF | External Hardware Interrupt Flag |
| 1 | RLIF | Register Length Interrupt Flag |
| 1 | URIF | Unknown Register Interrupt Flag |
| 1 | ISIF | Invalid SYNC Interrupt Flag |

The Interrupt Flags Register is a collection of one bit flags, each of which is set when the corresponding error condition is detected (see §?? for details).

When an interrupt condition is detected by the STEPchip hardware, if the corresponding enable flag is set (§6.25) an interrupt will be sent via the TBus to the STEP interface, which in turn will interrupt the host. Once the interrupt reaches the interface, further interface operation is suspended until the host restarts it.

An additional interrupt flag, associated with the verify-write mode, is discussed in §6.27.

The contention interrupts indicate that a signal sampled at the output of a bussed or bidirectional pin did not agree with the value that was being driven.

A Bus Parity error is indicated if the last completed TBus transaction didn't have overall odd parity, including the final Parity bit that was sent.

A Skewed Scan interrupt is signalled if the Scan Input is used, and disagrees with the internally generated Scan is Active signal.

An External Hardware Interrupt is signalled if the External Interrupt Input is used, and it goes low.

A Register Length Interrupt is signalled if a Register is written by the host with an incorrect length, or if a register is read by the host with too short a length.

An Unkown Register Interrupt occurs if an attempt is made by the host to access registers 29, 30, or 31.

An Invalid SYNC Interrupt is signalled when one of the 4-bit packets of video-SYNC timing information (see §6.1) coming in on the SYNC input of the STEPchip (cf. §??) fails to have even parity.

Note: the Interrupt Flags Register is not cleared by a soft (interface-initiated) STEPchip-reset. This is done to facilitate error recovery (see OSR and DOCR).

## 6.27 Verify Write Register

| 3 | VWR | Verify Write Register |
|---|---|---|
| 1 | VWE | Verify Write Enable |
| 1 | VWIE | Verify Write Interrupt Enable |
| 1 | VWIF | Verify Write Interrupt Flag |

With an ordinary write frame, the transmitted data overwrite the contents of the addressed register (or the contents of the addressed DRAM or SRAM area, if the register is an I/O register). The Verify Write Register allows one to control a special *verify-write* mode, where the transmitted data are simply compared with this contents without changing it. This mode is enabled by turning on the Verify Write Enable. Verify write is always disabled while writing to the VWE register itself, to make it possible to turn the mode on and off.

In verify-write mode, an interrupt condition is signaled—by setting the Verify Write Interrupt Flag—if data written to a register disagree with its contents. Verify-write errors are only signaled at the *end* of a data transfer (see the discussion of data transfer errors in §??).

Setting the Verify Write Interrupt Enable allows the interrupt condition to generate an actual interrupt. Enabling the verify-write mode on a given STEPchip without enabling the corresponding interrupt provides a way of *disabling* the writing of data on that chip. In this way, one can "write-protect" one or more of the sixteen layers, to change registers on some chips in the module and not on others. Note that this *write-per-bit* technique cannot be used with the LUTs.

## 6.28 DRAM Ones Count Register

| | | |
|---|---|---|
| 16 | DOCR | DRAM Ones Count Register |
| 8 | LDOC | Low DRAM Ones Count |
| 8 | HDOC | High DRAM Ones Count |

A DRAM *ones counter* is an up/down counter that is cleared, together with the DRAM, at initialization time. From then on, each time a 0 in the DRAM is changed to a 1, the counter is incremented; each time 1 goes to 0, decremented; otherwise, it remains unchanged. Thus the counter should always reflect the number of ones in the DRAM (modulo the counter's size).

By using the event counter to occasionally count all locations in the DRAM, and comparing this count with that recoreded by the ones counter, it should be possible to detect (with a high probability) soft errors in the DRAM.

Each STEPchip has one DRAM chip attached to it. The DRAM Ones Count Register consists of two independent DRAM Ones Counts (LDOC and HDOC) that are associated repectively with the low and high halves of this DRAM.

By having separate counts for the two halves of the DRAM, it becomes possible to effectively handle errors on a large machine. We split the machine into two halves, each employing half of each DRAM chip, and both performing the same computation. In the case of a miscompare between the operation of the two halves, the DRAM Ones Counts make it likely that we can resolve which of the two halves got an error, and then correct the errorneous data from the other half.

The DRAM Ones Count Register is not cleared by a soft (interface-initiated) STEPchip-reset. This is done to facilitate error recovery (cf. OSR (§6.18) and IR (§??)).

# 7 Step machine and controller

## 7.1 Overview

A complete STEP system consists of a three-dimensional array of STEP *boxes* driven by a *host* via a tree-shaped network called the *TBus*, as sketched in Fig. 7.1 (where only two dimensions of the array are shown).

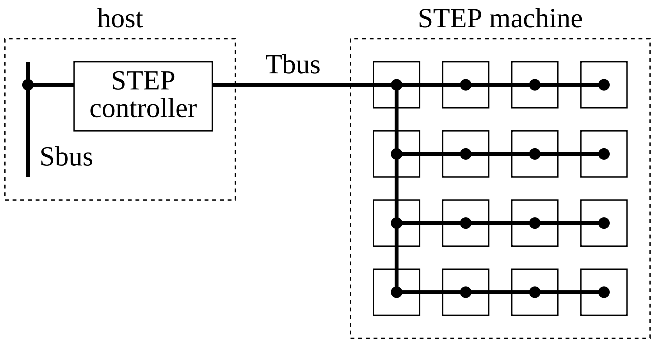



Figure 7.1: A STEP system consists of an array of STEP boxes (STEP *machine*) driven by a SPARCstation (*host*). The STEP *controller* interfaces the host's Sbus with the machine's TBus.

In the current implementation, the host is a SUN SPARCstation, which features a high-performance external bus called the SBus. The TBus is driven at the root by the STEP *controller*—a small custom board plugged into one of the SBus slots. From this board, a short cable—which represents the TBus *trunk*—leads directly to one of the boxes, located on a corner of the array. From this *corner box* the TBus branches out through the rest of the array. We shall call STEP *machine* the interconnected array of STEP boxes, when viewed as distinct from the host and the STEP controller.

## 7.2 Hardware hierarchy

Each box of the STEP machine array realizes, by means of eight identical printed-circuit boards called *modules*, a functional *block* of eight sectors (see Chapter 2.5). Besides, the box provides mechanical and electrical support (motherboard, enclosure, power supply and cooling) to the modules themselves, as well as TBus routing and buffering and some other support circuitry.

In turn, each module contains, besides conventional DRAM and SRAM chips, sixteen copies of the STEP*chip*—a custom VLSI chip that one may view as embodying the STEP architecture. In fact, each STEPchip has complete control over one DRAM chip's data and control lines; and, collectively, the sixteen STEPchips have complete control over the SRAM chips' data and control lines—as well as over transactions over the TBus. Thus, the STEPchips repesent the "glue" that turns plain electronic commodities into sophisticated computing resources.

The present Part II discusses the STEP system at four levels of the design hierarchy, namely,

1. the STEP machine with its controller (this chapter),
2. the STEP box (Chapter 8),
3. the STEP module (Chapter 9), and
4. the STEPchip (Chapter 10).

For each level, formal interface specifications (pinout and signal description) with the level above it are given.

The STEPchip (Level 4) is seen by the programmer as a set of registers accessible via the TBus. Part III expands Chapter 10 by giving a complete programmer's model of the STEPchip: it systematically describes structure and functions of these registers, giving, whenever appropriate, a functional model of the hardware controlled by the registers themselves.

One may view the host SPARCstation as representing Level 0 of the hierarchy. Its relevant hardware and system-software aspects are documented by SUN Microsystems. STEP-specific software is documented in the STEP *Programmer's guide* and *User's guide*.

## 7.3 The step controller

The STEP controller is a small board with, on one edge, a connector that plugs into one of the SBus slots. At the opposite edge of the card, a connector that shows through the rear of the SPARCstation represents the root of the TBus.

The main *raison d'être* of this piece of hardware is to provide an interface between the two buses, and to do so in such a way as to lessen the requirements for real-time response from the host.

While the SBus and the TBus are both synchronous, and at present have roughly comparable bandwidths, they are timed by independent clocks which are typically running at different rates. Moreover, the number and nature of the control and data lines, the protocols, and the signals' electrical specifications are quite different in the two buses.

The STEP controller requires bus-master as well as bus-slave capabilities.[1] In fact, in order to decouple STEP from the poor and rather impredictable real-time response of standard UNIX, the controller's normal operating mode is to act as a bus master and interpret according to the STEP machine's timing requirements a list of commands (STEP*list*) directly off the SPARCstation's memory, without going through the CPU. These commands would previously have been written to memory by the CPU at its own leisure.

The controller is addressed as a slave only at initialization time or to recover from runaway situations.

Depending on specific implementation choices, clocking for the STEP machine may be generated by the controller or by the machine itself. (A clock line running along the TBus trunk will carry a clock reference signal from the controller to the machine or vice versa.) In either case

- The interface between circuitry clocked by the SBus and circuitry clocked by the TBus lies within the controller, which has provisions for arbitrating between the two clocks.
- Tight synchronism (within a small fraction of the clock period) between logically contiguous modules is achieved by local phase-locked loops.

See §?? for details of clock distribution and synchronization.

The principles of operation of the STEP controller (i.e., how it interprets a "script" consisting of STEPlist commands, initiating and supervising any SBus or TBus transactions that may be called for) are explained in Chapter 4.

### 7.3.1 SBus signals

The STEP controller interfaces with the SBus by means of a standard SBus connector. Signal description and pinout can be found in *The SBus Specification* (specifically, Tables 2-1 and 3-8 of that manual). The following SBus signals are not used by the controller: **[Any unused signals?]**

### 7.3.2 TBus signals

The TBus runs from the STEP controller to the corner box (§7.7) and thence outwards from box to box (cf. Fig. 7.1 and Fig. 7.4) on short lengths of cable called *hops*. Each hop consists of a bundle of tranmission lines (cf. §7.6) carrying the following signals

TBus signals

| direction | name | |
|---|---|---|
| bidirectional | Data[0:F] | Data[0:F] |
| outbound | Reset* | Reset* |
| outbound | Sync | Sync |
| outbound | Frame* | Frame* |
| outbound | OH | Outbound Handshake |
| inbound | IH | Inbound Handshake |
| inbound | Status* | Status* |
| outbound | Scan* | Scan* |

(7.1)

The hop that runs from the STEP controller to the corner box is called the TBus *trunk*.

Within a box, the TBus is further distributed by the motherboard to the eight modules according to the pinout of table (9.2), and from there to the sixteen chips of each module according to the pinout of table (10.6).

The various TBus lines are discussed in detail in Chapter 3.

The inner end of a hop (the one closer to the tree root) plugs into one of a box's three TBus branch connectors (that's where a new TBus branch is grafted onto the tree); the outer end plugs into a box's TBus root connector (that's where the box faces towards the tree root). According to this terminology, the connector through which the STEP controller drives the TBus trunk is a TBus branch connector. Its pinout is the following

TBus branch connector

| type | name | |
|---|---|---|
| B2 | Data[0:F] | Data[0:F] |
| O2 | Reset* | Reset* |
| O2 | Sync* | Sync* |
| O2 | Frame* | Frame* |
| O2 | OH | Outbound Handshake |
| I2 | IH | Inbound Handshake |
| I2 | Status* | Status* |
| O2 | Scan* | Scan* |

(7.2)

In the TBus trunk (§7.3.2), the Scan* line carries master-clock (§??) rather than scan-in-progress (§??) information.

### 7.3.3 TBus Sync

The TBus Sync line carries, encoded as 5-bit bursts on a background of all zeros, messages used for achieving synchronization between sweeps and scans of the STEP machine (§??) and, respectively, horizontal and vertical scans of a CRT monitor (or similar external equipment). The possible messages are tabulated in (??).

[1] On the SPARCstation-1, one of the three SBus slots only offers bus-slave capabilities, and cannot be used for this purpose. At any rate, this slot is typically dedicated to the graphics interface.

| code | message | recipient |
|---|---|---|
| 1 00 1 0 | HMON | CRT monitor |
| 1 01 0 0 | VMON | CRT monitor |
| 1 10 0 0 | HCAM | STEP machine |
| 1 11 1 0 | VCAM | STEP machine |

Figure 7.2: The above codes, transmitted on a background of all zeroes, signal the appropriate times for a CRT monitor to start horizontal and vertical scans, and for the STEP machine to start sweeps and scans. Each message consist of a start bit ('1'), a two-bit field carrying the intelligence of the message, a parity bit, and a stop bit ('0').

The above messages are generated on the STEP controller as follows. Two ganged programmable counters, namely, the *horizontal counter* and the *vertical counter* (the latter is clocked by the rising edge of the former), produce the waveforms of Fig. 7.3. For each counter, "mark" and "space" lengths are independently programmable; as a consequence, one can independently set the period and the duty-cycle. HCAM, VCAM, HMON, and VMON are issued at the rising and falling edges of these waveforms, as indicated in Fig. 7.3.

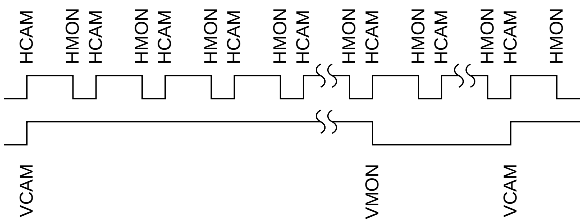



Figure 7.3: Sync messages are issued on the rising and falling edges of the waveforms generated by the horizontal and vertical counters.

The HCAM message that would occur concurrently with a VCAM is superseded by the latter; the STEPchip will infer an HCAM from the presence of a VCAM.

While the coincidence of VCAM with an HCAM is a logical necessity, the fact that also VMON exactly coincides with an HCAM (and supersedes the latter) is a nonessential feature of the present version of the STEP controller.[2] Since the STEPchip does not respond to HMON and VMON messages,[3] a STEP scan that extends past VMON and is synchronized to HCAM (cf. §6.1) will be suspended for one horizontal counter period when VMON occurs.

## 7.4 The step machine

The STEP machine is an assembly of STEP *boxes* physically arranged and interconnected as a three-dimensional array. The three dimensions are labeled $x$, $y$, and $z$, and along each dimension the elements of the box array are sequentially numbered from 0 upwards. Thus, the *corner box* (§7.1), located at the origin of the box array, has coordinates $\langle 0,0,0\rangle$, and in a system of $2\times3\times5$ boxes the box on the opposite corner would have coordinates $\langle 1,2,4\rangle$.

Along any of the three dimensions, the array need not extend for more than one box. In particular, machines

[2]Note that the STEP controller could be replaced by, say, a video camera as a source of sync messages, in order to have video-on-video.

[3]Each STEPchip monitors the integrity of all sync messages, by checking the parity bit, but otherwise ignores messages directed to the CRT monitor.

consisting of a single box are viable (and we expect that they will be in quite common use).

A box interfaces to the outside world through three groups of signals, namely *glue* signals and *TBus* signals, which travel between boxes on transmission lines (§7.6) and turn the collection of boxes into a tightly-knit whole, and *data stream* signals, which offer optional access to the box's internal data stream. In addition, there are a small number of signals used for configuration and test purposes. All these signals are described in detail in Chapters ?? and ??

The glue signals exchange, in real time, site data between adjacent sectors, according to the current data-transport specifications for each layer (cf. §2.10). Six glue ports are provided—each port corresponding to one of the six "faces" of the box, i.e., the six spatial directions in which a box can interconnect with other boxes, denoted by XM ("x minus"), XP ("x plus"), and similarly YM, YP, and ZM, ZP. Besides the data lines, each glue port includes two lines that exchange clock-phase information between adjacent boxes (§7.5).

The TBus signals ultimately connect the boxes to the host. Each box has one *root* port, facing inside (that is, towards the host), and three *branch* ports, facing outside in the positive $x$, $y$, and $z$ directions. The root port is always used: the one on the corner box connects directly to the TBus trunk, the others, to bus hops that come from intermediate boxes. On the other hand, a branch port is used only if the box is the root of a subtree . Thus, for example, in a machine of size 2×2×2 boxes (see Fig. 7.4), the corner box $\langle 0,0,0\rangle$ uses all three branch ports, box $\langle 0,0,1\rangle$ uses only $x$ and $y$, boxes $\langle 0,1,0\rangle$ and $\langle 0,1,1\rangle$ use only $x$, and the remaining four boxes, with addresses of the form $\langle 1,y,z\rangle$, use none, since they are at the leaves of the tree.

Unlike glue signals and TBus signals, the data stream signals that appear on each box do not connect to other boxes; in fact, they do not need to be connected to anything in particular for the STEP machine to operate properly.

Through the Flywheel ports, external equipment can selectively extract any of the data involved in the simulation and optionally replace them with arbitrary new data; these lines are thus used for something like "local open-brain surgery" (it is unlikely that even the most advanced user will connect external equipment to more than one box).

Through the Video port, one can send to a color monitor and view in real time the contents of a selected STEP sector.

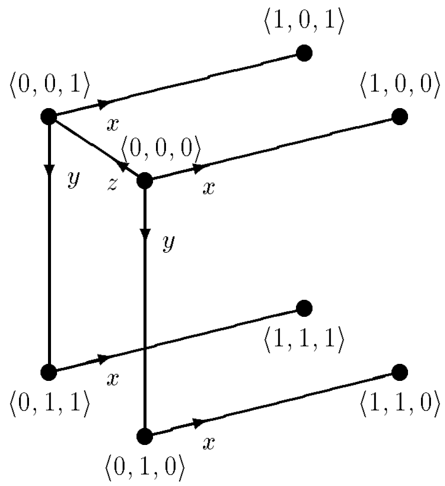



Figure 7.4: TBus port usage in a 2×2×2 box machine. The dots represent boxes; the arcs, TBus hops. The $x$, $y$, $z$ labels indicate to which branch port of a box the inner end of a hop is connected.

## 7.5 Clock distribution and synchronization

The current STEP implementation consists entirely of *synchronous* digital circuitry. Each box has its own *local master clock*, which drives the clock distribution tree for the entire box. The issue discussed here is the mutual synchronization of the local masters within prescribed tolerances.

Though the STEP array may extend arbitrarily in all three dimensions, its elements (the boxes, in the present context) are only *locally* interconnected. Thus, all that matters is that the clock phase difference between a box and each of its six neighbors (the ones with which the box exchanges *glue* data) be within tolerances. There will be no harm if, in moving from one box to the next along a given direction, these differences add up—leading perhaps to a cumulative phase difference of several clock cycles between widely separated boxes.

In STEP, local master clocks in adjacent boxes keep within tolerances with respect to one another by exchanging clock-phase information. This provides the feedback by which the phase of each individual clock is dynamically adjusted according to a *phase-locked loop* scheme: i.e., this phase is continually made to track (according to a suitable response function) the mean of the phases of its six neighbors.

**[Revise and partially scrap this]** Usually, at the heart of a phase-locked loop there is a variable-frequency oscillator (VFO). The phase difference between the oscillator and the reference signal is used to pull the oscillator's frequency in such a direction as to reduce this difference. However, since the phase is proportional to the integral of the frequency rather than the frequency itself, one obtains in this way a *second-order* loop; such a loop imposes certain trade-offs between stability and agility of response.[4] On the other hand, suppose that a signal having the desired *frequency* were distributed to all boxes, reaching each box with a different phase (owing to propagation delays through the distribution network). Then the secondary master could match its phase to that of its neighbors by means of a *first-order* loop where the phase is adjusted directly (rather than indirectly through the frequency), with improved response characteristics. Alternatively, the distributed reference frequency could be used to adjust the VFO natural frequency, so as to minimize the amount of error feedback needed to pull the oscillator into phase lock.

Though the details of the local masters' strategy for achieving phase lock are left unspecified, a single reference frequency may be made available to all boxes through a TBus Master Clock signal originating from the STEP controller. This signal could be used directly as the local master clock by the corner box, which would ignore phase feedback from its neighbors and would thus force them to do all the necessary adjusting. By directly using a copy of the Master Clock as a local master, "bare bones" boxes designed to work only in small machines (one to eight boxes) could dispense with the phase-locked loop altogether.

[4] Intuitively, in a second-order loop the phase has *inertia* that needs to be damped. If is overdamped, it will never reach the mark; if it is underdamped, it will overshhot it.

It may be decided at a later time to use the oscillator of the corner box as the primary master. In this case, on the TBus trunk the *Master Clock* signal would run from the corner box to the controller (rather than the other way around).

## 7.6 Interbox communication

Within a box, signals run short distances; they have a stable and uniform voltage reference provided by the modules' and the motherboard's ground plane; they are shielded from one another by the close proximity of the ground plane, and from external fields by the box's enclosure. In this environment, a signal line, even though it may extend and branch out over several inches, can still be usefully treated as a voltage node.

Communication between boxes takes place in a rather different environment. The physical length of the lines from driver to receiver may run up to a few feet; glue lines run in parallel bundles of over a hundred lines each;[5] each box is the source or destination of about a thousand 30-Mbit/sec signals; and boxes will typically be tied together by a mesh of ground bars or ground strips several inches in length rather than by a continuous copper plane. For these reasons, it is desirable to have interbox signals travel on well-characterized *transmission lines*, which feature low dispersion and (when properly terminated) low reflections, good common-mode noise rejection, and good decoupling between adjacent signals.

Though implementation details for transmission-lines, line-drivers, and line-receivers are left, for the moment, unspecified, we shall assume that each transmission line consists of a *pair* of wires (such as the two elements of a twisted-pair line, or core and shield in a coaxial cable). Accordingly, for each signal handled in this way we shall provide two separate pins on the connectors where the transmission lines terminate (cf. §9.1).

## 7.7 Corner box

Because of its unique position—at the origin of the STEP box array and at the origin of the TBus distribution tree—the STEP machine's corner box plays a distinguished role in certain setup and housekeeping functions.

The STEP modules come out of power-up with all activity on their *TBus branch* ports disabled (see §6.21);

[5] Sixteen layers times eight strands gives 128 lines, to which one must add two clock-phase feedback lines.

thus, while the controller can communicate with the corner box, all other boxes are cut off from the TBus for the time being. Moreover, module 0 of a box is physically distinguished from the other modules of the same box by having the Multipurpose Pin A of STEPchip 0 pulled down by an external resistor, so that its module ID at power up will be FFFE rather than FFFF.[**Norm, confirm this**]

Taking advantage of the above features, the host runs an initialization procedure that systematically assigns module IDs, starting with 0000 for module 0 of box 0 (cf. §3.14). Once the corner box's privileged geographical position has been recorded in distingushed IDs for its modules, the other boxes are gradually brought into communication with the TBus, proceeding outwards from the corner box.

The local master clock of the corner box ignores clock-phase information coming from the adjacent boxes (cf. §??), and thus forces the latter to match their phases with its own.

If a reference clock signal is distributed along the TBus (cf. end of §7.6), the corner box may send rather than receive this signal on its TBus root port, thus clocking the STEP controller rather than being clocked by it.

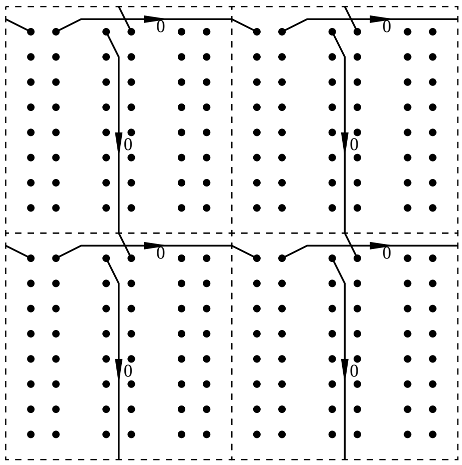


Figure 7.6: Backplane wiring for an indefinitely extended array of STEP boxes, with no wraparound. Four boxes are shown, and only the wiring for strand 0. The arrow direction goes from a '+' port to a '−' port (cf. table (2.2)).

## 7.8 Block wiring

In the present realization, a STEP module consists of a *board* having all of its connectors on one edge, as in Fig. 7.5, and a STEP block will consist of a *box* containing a card cage with a backplane where eight of these boards can be plugged in.

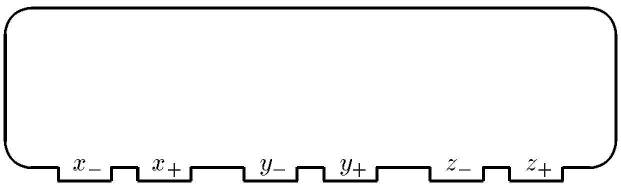



Figure 7.5: STEP module board, showing the six glue port connectors.

For such a realization of a STEP block, table (2.2) can be interpreted as a template for the backplane, indicating the eight slots (horizontal rows) and, for each slot, the position of the six glue-port connectors.

Using this template, Fig. 7.6 shows in its essentials the actual wiring between boxes. We only show the wiring for strand 0, and then only for the $x$ and $y$ dimensions (the wiring for the $z$ dimension would go "through the page").

Still using template (2.2), Fig. 7.7 shows the actual placement of the shorting plugs (cf. Fig. 2.14) at the edges of the array (using an array that is two blocks deep in

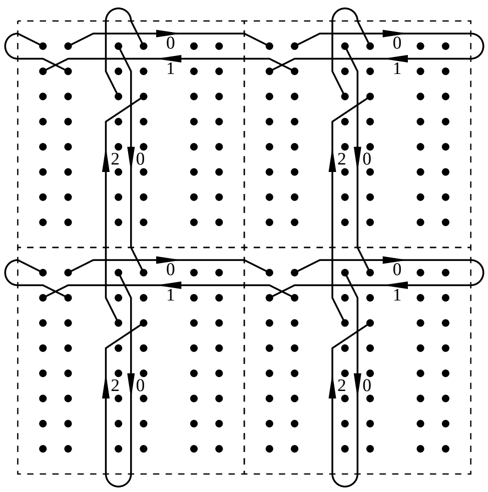
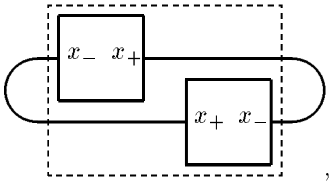


Figure 7.7: How strands are connected at the edges of the array to form loops.

both the $x$ and $y$ dimensions; as in Fig. 7.6, the $z$ interconnections are not shown). With these plugs, strand 0 forms a single loop with strand 1 in the $x$ dimension, and with strand 2 in the $y$ dimensions (in the $z$ dimension, it would loop back as strand 4).

### 7.8.1 Single-block wiring

For a small STEP system consisting of just one block (eight modules), each dimension exhausts itself within the block, as indicated by the following diagram for just the $x$ dimension




,

and the complete wiring diagram will appear as in Fig. 7.8.

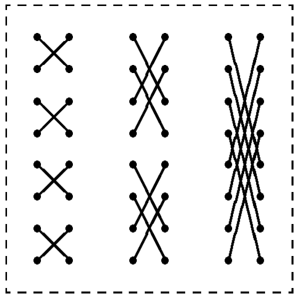


Figure 7.8: Complete backplane wiring for a STEP system consisting of a single block.

### 7.8.2 Block wiring, improved

Each of the "wires" shown in Fig. 7.6 or Fig. 7.8 is actually a collection of 16 lines (one per layer), and typically will be implemented by a shielded flat cable. Since such cables are stiff and bulky, it is desirable to route them as neatly as possible, trying to keep them flat and parallel and avoiding repeated crossings.

By systematically rearranging the order of modules within a box and the order of ports within a module it is possible to arrive, for a single block, at the wiring pattern of Fig. 7.9, to be constrasted with that of Fig. 7.8. The improved wiring is arrived at as follows.

*Port swapping.* Fig. 2.14 suggests that, if strand 0 loops back as strand 1 in the $x$ dimension, it would be more convenient if the modules of strand 1 had their $x_+$, $x_-$ ports arranged in the opposite order as those of strand 0. More generally, a similar argument holds for the $u$-dimension ports ($u = x, y, z$) of any pair of strands whose labels differ only in the $s_u$ digit (cf. (2.2)).

In order to respond to design requirements of the above kind, in a STEP module the correspondence between logical ports ($x_+$,$x_-$, etc.) and physical ports (the six connectors at the bottom of Fig. 7.5) is software assignable rather than hardwired (cf. §6.22). Taking advantage of this feature, one can make the two $x$ ports appear on the edge of the board in the order $x_-, x_+$ or $x_+, x_-$, depending on whether the $s_x$ bit is 0 or 1 (and similarly for $y$ and $z$), leading to the following backplane template

| *strand* | | *ports* | | |
|---|---|---|---|---|
| $s_x s_y s_z$ | $S$ | $x$ | $y$ | $z$ |
| 000 | 0 | $-+$ | $-+$ | $-+$ |
| 100 | 1 | $+-$ | $-+$ | $-+$ |
| 010 | 2 | $-+$ | $+-$ | $-+$ |
| 110 | 3 | $+-$ | $+-$ | $-+$ |
| 001 | 4 | $-+$ | $-+$ | $+-$ |
| 101 | 5 | $+-$ | $-+$ | $+-$ |
| 011 | 6 | $-+$ | $+-$ | $+-$ |
| 111 | 7 | $+-$ | $+-$ | $+-$ |

(7.3)

*Strand swapping.* Finally, we shall permute the boards in the backplane's eight slots so that the strand labels $S$ are ordered according to *Gray code* rather than straight *binary* code:

| *strand* | | *ports* | | |
|---|---|---|---|---|
| $s_x s_y s_z$ | $S$ | $x$ | $y$ | $z$ |
| 000 | 0 | $-+$ | $-+$ | $-+$ |
| 100 | 1 | $+-$ | $-+$ | $-+$ |
| 110 | 3 | $+-$ | $+-$ | $-+$ |
| 010 | 2 | $-+$ | $+-$ | $-+$ |
| 011 | 6 | $-+$ | $+-$ | $+-$ |
| 111 | 7 | $+-$ | $+-$ | $+-$ |
| 101 | 5 | $+-$ | $-+$ | $+-$ |
| 001 | 4 | $-+$ | $-+$ | $+-$ |

(7.4)

It is easy to verify that the port-and-strand arrangement of template (7.4) indeed leads to the improved backplane wiring pattern given in Fig. 7.9. Similar methods can be used to streamline backplane wiring in more complex situations, such as that of Fig. 7.7.

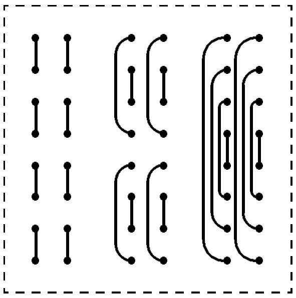


Figure 7.9: Improved backplane wiring for a STEP system consisting of a single block.

# 8 Step box

## 8.1 Overview

As we explained in Chapter 2.5, when a three-dimensional torus is flattened onto three-dimensional space, eight separate strands of the torus are brought together. A STEP box is responsible for simulating the activity of a STEP *block*, i.e., a sector-size window cutting through all eight strands (cf. Fig. 2.16). The entire torus can then be simulated by arraying in a straightforward way an appropriate number of boxes in the three dimensions of ordinary physical space.[1]

Besides the resources directly devoted to the simulation (site-data transport and interaction), a STEP system makes uses of other resources (e.g., clock distribution, communication with the host) that must somehow be distributed and interconnected in an indefinitely expandible way within the same geometric framework as the blocks. It is natural to lump such resources on a per-block basis and incorporate one lump in each box. Thus, while the module represents the smallest repetitive unit of STEP from a functional viewpoint, in physical terms it is the box that represents the smallest repetitive unit—from which an arbitrarily large STEP system can be built by mere juxtaposition and nearest-neighbor interconnection.

Physically, a STEP box is a roughly cubical enclosure containing a regulated power supply and a motherboard with eight slots—for up to eight STEP modules. Additional circuitry on a per-box basis is present on the motherboard. Though, for convenience, some or all of this circuitry may reside on a card that plugs into a ninth slot of the motherboard, we shall still refer to it as 'motherboard circuitry'.

As outlined in §7.4, a box interfaces to the outside world through three groups of signals, namely *glue* signals and *TBus* signals, which travel between boxes (§7.6), and *data stream* signals (data flywheel and video output), which allow external equipment to have real-time access to the box's internal data stream.

The signals of each group are bundled into one or more *ports* according to their geographical destination. Thus, the glue signals are divided into six glue ports according to which of the six adjacent boxes they connect to. Each port, in turn, is physically represented by one or more *connectors* into which cable headers may be plugged. For convenience, also configuration and test signals are grouped together and brought to connectors.

In the present implementation, all of the above connectors (which deal with those signals that the box makes available externally) are on the motherboard, independently of whether the signals themselves are generated (or received) by motherboard circuitry or by circuitry on the individual modules.

Fig. 8.1 gives a simplified backview of the motherboard, showing the individual connectors and their groupings as ports. The modules plug in on the other side of the motherboard, and many of the port connectors are just through-the-board extensions of connectors on the modules.

## 8.2 Glue ports

Each module contributes to its box six glue connectors (XM glue through ZP glue; cf. Fig. ?? and Fig. 9.1) for bidirectional communication between adjacent sectors. On the box, each *glue port* brings together a set of eight homologous connectors (one from each module), so that, for instance, the eight XM glue connectors come together as the box's XM glue port (Fig. 8.1). Each glue port has, in addition, a small connector with two signals (Clock Phase In and Clock Phase Out), handled by the motherboard circuitry, which exchange clock phase information between the local clock masters (cf. §7.5) of adjacent boxes.

Thus, between the XP glue port of a box and the XM glue port of the box next to it in the positive $x$ direction there will run 128 bidirectional glue lines (16 layers times 8 strands) plus 2 clock-phase lines, for a total of 130 signal lines.

Note that the six physical directions XM, XP, ..., ZP are distinct from the six *logical* directions, denoted by $x_-$, $x_+$, ..., $z_+$, which refer to the orientation of an individual strand. Typically, the XM glue port will be configured as an $x_-$ port by four of the modules in a box, and as an $x_+$ port by the remaining four (cf. Fig. ??). A more general remapping of logical ports to physical ports is possible; for instance, when the three-dimensional array of modules is configured as a pleated two-dimensional array (§??), a Z port may be used to carry $y$ information.

On each of the external faces of the parallelepiped—where a box is not mated to an adjacent box—four of the strands double back on the other four via the glue signals. For example, on an XM face, the four XM glue signals that carry $x_-$ information (cf. Fig. ??) for a certain layer will

[1] Note that while the eight strands that run through the boxes are interconnected so as to yield the topology of a torus, the physical array of boxes forms an ordinary *parallelepiped*: opposite faces don't have to be brought together!

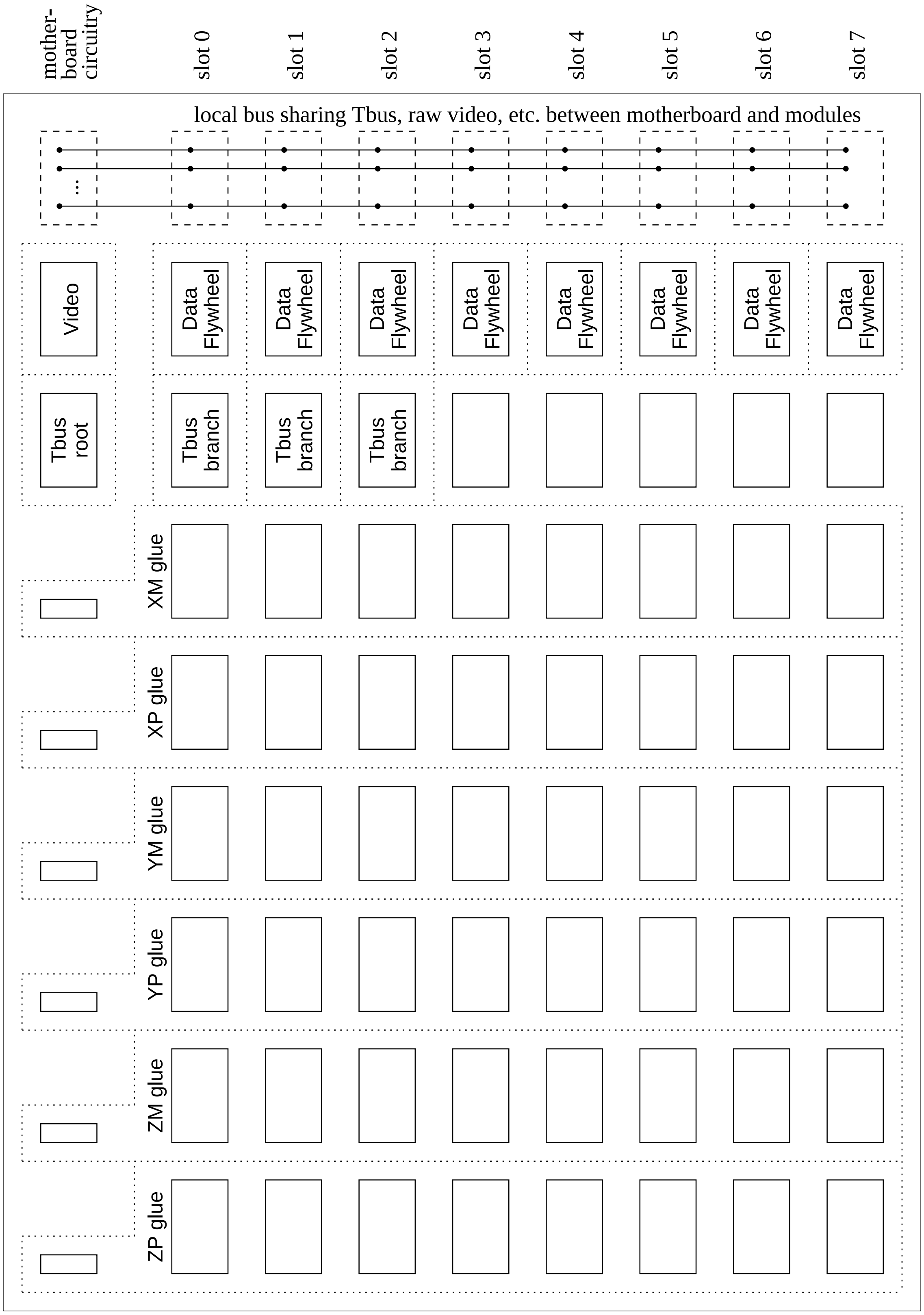



Figure 8.1: Simplified backview view of the STEP-box motherboard. Dotted lines delimit ports, each consisting of one or more connectors. The small connectors on the glue ports are for clock-phase feedback between adjacent modules.

fold back onto the corresponding four XM glue signals that carry $x_+$ information. This is done by a passive shorting plug or by an internal multiplexer (cf. §??). In a similar way, a Clock Phase Out signal that emerges from an external face of the parallelepiped is looped back into the corresponding Clock Phase In pin.

## 8.3 Data flywheel ports

On each STEPchip, real-time ("on-the-fly") access to the internal data stream is provided by two taps on this stream, namely Data Flywheel Output and Data Flywheel Input (§??). On the module, these sixteen outputs and sixteen inputs are brought together on a Data Flywheel connector (Fig. 9.1) together with the appropriate timing signals. In turn, this connector is passed through to the motherboard as a Data flywheel port, for a total of eight such ports (Fig. 8.1).

Data flywheel access is optional, and is provided for advanced experimentation, custom applications, attachment of custom hardware, specialized display or analysis needs, and in-depth testing and diagnostics.

As implemented in the standard STEP box described here, the Data flywheel ports are meant to provide direct real-time access to simulation data from any one module of a STEP machine, or at most a few modules: it would be physically impractical to connect external equipment to all modules at once. On the other hand, using specially designed boxes, it would be feasible to use the Data Flywheel connectors to add an external "coprocessor" to each module or group of modules.

## 8.4 Video port

A separate Video tap (§10.2.5, §??) on each STEPchip's internal data stream extracts data intended for real-time video display. On a module, these sixteen signals are brought out to the Raw video connector (Fig. 9.1). On the motherboard, the sixteen *Raw video* signals are bussed across the eight modules of a box (cf. §8.6); only one module at a time will have its tri-state bus drivers enabled. The Video port on the motherboard provides a buffered version of these sixteen raw-video signals, together with horizontal and vertical sync signals suitable for driving a CRT monitor.

Signal conditioning circuitry on the motherboard takes in the raw video signals and puts out, throught the Video connector, appropriately formatted video output (such as analog and Horizontal and Vertical Sync).

It is desirable to turn the raw video into analog Red, Green, and Blue suitable for a standard color monitor, by means of a downloadable color palette and digital-to-analog converters. Though these functions may be provided by external circuitry having access to buffered raw video, eventually they should be provided by the motherboard itself. At any rate, pins for RGB output are reserved on the Video port.

Typically, a dedicated CRT monitor permanently connected to the Video port of one of the boxes will be used to provide a real-time, sector-size window on the simulation data; this window can be panned at will through the site array.

## 8.5 TBus ports

A box also acts as a relay for *TBus* signals (§??), receiving the TBus via a TBus root connector on the motherboard and fanning it out to up to three adjacent boxes (in the XP, YP, and ZP directions) via the TBus branch connectors (Fig. 8.1). Since TBus relaying is done by the modules of behalf of the box, there are on the motherboard eight connectors (one on each module slot) available for the TBus branch function, with the potential for fanning the TBus out to eight boxes. However, no more than three of these connectors are used, since TBus distribution is organized on a three-dimensional tree basis (cf. §7.4).

## 8.6 Local bus

Besides relaying the TBus to other boxes via the TBus branch ports, the motherboard also conditions and distributes the TBus signals to the eight modules. This is done through a *local bus* (see Fig. 8.1), which also collects Raw video signals from the modules (§8.4), distributes the clock, and perform a number of miscellaneous functions.

## 8.7 Configuration and diagnostics

Finally, there are on the motherboard a number of pins used for configuration, testing, and diagnostics.

# 9 Step module

Physically, a STEP module consists of a printed circuit board which plugs into a slot of a STEP box motherboard (see §8.1) by means of a row of high-density edge connectors. The overall interconnection pattern for a box consisting of a motherboard and up to eight modules is sketched in Fig. 8.1 from the viewpoint of the motherboard, and in Fig. 9.1 from the viewpoint of a module.

## 9.1 Module pin types

The pins are sorted into functional groups, each of which is discussed in a separate section. In the pinout table in each section, the table header gives the total number of pins in the group and a descriptive name for the group itself (TBus interface, glue, video, etc.). For each pin, the tables give the pin type, according to table (9.1), and the pin's mnemonics followed by its full name.

Module pin names are distinguished by the prefix 'M/', to avoid confusion with STEPchip or box pin names.

As mentioned in §7.6, communication between STEP boxes (as well as between the controller and the corner box) takes place on transmission lines. The present pinout specifications are compatible with either unbalanced (e.g., coax cable) or balanced (e.g., twisted pair) transmission lines. In either case there are a *pair* of pins associated with each signal. Depending on the implementation, the two elements of the pair would be called "tip" and "ring," "core" and "shield," "signal" and "return," "plus" and "minus," or similar names. Here we'll use the noncommittal designators 'A' and 'B'.

In the pinout tables below, a signal-type designator ending with the character '2' (cf. table (9.1)) denotes a transmission line pair, listed as a single entry but with the understanding that two pins are involved. Thus, an entry of the form

| | | |
|---|---|---|
| O2 | M/FB* | M/Frame Branch* |

implies the existence of two pins, namely M/Frame Branch A* and M/Frame Branch B*.

Some of the transmission lines terminate on the motherboard (TBus root lines in Fig. 9.1), where they are handled by discrete drivers and receivers appropriate to the type of line. Most of the transmission lines, however, terminate directly on the module boards (e.g., XM glue or TBus branch in Fig. 9.1) and are associated with individual STEPchip pins. If these pins have sufficiently powerful internal drivers, they will be able to drive unbalanced transmission lines directly, i.e., without the need for separate discrete drivers. In what follows, we shall assume that this is the case; however, the module pinout will not change if discrete drivers are inserted between the STEPchip pins and the transmission lines, and/or if balanced transmission lines are used.

Note that the pins of motherboard connectors XM glue through ZP glue and *Tbus branch* (cf. Fig. 9.1) are not connected to any signal traces on the motherboard;[1] they simply provide mechanical support and passive feedthrough from the modules to the transmission lines.

| type | description |
|---|---|
| I | *Input.* Input signal driving a discrete on-board buffer which fans out the signal to the sixteen STEPchips of the module. |
| O | *Output.* Output signal driven by the Low-Drive Multipurpose Pin of one of the sixteen STEPchips. |
| B | *Bidirectional.* This pin is sensed as an input, and is driven as a tri-state output. |
| I2 | *Input pair.* Pair of pins (A and B) at the receiving end of a transmission line. |
| O2 | *Output pair.* Pair of pins (A and B) driving a transmission line. |
| B2 | *Bidirectional pair.* Pair of pins (A and B) which may drive or sense a transmission line. |

(9.1)

## 9.2 TBus interface

TBus local

| type | name | |
|---|---|---|
| B | Local.D[0:F] | Local Data[0:F] |
| I | Local.Reset* | Local Reset* |
| I | Local.Sync* | Local Sync* |
| I | Local.Frame* | Local Frame* |
| I | Local.OH | Local Outbound Handshake |
| O | Local.IH | Local Inbound Handshake |
| O | Local.Status* | Local Status* |

(9.2)

[1] In the case of unbalanced transmission lines, the B side of each pair would probably be tied to the ground plane.

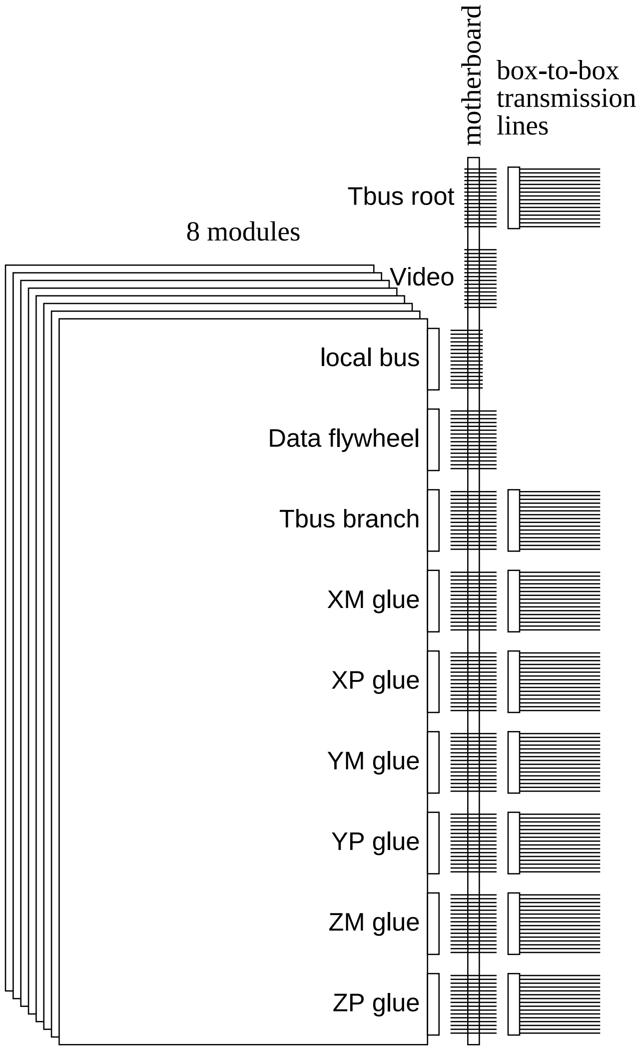



Figure 9.1: Overall interconnection pattern for the modules in a STEP box. On the motherboard, the six glue connectors (XM glue thru ZP glue) devoted to each module—for a total of 48 connectors—carry plain feed-through pins: the signals go directly from a module in one box to a module in an adjacent box via a transmission line. The situation is similar for the TBus branch signals, where, however, only at most three of the eight connectors are used (to extend the TBus tree in the $x$, $y$, and $z$ directions). There is only one TBus root connector, on the motherboard; outbound signals arriving at this connector are fanned out by the motherboard and distributed to the eight modules via the *local bus* connectors (one per module); similarly, inbound signals from these connectors are ORed into a single TBus root signal by the motherboard. The Video connector on the motherboard produces video information gathered from the modules via the local bus.

TBus branch

| type | name | |
|---|---|---|
| B2 | Branch.D[0:F] | Branch Data[0:F] |
| O2 | Branch.Reset* | Branch Reset* |
| O2 | Branch.Sync* | Branch Sync* |
| O2 | Branch.Frame* | Branch Frame* |
| O2 | Branch.OH | Branch Outbound Handshake |
| I2 | Branch.IH | Branch Inbound Handshake |
| I2 | Branch.Status* | Branch Status* |

(9.3)

The TBus local signals are handled by the motherboard, which fans out to the eight modules signals coming from the host (TBus root connector), and ORs signals from eight modules into a single hostward signal.

The TBus branch signals represent the place where a subtree of the TBus tree can be grafted.

## 9.3 Glue

Glue

| type | name | |
|---|---|---|
| B 2 | Glue0.G[0:F] | Glue X-Minus[0:F] |
| B 2 | Glue1.G[0:F] | Glue X-Plus[0:F] |
| B 2 | Glue2.G[0:F] | Glue Y-Minus[0:F] |
| B 2 | Glue3.G[0:F] | Glue Y-Plus[0:F] |
| B 2 | Glue4.G[0:F] | Glue Z-Minus[0:F] |
| B 2 | Glue5.G[0:F] | Glue Z-Plus[0:F] |

(9.4)

These are the signals that allow one to glue modules edge-to-edge so as to achieve an array of indefinite extent. Each glue signal runs point-to-point (no fan-out or fan-in) between two modules; in a large system, the two modules will reside in different boxes.

## 9.4 Data stream

Data flywheel

| type | name | |
|---|---|---|
| Out | Flywheel.O[0:F] | Flywheel Data Output[0:F] |
| In | Flywheel.I[0:F] | Flywheel Data Input[0:F] |
| Out | Flywheel.Clock | Flywheel Clock |
| Out | Flywheel.DSV* | Flywheel Data Stream Valid* |

(9.5)

Raw video

| type | name | |
|---|---|---|
| IO | Local.Video[0:F] | Local Video[0:F] |

(9.6)

## 9.5 Wired-AND communication

# 10 stepchip

Fig. 10.1 shows in a very schematic way the logical organization of a STEPchip, especially in relation to the chip's immediate surroundings.

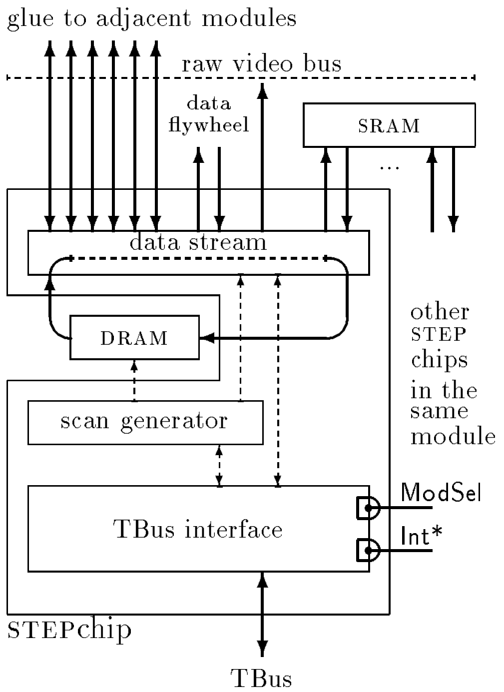



Figure 10.1: Logical organization of a STEPchip and its immediate surroundings. See text for details.

Some of the STEPchip's I/O lines (§??) connect it to other circuitry on the same module, some to homologous STEPchips on adjacent modules, some to the TBus, and some to optional external circuitry. These different types of interconnection are better understood with reference to the STEPchip's functions, as detailed below.

Though physically residing on a separate memory chip, the block labeled DRAM should logically be treated as part of the STEPchip, since all of its inputs and outputs are directly handled by the latter.

The main function of the STEPchip is to pump a data stream out of and then back into the DRAM. Along its course, the data stream may be modified by

- Data exchanges with adjacent modules, via the *glue lines*, to achieve indefinite spatial extension for the array of sites.
- Real-time data exchanges with optional external circuitry, via the *data flywheel* (§??).
- Real-time output to a raster-scan monitor, via the *raw video bus* (§??). All modules tristate their video lines onto a single Raw video bus, so that video output from any single module can be routed to the monitor.
- Data replacement by lookup in the SRAM table. The table is shared by the 16 STEPchips of a module, each of which provides one bit of the lookup *key* (table address) and and receives one bit of the corresponding *entry* (table data). Of course, this collective lookup is the main mechanism for interaction between data streams, and ultimately for the realization of the desired dynamical behavior as well as of real-time analysis.

The TBus (Chapter 3) is the exclusive channel of communication between the host and the STEP modules. All exchanges between the host and SRAM or DRAM chips are mediated by the STEPchip.

Beside being mutually coupled via the SRAM table, the 16 STEPchips of a module are also mutually coupled via a wire-AND ModSel line for the purpose of responding as a single addressable entity to TBus operations (§3.13), and a wire-AND Int* line for collecting interrupts from the sixteen chips and eventually merging them onto the TBus Status line.

## 10.1 stepchip block diagram

The internal structure of the STEPchip shown in Fig. 10.1 is further detailed in Fig. 10.2, which is taken from the STEPchip's actual design files (this diagram does not show the I/O buffers that interface the internal signals to the outside world).

## 10.2 Pinout

### 10.2.1 Chip pin types

The STEP chip is packaged in a 68-pin PLCC.

The pins are sorted into functional groups, each of which is discussed in a separate section. In the pinout table in each subsection, the table header gives the total number of pins in the group and a descriptive name for the group itself (DRAM interface, TBus interface, etc.). For each pin, the tables give the pin number, its type, according to table (10.1), and then the pin's mnemonics followed by its full name.

Figure 10.2: STEPchip block diagram, taken from actual design files.

| type | description |
|---|---|
| I | *Input.* This pin is sensed as an input. |
| O | *Output.* This pin has an output driver, which is always enabled. |
| T | *Tri-state.* This pin has an output driver which can be disabled. |
| B | *Bidirectional.* This pin is sensed as an input, and is driven as a tri-state output. |
| W | *Wire-AND.* This pin is driven in a way suitable for wire-ANDing. |

(10.1)

### 10.2.2 DRAM interface

DRAM interface

| type | name | |
|---|---|---|
| O | DA(10:0) | DRAM Address(10:0) |
| B | DD(3:0) | DRAM Data(3:0) |
| O | DRAS* | DRAM Row Address Strobe* |
| O | DCAS* | DRAM Column Address Strobe* |
| O | DWE* | DRAM Write Enable* |
| O | DOE* | DRAM Output Enable* |

(10.2)

These are the address, data, and control signals for the DRAM chip associated with each STEPchip.

Since we use nybble-wide chips, total DRAM capacity with 11 address lines (DA(10:0)) is $4 \cdot 2^{2 \cdot 11}$, or 16 Mbits.

Once we are in fast-page or static-column mode, typically a DRAM read/write operation takes 4 clock cycles; in the first two, a nybble is read, in the second, a nybble is written.

### 10.2.3 SRAM interface

SRAM interface

| type | name | |
|---|---|---|
| O | SA0 | SRAM Address, Bank 0 |
| O | SA1 | SRAM Address, Bank 1 |
| B | SD0 | SRAM Data, Bank 0 |
| B | SD1 | SRAM Data, Bank 1 |

(10.3)

These are the address and data signals for the two SRAM lookup tables, LUT0 and LUT1. The SRAM control signals are distributed on a module-wide basis by means of four of the Multipurpose pins (§6.21).

### 10.2.4 Glue

Glue

| type | name | |
|---|---|---|
| B | XM | X-Minus |
| B | XP | X-Plus |
| B | YM | Y-Minus |
| B | YP | Y-Plus |
| B | ZM | Z-Minus |
| B | ZP | Z-Plus |

(10.4)

These are the signals that allow one to "glue" modules edge-to-edge so as to achieve an array of indefinite extent.

### 10.2.5 Data stream

Data stream

| type | name | |
|---|---|---|
| O | FDO | Flywheel Data Output |
| I | FDI | Flywheel Data Input |
| T | Video | Video |

(10.5)

At every step, the entire DRAM contents (or whatever fraction of it is actually used for the model's state variables) is serially made visible to the outside world via the Flywheel Data Output pin, before being sent to the lookup table. External circuitry can use these data for analysis purposes; and can replace any of these data bits with externally supplied ones. On every clock cycle in which Enable* is asserted, the internal data bit is replaced with one that is externally supplied via the Flywheel Data Input pin. The timing of this injection—either one or four clock cycles[1] later than when the original bit appeared on the FDO pin—allows for the original bit to be taken into consideration when generating the replacement bit.

### 10.2.6 TBus interface

TBus

| type | name | |
|---|---|---|
| I | TF* | TBus Frame* |
| I | TOT | TBus Outbound Tag |
| I | TIT | TBus Inbound Tag |
| B | TD | TBus Data |
| B | TDB | TBus Data Branch |
| I | Sync* | Sync* |
| I | Reset* | Reset* |

(10.6)

The following output signals are produced on a per-module rather than per-chip bases, and are handled by a multipurpose pin: TBus Inbound Tag Local (TITL), TBus Outbound Tag Branch (TOTB), TBus Frame Branch* (TFB*), TBus Sync Branch* (TSB*),

[1] Depending on the setting of the FPL bit in the Hardware Environment Register (§6.20).

Other TBus related signals go through significant on-board conditioning, and are listed in §10.2.7.

### 10.2.7 Miscellaneous

Miscellaneous

| type | name | |
|---|---|---|
| I | Clock | Clock |
| O | MP | Multipurpose |
| W | Int* | Interrupt* |
| W | ModSel | Module Select |

(10.7)

The Interrupt* pins from the 16 chips of a module are wire-ORed together. The resulting signal is merged with Scan-in-Progress information and output on one of the 16 Multipurpose pins (§6.21), to be used, after proper conditioning, as the TBus Status* signal contributed by the module.

The Module Select pins from the 16 chips of a module are wire-ANDed together, to allow the individual chips to contribute to a module-wide AND and sense its result, as explained in §3.13. As each pin will be actively driven immediately before the clock cycle in which the AND operation will be performed, the pull-up resistor connected to this wire is only for long-term bleeding purposes,

Reset* and Clock are derived from the corresponding TBus signals. Redistribution of these signals to the tree branches is handled by on-board discrete circuitry rahter than by the STEPchip.

### 10.2.8 Spares, reserved

Reserved

| type | name | |
|---|---|---|
| I | Esc0 | Escape0 |
| I | Esc1 | Escape1 |

(10.8)

The behavior of the STEPchip is characterized only when the two pins Escape0 and Escape1 are tied to logical 0.

In some versions of the die used for the STEPchip, for fabrication convenience some of the spare die area may have been used for other chips. A scheme presently being considered for this multiplexing of functions on the same die is the following

| Esc0 | Esc1 | chip selected |
|---|---|---|
| 0 | 0 | STEPchip |
| 0 | 1 | CTLSV chip |
| 1 | 0 | FSM chip |
| 1 | 1 | DATAREGS chip |

. (10.9)

The STEP controller will use one FSM, two STLSV, and two DATAREGS chips.