Signetics

VMSbus Specification Manual REV A4

Nov. 1983

VMSbus

(VME SERIAL BUS)

SPECIFICATION MANUAL

P R E L I M I N A R Y

PUBLISHED BY

THE

VMEbus MANUFACTURERS' GROUP

Revision A4

November, 1983

1. INTRODUCTION: WHY DO WE NEED THE SERIAL BUS ?

No matter how fast and capable computers and microprocessors become, there always seem to be applications which are just beyond the ability of a particular processor. A processor may not be fast enough to process inputs in "real time", or may not provide fast enough response time to satisfy users, etc. It simply takes too long to "get the job done".

In the past such applications often had to be delayed until a new and faster processor was developed, which was also cost-effective for the application. But recent microprocessors have added hardware features which allow more than one of them to be hooked together in a system. Such multiprocessor systems provide an alternative to waiting for a faster processor to be developed. Two or more processors can share the processing load of the system and provide the necessary "real time" processing, faster response time, and so on.

A well-known and respected lady computer scientist uses an analogy from the early days of oxcarts. When the first oxcart users and engineers encountered a load which was too heavy for their ox to pull, they did not put off moving the load until a larger ox was developed. They simply hooked up two oxen to the cart and did the job. Her message is that multiprocessor systems should be more widely used.

To some extent, multiprocessing is a field in which hardware development has outrun software development. Much of current software thinking and techniques are tied to the single-processor systems for which existent software was developed. Multiple processor systems pose new problems for system software designers, and solutions to these problems are still evolving.

A number of possible ways to design software for multiprocessing have been developed. Single-processor multitasking techniques can be extended for multiple processors. Multitasking can be thought of as the sharing of a processor among several tasks (programs or parts of a program). The processor is first assigned to one task which it executes for a while, then to another task for a while, then to another, etc.

The most straightforward way to extend single-processor multitasking into the multiprocessor environment is to give each processor a more or less fixed set of tasks which it executes. More complex multiprocessor software techniques include "anonymous processor" systems, wherein any available processor can execute any task in the system.

Other computer scientists argue that classic multitasking techniques are unreliable in the multiprocessor environment because the number of ways the processors can interact is very large, and it is too easy for some interactions to be overlooked by system designers. These people say that the only way to assure reliable, fully debugged multiprocessor software is to program all the processors in one program, which specifies and controls all the ways in which the processors can interact.

Alternative approaches will probably continue to be developed and debated for some time. However, in all of the approaches there are certain common needs. If we can design system hardware which helps satisfy these needs, it makes the implementation of multiprocessor systems more straightforward for everyone.

The serial bus is intended to meet the needs of multiprocessor systems. The following sections describe some of these needs and how the serial bus provides for them.

1.1 THE SERIAL BUS ALLOWS THE COMMUNICATION OF EVENTS

An event could be defined as any significant occurrence which is not a direct, immediate, and normal result of what the processor is currently doing. For example, if a program commands a disk controller to start a transfer, we would not consider the fact that the disk controller starts to be an event. The completion of the disk transfer is an event.

We will discuss events as they impact multitasking systems. In such systems there is normally a task for each event that the system recognizes. When an event occurs, the processor may be reassigned from executing one task to executing another.

A task can also cause the reassignment of the processor. For example, the task which is currently being executed by the processor may need to wait for an event to occur before it can proceed. In this case it requests system software to reassign the processor to other tasks until the event occurs. As another example, a task may request that another task be executed. Communication between tasks is often done through such requests.

Each task in a multitasking system has an associated "priority". (All of the tasks can be "rank ordered" from the highest priority to the lowest.) If an event occurs, for which the task priority is higher than that of the task which the processor is currently executing, a reassignment takes place. The processor stops executing the instructions of the old task and starts executing those of the new event's task.

There are two kinds of events: those which occur outside the processor (external events), and those which occur inside the processor (internal events). External events are commonly communicated to the processor by means of interrupts. Internal events don't need to be communicated to the processor, but they may still cause it to be reassigned to a new task. (An example of an internal event is an attempt to divide by zero.)

Microprocessors have a small number of "interrupt request" lines that can be used to communicate events to them. Since there are usually many possible events, they must all share these interrupt request lines. When the processor detects an interrupt, it must gather more information in order to figure out which event has occurred and which task is to be executed. There are several methods for doing this: polling of I/O status registers, reading a "vector" from the interrupting device, etc. In each case the processor does one or more "read cycles" to collect this information.

In some systems, an interrupt requires the processor to read from the status registers of all possible interrupting devices until it finds one that is generating an interrupt. Since a lot of reads may be required, this method is rather inefficient. More recent processors use vectored interrupt techniques.

Vectored interrupts work as follows. Each device capable of generating interrupts is assigned one or more "vectors". Each vector leads to an address where that event's task begins in the processor's memory. When the event occurs and the processor is interrupted, it reads this vector from the interrupting device. This tells the processor the starting address of the new task to execute.

Earlier we said that tasks can communicate with each other to accomplish a common purpose. When there are several processors, it may turn out that a task being executed on one processor may need to communicate with a task which is being (or will be) executed on another. Thus it is important that the processors have a quick way to communicate events BETWEEN EACH OTHER. (It's no longer sufficient to simply communicate all events to a single microprocessor.)

When multiprocessing systems are built using existing backplane buses this may not be possible. For example, backplanes which were designed for single processor systems may have only one interrupt line. When building multiprocessing systems, such an interrupt structure simply won't do the job.

The newer backplane buses have a provision for directing interrupts to several different processors. For example, VMEbus has seven interrupt request lines which can each be monitored by a different processor. If a device in the system needs to communicate an event to a processor, it can request an interrupt on a line that is monitored by that processor.

But even this approach has its limitations. When a processor acknowledges (answers) an interrupt request, the processor typically must read a vector before assigning itself to the new task. If this is done as a transfer on the system bus, the processor may have to request the bus and wait until it becomes available. If other bus masters are using the bus, this delay may be too long or too unpredictable to satisfy the needs of time-critical, high performance applications. Sometimes the problem can be solved by rearranging the priorities of the various bus masters, but the other masters' needs for the bus may be equally or more important to the performance of the system.

We conclude from all this that what is needed is a path specifically designed for communicating events. To be useful in a multiprocessing system, this path must allow any processor to communicate events to any other, as well as permitting other devices in the system to communicate events to any of the processors. Such communication is one of the primary purposes of the serial bus.

1.2 THE SERIAL BUS AIDS IN "FAULT TOLERANT" SYSTEM DESIGN

Another trend in microprocessor systems is toward applications which demand "fault tolerance". This simply means that a system must be able to continue operation despite one or more hardware failures. One way to satisfy this need is by using several processors in a system, and providing ways for each processor to observe the operation of all the other boards, to detect when one is malfunctioning. Such systems can also include hardware registers on each board which control the board's backplane bus drivers. If one of the processors detects a malfunctioning board, it can write into a control register on that board and turn off the board's bus drivers, effectively "disconnecting" it from the backplane.

Since even a failure in a processor could be detected by another processor, at first glance this seems like a good answer to the question of fault tolerance. The scheme turns out to be inadequate, however, where there is only one pathway between boards. If the failure on a board is such that it prevents proper operation of the system bus, a processor may be able to detect the failure, but it cannot use the system bus to disable the failed board.

The serial bus provides another pathway between boards, which can be used to prevent a failed module from interfering with the operation of the system bus. The serial bus can be used to disable the drivers of such a board, or direct a board-specific Reset to it. Conversely, if a failed board prevents the proper operation of the serial bus, the system bus can be used to disable the board's access to the serial bus.

1.3 THE SERIAL BUS PROVIDES FOR "INTELLIGENT SEMAPHORES"

The third primary use of the serial bus is again related to multiprocessor systems. When a system includes multiple processors which can simultaneously try to access and use a variety of shared resources (parts of the system), some means must be found to control this access and use. A simple example of such a problem is when two processors simultaneously set out to use a shared hardware device such as a printer or disk. Other cases where interprocessor control is needed include access to: a data file, a data or control table in memory, or a section of program code which must be used by only one processor at a time.

The most widely used solution to these problems is the "semaphore". A semaphore is simply a location in memory which multiple system processors can access, but not simultaneously. A processor first reads the memory location to test whether another processor already has control of the shared resource, and then (if not) it writes to the location to show that it now has control. The system hardware must ensure that these two steps happen without allowing any other processor to access the semaphore location. The newer backplane buses include provisions for semaphore operations.

A semaphore in common memory solves some of the problem but is less than perfect. As the number of processors in a system increases, it becomes more likely that several of them need to use a resource at the same time. Many shared parts of a system such as data and control tables are designed to be in use by a processor for a relatively short time. For such cases, if the associated semaphore is found to be already set by another processor, a common software tactic is simply to rerun the "Read-Modify-Write" (RMW) or "Test and Set" operation until it succeeds. Studies of heavily loaded multiprocessor systems indicate that a significant percentage of processor cycles and cycles on the system bus are wasted on repetitive accesses to semaphore locations. This in turn can reduce the availability of the bus for other uses.

One way to solve this problem is to provide a duplicate copy of each semaphore on each processor board. The copies of the semaphore should, of course, always be kept synchronized. If a processor needs to set a particular semaphore, the corresponding on-board copy is first checked.

If and only if the onboard copy of the semaphore is reset, a RMW operation is run on the system bus, which affects all the copies of the semaphore in the system (including the on-board one). If this RMW "succeeds", all copies of the semaphore are simultaneously set, and the requesting processor is entitled to use the associated resource. (If two processors issue the RMW at the same time, system bus arbitration ensures that only one succeeds.) A scheme like this largely eliminates wasting system bus cycles on polling semaphore locations in memory.

If the local semaphore is set or the RMW operation on the system bus fails, system software has two choices. If the nature of the resource controlled by the semaphore is such that the semaphore will be cleared in a short time, it is more efficient to "loop" testing the <u>local</u> copy of the semaphore until it is cleared. An example of such a "short term" semaphore would be one controlling access to a data or control table in memory.

But if the semaphore will typically stay set for a relatively long time, system software and the processor should go on to other tasks. In this case, onboard hardware must generate an interrupt when the semaphore is cleared so that the processor can retry the RMW on the system bus. Better still, the onboard hardware can itself retry setting the semaphore and interrupt the processor only when it has gained control. An example of such a "long term" semaphore would be one controlling access to a physical device like a printer or a disk.

For "long term" semaphores, the serial bus offers a great improvement. A processor can turn the entire operation of setting the semaphore over to onboard serial bus hardware, and be interrupted only when the semaphore has been set and the associated resource is actually available.

1.4 THE SERIAL BUS ALLOWS TOKEN PASSING SCHEMES

Sometimes a system has a group of several interchangeable resources. If semaphores are used to allocate these resources, then whenever a resource is needed, the semaphores must be polled until one is found to be available. In some cases it may make more sense to have a "token" in the system for each of these interchangeable resources, which is passed around among its "users" while the resource is available, but is retained by a user while it uses the resource. (Such "token passing" operation can be likened to a "daisy chain" in a backplane bus, which has been looped back on itself to make a ring.)

The serial bus allows a large number of such tokens (up to 1024) to be created and passed from board to board with very little software overhead.

1.5 THE SERIAL BUS PROVIDES A LOW COST ALTERNATIVE BUS

The serial bus is a unique new concept in microprocessor system interfacing. Its capabilities make it very attractive for use in complex multiprocessing systems. But in addition to its power, a serial bus interface can be implemented at lower cost than a system bus interface. With the LSI support which will soon be available, serial bus hardware can be implemented in a fraction of the board space required for a parallel backplane bus.

Thus the serial bus offers a very attractive alternative to use of a parallel system bus for boards which do not require a high data rate. It can even be used as the primary system bus in some applications.

2. SERIAL BUS OPERATION

The serial bus interface system consists of two signal lines named SERCLK and SERDAT* and six module types called HEADER SENDERS, HEADER RECEIVERS, DATA SENDERS, DATA RECEIVERS, FRAME MONITORS and a SERIAL CLOCK. The SERCLK line is driven by a totem pole driver in the SERIAL CLOCK module. The SERDAT* line can be driven Low by all six module types using open collector drivers. When no module drives SERDAT* Low, the bus terminating resistors pull it to a High level. The SERDAT* line is a Low-True signal (i.e. a "one" is represented by a Low level). This fact, plus the open collector characteristics of the drivers, results in "logical OR'ing" when data is placed on SERDAT* by more than one module.

In some cases on-board signals connect modules on the same board, but most communication between modules is done by sending "frames" on the SERDAT* line. These frames are composed of "subframes" which are sent by various modules. A frame is initiated when a HEADER SENDER module sends a "Header subframe". The other modules then respond by sending subframes according to a prescribed protocol until the end of the frame is reached.

During the Header subframe transmission, HEADER SENDERS are required to sample each bit on the SERDAT* line while they are sending. If a HEADER SENDER detects SERDAT* Low when it is sending a "zero" (i.e. when it isn't driving SERDAT* Low), it stops sending. This allows other HEADER SENDER(S) to finish sending the Header subframe without interference. This method of arbitration allows several HEADER SENDERS to start sending frames simultaneously, without affecting each others' transmissions. (One of the transmissions will be successful while the others are tried again later.)

Serial bus modules are found on boards in groups. The following are the most common groups:

TYPE 1) A HEADER SENDER and a FRAME MONITOR

TYPE 2) A HEADER RECEIVER and a flip-flop

TYPE 3) A HEADER RECEIVER and a DATA SENDER

TYPE 4) A HEADER RECEIVER and a DATA RECEIVER

TYPE 5) A HEADER RECEIVER, a DATA SENDER, and a DATA RECEIVER

A TYPE 1 group is used to initiate frames by sending a "Header subframe". The Header subframe specifies what other modules will participate in the frame transmission by providing two ten bit "selection codes". Each HEADER RECEIVER on the serial bus has a ten bit code which it compares with the two codes in the subframe. If its code matches either of them, depending on the type of group the HEADER RECEIVER is in, it responds by changing the state of its flip-flop, by telling its DATA SENDER to send, or by telling its DATA RECEIVER to receive.

2.1 USING THE SERIAL BUS TO TRANSFER DATA

The HEADER SENDER can determine whether there is a frame in progress from the FRAME IN PROGRESS signal generated by its FRAME MONITOR. If there is no frame in progress it can initiate one by sending a Header subframe. This subframe has a ten bit "S field" and a ten bit "R field". To transfer data, the HEADER SENDER puts a selection code in the S field that corresponds to some TYPE 3 or TYPE 5 group on the bus, and a selection code in the R field that corresponds to some TYPE 4 or TYPE 5 group. (The actual codes used to select these groups depend on how the system software or firmware has configured the system. There are no selection codes used exclusively to select TYPE 4 groups, etc.)

Each HEADER RECEIVER on the serial bus compares these codes against its code. One or more of the TYPE 3 or TYPE 5 HEADER RECEIVERS finds a match with the S field. It tells its DATA SENDER to send data. In a similar way, one or more of the TYPE 4 or TYPE 5 HEADER RECEIVERS finds a match with the R field and tell its DATA RECEIVER to receive data.

The actual number of bytes transferred is left up to the DATA SENDER. After the HEADER SENDER has sent the Header subframe, the DATA SENDER sends a three bit subframe indicating the number of bytes it intends to send to the DATA RECEIVER, followed by the data bytes. The DATA RECEIVER then responds with an indication that it has received the data bytes.

2.2 USING THE SERIAL BUS TO SET AND RESET FLIP-FLOPS

When a HEADER SENDER is used to set or reset a flip-flop, it sends a Header subframe as in the case described above. Instead of sending codes for TYPE 3, 4 or 5 groups in the S and R fields, however, it sends the code for a TYPE 2 group in one of the fields, and a "dummy" code (all ones) in the other field. If the frame is intended to <u>set</u> a flip-flop, it sends the TYPE 2 code in the S field, and the dummy code in the R field. If the frame is intended to <u>reset</u> a flip-flop, it sends the dummy code in the S field and the code for the TYPE 2 group in the R field.

When the Header subframe is sent, each HEADER RECEIVER on the serial bus compares the codes in the S and R fields to its own code. One or more HEADER RECEIVER(S) in a TYPE 2 group matches the S (or R) field and sets (or resets) its on-board flip-flop.

2.3 OTHER USES FOR THE SERIAL BUS

As we will see later, groups of serial bus modules like those described above can be used as building blocks for very powerful system configurations. For example, the combination of a TYPE 1 and a TYPE 5 group, on each of two boards, can be used by one board to pass an address and read the contents of a memory location on the other. TYPE 2 groups can be used to reinitialize one or more of the boards in a system, or to selectively disconnect a failed board from the system bus. TYPE 2 groups can be also be used to provide "semaphores" which are functionally superior to semaphores in a common memory, or to provide "token passing".

3. SERIAL BUS FRAMES

As we said in Section 2, a frame is actually composed of subframes which are sent by several modules in sequence whenever a HEADER SENDER sends a Header subframe. Depending on the types of module groupings that a frame selects, different modules drive and receive the various subframes, as follows:

Frames that select TYPE 2 module groups...

Subframe	Sent By	Received By
Header	HEADER SENDER	HEADER RECEIVER
Frame Type	Nobody (000)	HEADER RECEIVER, FRAME MONITOR
Frame Status	HEADER RECEIVER	FRAME MONITOR

Frames that select TYPE 3, 4, and 5 module groups...

Subframe	Sent By	Received By
Header	HEADER SENDER	HEADER RECEIVER
Frame Type	DATA SENDER	DATA RECEIVER, FRAME MONITOR
Data	DATA SENDER	DATA RECEIVER
Frame Status	DATA SENDER, DATA RECEIVER	FRAME MONITOR, DATA SENDER, DATA RECEIVER

Frames that get cancelled...

Subframe	Sent By	Received By
Header	HEADER SENDER	HEADER RECEIVER
Frame Type	HEADER RECEIVER	FRAME MONITOR, DATA SENDER, DATA RECEIVER

When a Header subframe is sent, modules on the serial bus are selected to interact during the remainder of the frame. This interaction can be seen on the serial bus as a sequence of subframe transmissions. Depending on what module groups are selected, we may see eleven possible kinds of frames.

1) A Flip-flop Set frame
2) A Flip-flop Reset frame
3) A Semaphore Set frame
4) A Token Passing frame
5) A 1 byte Data Transfer frame
6) A 2 byte Data Transfer frame
7) A 4 byte Data Transfer frame
8) An 8 byte Data Transfer frame
9) A 16 byte Data Transfer frame
10) A 32 byte Data Transfer frame
11) A Cancelled frame

3.1 A FLIP-FLOP SET FRAME

A Flip-flop Set frame contains 4 subframes:

Header	Frame Type	Frame Status	Jam Bit
S field=TYPE 2 R field=all ones	000	010	0

The Header subframe is sent by a HEADER SENDER. This subframe has two selection code fields in it: the S field and the R field. The Flip-flop Set frame has the selection code of a TYPE 2 group in its S field and all ones in the R field. One (or more) HEADER RECEIVER(S) on the serial bus finds a match between the S field code and its own code, sets its flip-flop, and responds with 010 in the Frame Status subframe.

The Flip-flop Set frame selects TYPE 2 module groups. The HEADER RECEIVER in a simple TYPE 2 module group never drives SERDAT* during the Frame Type subframe. Since a TYPE 2 module doesn't include a DATA SENDER, no module drives SERDAT* in the Frame Type subframe. This results in a three bit Frame Type value of 000 (all High). This value tells all of the FRAME MONITORS on the serial bus that there will be no data bytes (i.e. that the Frame Status subframe follows directly, and that the serial bus will be free thereafter). The FRAME MONITORS use this information to signal their HEADER SENDERS when the next frame can begin.

After the Frame Type subframe is complete, the HEADER RECEIVER sends 010 in the Frame Status subframe to indicate that at least one HEADER RECEIVER matched the S field. This acknowledgment is captured by the FRAME MONITOR that is paired with the HEADER SENDER, and reported to its on-board logic.

Since the R field in the header contained a "dummy" code of all ones, no HEADER RECEIVER matches the R field. (The "all ones" code is never used to select modules.) Because of this, the Frame Status subframe is only driven by the HEADER RECEIVER(S) which matched the S field, and remains 010. Since the header was intentionally arranged not to select an "R-module", the 010 value is the expected response.

3.2 A FLIP-FLOP RESET FRAME

A Flip-flop Reset frame contains four subframes:

Header	Frame Type	Frame Status	Jam Bit
S field=all ones R field=TYPE 2	000	001	0

The Header subframe is sent by a HEADER SENDER. As in the Flip-flop Set frame, this subframe has two selection codes in its S field and R field. The Flip-flop Reset frame, however, has the selection code of a TYPE 2 group in its R field and all ones in the S field. One (or more) HEADER RECEIVER(S) on the serial bus finds a match between the R field code and its own code, resets its flip-flop, and responds with 001 in the Frame Status subframe.

The Flip-flop Reset frame selects TYPE 2 module groups. The HEADER RECEIVER in a simple TYPE 2 module group never drives SERDAT* during the Frame Type subframe. Since a TYPE 2 module doesn't include a DATA SENDER, no module drives SERDAT* in the Frame Type subframe. This results in a three bit Frame Type value of 000 (all High). This value tells all of the FRAME MONITORS on the serial bus that there will be no data bytes (i.e. that the Frame Status subframe follows directly, and that the serial bus will be free thereafter). The FRAME MONITORS use this information to signal their HEADER SENDERS when the next frame can begin.

After the Frame Type subframe is complete, the HEADER RECEIVER sends 001 in the Frame Status subframe to indicate that at least one HEADER RECEIVER matched the R field. This acknowledgment is captured by the FRAME MONITOR that is paired with the HEADER SENDER, and is reported to on-board logic.

Since the S field in the header contained a "dummy" code of all ones, no HEADER RECEIVER matches the S field. Because of this, the Frame Status subframe is only driven by the HEADER RECEIVER(S) which matched the R field, and its value remains 001. Since the header was intentionally arranged not to select an "S-module", the 001 value is the expected response.

3.3 A SEMAPHORE SET FRAME

A Semaphore Set frame which is "successful" contains four subframes:

Header	Frame Type	Frame Status	Jam Bit
S field=TYPE 2 R field=Req. Code	000	010	0

The Header subframe is sent by a HEADER SENDER. As in all frames, this subframe has two selection codes in its S field and R field. The Semaphore Set frame has the selection code of a TYPE 2 group in its S field.

The R field of a Semaphore Set frame contains a ten bit code which represents the unique identity of the Requester that caused the frame to be sent. System software should ensure that two HEADER SENDERS are never allowed to send Semaphore Set frames with the same R field code. (In a multitasking system, this could be assured by assigning a unique Requester number to each task in the system.) The uniqueness of these Requester numbers guarantees that if two or more HEADER SENDERS try to set the same semaphore at the same time, only one of them will survive the serial bus arbitration and finish the frame.

A TYPE 2 group used for semaphore operations differs from a simple Type 2 group in having the state of its flip-flop fed back into its HEADER RECEIVER. The HEADER RECEIVER in a TYPE 2 group used for semaphore operations will not accept a Set frame if its flip-flop is already set. Whenever such a frame is sent, the HEADER RECEIVER drives the value 111 in the Frame Type field, which "cancels" the frame. (A Cancelled frame is shown below.)

If the HEADER RECEIVER matches the S field, and its flip-flop is reset, operation proceeds as described for a Flip-Flop Set frame.

> NOTE 1: Any "Requester code" that is used in the R field of a Semaphore Set frame should be an otherwise unused selection code, so that it doesn't select some HEADER RECEIVER on the bus.
>
> NOTE 2: Semaphores are reset using Flip-flop Reset frames, except that the S field may contain either all ones or the same Requester code used in the Semaphore Set frame.

3.4 A TOKEN PASSING FRAME

A Token Passing frame which is "successful" contains four subframes:

Header	Frame Type	Frame Status	Jam Bit
S field=TYPE 2 R field=TYPE 2	000	011	0

The Header subframe is sent by a HEADER SENDER. The Token Passing frame has the selection code of one TYPE 2 group in its S field and the code of another TYPE 2 group in its R field.

As with a TYPE 2 group used for semaphore operations, the HEADER RECEIVER in a TYPE 2 group used for token passing operations will not accept a Set frame if its flip-flop is already set. In addition, it will not accept a Reset frame while its flip-flop is reset. Whenever such a frame is sent, the HEADER RECEIVER "cancels" the frame.

A Token Passing frame is sent to simultaneously clear one flip-flop and set another. (In effect, it is passing a "token bit" from the flip flop it resets to the one it sets.) If the flip-flop it is trying to reset is already reset (i.e. it doesn't have a token to pass) the frame will be cancelled. Likewise, if the flip-flop it is trying to set is already set (i.e. it is already holding a token) the frame will also be cancelled.

These frame cancellation features ensure that a token bit is never lost or created in the process of moving it from one board to another.

In a Token Passing frame, the S field of the header identifies the Type 2 group to which these token is being passed, and the R field identifies the Type 2 group from which it is being passed. Assuming that the HEADER RECEIVER which matches the S-field has its flip-flop reset, it sends 010 in the Frame Status subframe. Assuming that the HEADER RECEIVER which matches the R field has its flip-flop set, it sends 001 in the Frame Status. These two values are logically OR'ed on the serial bus, resulting in the value 011 as the "correct" value reported by the FRAME MONITOR paired with the HEADER SENDER which initiated the frame.

3.5 A DATA TRANSFER FRAME

A Data Transfer frame which is "successful" contains five subframes:

Header	Frame Type	Data	Frame Status	Jam Bit
S field=TYPE 3 or 5 R field=TYPE 4 or 5	001-110	(Varies)	011	0

The Header subframe is sent by a HEADER SENDER. As in all frames, this subframe has two selection codes in its S field and R field. The S field has the selection code of a TYPE 3 or Type 5 group, which includes a DATA SENDER. Similarly, the R-field has the selection code of a TYPE 4 or 5 group, which includes a DATA RECEIVER. At the conclusion of the Header subframe, the selected DATA SENDER(S) drive a code on SERDAT* during the Frame Type subframe, to indicate how many bytes will be sent. While it is sending this code, a DATA SENDER also samples SERDAT*. If it samples the value 111 in the Frame Type, the frame is "cancelled", and the DATA SENDER terminates its transmission.

Except in a Cancelled frame, the DATA SENDER sends a Data subframe after the Frame Type subframe. The Data subframe may be 1, 2, 4, 8, 16, or 32 bytes long.

If two or more DATA SENDERS are selected by the header subframe, the serial bus protocol requires that they agree on the amount of data to be sent. If a DATA SENDER samples a different value in the Frame Type subframe than the one it sent (other than 111), it sends 110 in the Frame Status subframe which follows the Data subframe, indicating that the two DATA SENDERS tried to send different size Data subframes. If it sampled its own Frame Type value on SERDAT*, and if no other module sends a "one" in the high-order bit of the Frame Status, a DATA SENDER sends 10 in the the next 2 bits.

Similarly, the serial but protocol requires that DATA RECEIVERS must be able to handle the amount of data sent by DATA SENDERS in a frame. If the frame is not cancelled, and after the Data subframe has been sent, the DATA RECEIVER sends 101 in the Frame Status if the data was too long for it. If the frame was not cancelled, and if no other module sends a "one" in the high-order bit of the Frame Status, a DATA RECEIVER sends 01 in the next 2 bits.

In a successful frame, the Frame Status values from the DATA SENDER and DATA RECEIVER are logically OR'ed by the serial bus to produce 011. This value is reported by the FRAME MONITOR paired with the HEADER SENDER which initiated the frame.

3.6 A CANCELLED FRAME

A Cancelled frame contains only three subframes:

Header	Frame Type	Jam Bit
S field=TYPE 2, 3 or 5 R field=TYPE 4 or 5	111	0

The Header subframe is sent by a HEADER SENDER. It may be intended to set a semaphore or transfer data. In the former case, the cancelling of the frame indicates that the semaphore is already set. In the latter case, the cancelling of the frame indicates that the data transfer cannot be performed because one or more of the DATA SENDERS or DATA RECEIVERS selected by the S and R codes is not ready for the transfer. A selected DATA RECEIVER may not yet have "disposed of" data it received in a previous transmission. Or, a DATA SENDER may not have been loaded with data to send.

In any of these cases, the HEADER RECEIVER with the flip-flop, DATA SENDER or DATA RECEIVER recognizes the problem and "cancels" the frame by driving all three bits of the Frame Type subframe to "one" (Low). In the case of a Data Transfer frame, the selected DATA SENDER(S) samples all three Frame Type bits as ones and doesn't send the data bytes. The 111 in the size field tells all selected DATA RECEIVERS and all of the FRAME MONITORS on the serial bus that there will be no data bytes nor any status. All FRAME MONITORS use this fact to signal their HEADER SENDERS that the serial bus is available for another frame. The FRAME MONITOR with the HEADER SENDER which initiated the frame also signals the problem to its onboard logic.

3.7 A "JAMMED" FRAME

A "jammed" frame may look like any of the frames described above. It differs from them in that (at least) the final single-bit "Jam Bit subframe" is one rather than zero as shown in the above frames. A Jammed frame occurs when one or more FRAME MONITORS in the system detects that the serial bus is "out of frame synchronization" due to an error induced by system noise, and sends a long series of ones on SERDAT*. A Jammed frame is ignored by all modules on the serial bus: no HEADER RECEIVER sets or clears an associated flip-flop, no DATA SENDER considers itself to have sent data, nor does any DATA RECEIVER consider itself to have received data. The frame which was jammed will be re-sent after the serial bus is "resynchronized".

******* THIS PAGE INTENTIONALLY LEFT BLANK *******

4. SERIAL BUS SUBFRAMES

The activity on the serial bus, like a carefully rehearsed drama, can be described in two ways:

1) We can use a script which presents all of the words spoken, with marginal comments indicating who speaks at each time. (This is the approach used in this section.)

2) We can make a list of cues for each "actor" in the drama that says "WHEN someone says this... THEN you say this..." (This is the approach used in section 6.)

Both approaches are useful to understand how the serial bus works.

4.1 SUBFRAMES COMMON TO ALL FRAMES

All frames start with a Header subframe and a Frame Type subframe, and end with a Jam Bit subframe. The Header subframe determines which modules will participate in the rest of the frame, which in turn determines the type of frame that is sent. The Frame Type subframe makes the type of frame known to all serial bus modules, and specifies the length of the frame. The Jam Bit subframe "validates" the complete frame and ensures "frame synchronization" among all serial bus modules.

4.1.1 The Header Subframe

A Header subframe is composed of 26 bits arranged in 6 fields:

Start Bit	Message Priority	S	R	DATA SENDER Arb enable	HSVAL
1	3	10	10	1	1

The Header subframe consists of the following fields:

1) A single "start bit". When the start bit is transmitted, it signals serial bus modules that a frame is beginning.

2) A three-bit "Message Priority" field. If two or more HEADER SENDERS attempt to send a frame at the same time, this field is used to arbitrate control of the serial bus so that the frame with higher priority is sent first.

3) A ten-bit S field which contains a selection code for serial bus modules. The code in this field and the type of serial bus frame are interrelated as shown below.

FRAME TYPE	CONTENTS OF S FIELD
Data Transfer frame	Sender selection code
Semaphore Set frame	Semaphore selection code
Flip-flop Set frame	Flip-flop selection code
Flip-flop Reset frame	(All ones)
Token Passing frame	Flip-flop selection code

In a Data Transfer frame this field specifies which DATA SENDER(S) should send data in the Data subframe which follows. In the Semaphore Set frame, Flip-flop Set frame, and Token Passing frame, this code selects the semaphore(s) or flip-flop(s) that are to be set. (The S code may select several groups of modules if their HEADER RECEIVERS have all been set up to recognize it.)

4) A ten-bit R field which contains a selection code for serial bus modules. The code in this field and the type of serial bus frame are interrelated as follows:

FRAME TYPE	CONTENTS OF R FIELD
Data Transfer frame	Receiver selection code
Semaphore Set frame	Requester code
Flip-flop Set frame	(All ones)
Flip-flop Reset frame	Flip-flop selection code
Token Passing frame	Flip-flop selection code

In a Data Transfer frame this field specifies which DATA RECEIVER(S) should capture the data in the Data subframe which follows. In a Flip-flop Reset or Token Passing frame, this code selects the flip-flop(s) which are to be reset. In a Semaphore Set frame this code does not select a module, but rather guarantees that only one HEADER SENDER wins the serial bus arbitration and sets the semaphore. (The R code may select several groups of modules if their HEADER RECEIVERS have all been set up to recognize it.)

5) A DATA SENDER arbitration enable bit. When two or more DATA SENDERS share a common selection code, they are both selected to send at the same time. In this case, the resulting Data subframe can contain either the logical OR of their values, or the largest value among the DATA SENDERS. To specify that a logical OR is required, the HEADER SENDER sends a "zero" in this bit position. To specify that the largest value is required, it sends a "one". (A "one" tells each DATA SENDER to stop sending data if it samples a "one" on SERDAT* in a bit in which it is sending a "zero". A "zero" tells the DATA SENDERS to keep sending regardless.)

6) A HEADER SENDER validation bit. This bit is always sent as a "one" (Low) by the HEADER SENDER at the conclusion of the Header subframe. This bit is specified as part of the Header subframe to insure that some error on the serial bus won't cause all the HEADER SENDERS to think they have lost the arbitration and retire from the bus. If this occurs, it leaves a string of "zeroes" on SERDAT* for the rest of the Header subframe. This could be misinterpreted by other modules as a valid Header subframe.

The validation bit solves this problem because if all of the HEADER SENDERS retire it will be a "zero", causing all serial bus modules to ignore the frame.

4.1.2 The Frame Type Subframe

The second subframe present in all frames is the Frame Type subframe. It is composed of a single 3-bit field:

bit 2 MSB	bit 1	bit 0 LSB

Although this 3 bit subframe is present in all frames, it is not always driven by a module. For example, when a simple TYPE 2 module group is selected by a Flip-flop Set frame these bits are 000 because a simple TYPE 2 group doesn't send the Frame Type bits.

When one (or more) DATA SENDER(S) is selected by the S code, it sends a three bit Frame Type code in this subframe. This code indicates how many bytes of data the DATA SENDER is planning to send. The figure below shows the meaning of the various Frame Type codes.

CODE	MEANING
000	No DATA SENDER selected
001	1 byte data transmission to follow
010	2 byte data transmission to follow
011	4 byte data transmission to follow
100	8 byte data transmission to follow
101	16 byte data transmission to follow
110	32 byte data transmission to follow
111	Frame cancelled

Suppose that a DATA RECEIVER has received data in a previous frame and has not "disposed of" that data yet. In order to keep from "losing" the data from the previous frame, it has signalled its own HEADER RECEIVER that it is not ready for another frame. If a HEADER SENDER sends another frame with the DATA RECEIVER'S code in the R field, before the DATA RECEIVER has "disposed of" the previous data, its HEADER RECEIVER "cancels" the new frame by sending "ones" during all three of the size bits. The DATA SENDER samples the Frame Type subframe as it sends it, and stops sending if it samples a 111. (This "Cancelled" frame ends after the Frame Type bits.)

Suppose a DATA SENDER has not been provided with data by its onboard logic. Like a DATA RECEIVER which has not "disposed of" earlier data, it has signalled its own HEADER RECEIVER of this fact via an onboard signal. If a HEADER SENDER sends a frame with the DATA SENDER'S code in the S field while the DATA SENDER "has no data", its HEADER RECEIVER cancels the frame by sending 111 in the Frame Type subframe. The DATA RECEIVER(S) which is selected by the R field sees this code and ignores the frame.

A system can be set up so that more than one DATA SENDER is selected by the same S field code. This allows their data to be logically OR'ed or the largest among their data values to be determined. If multiple DATA SENDERS are selected by the S field and one or more of them is not ready to send, those which are ready see the 111 code and don't send any data.

The HEADER RECEIVER in a Type 2 group intended for semaphore or token passing operation may also send 111 in the Frame Type subframe to cancel a frame. If it sees its code in the S field and its flip-flop is already set, or (in token passing) if it sees its code in the R field and its flip-flop is already reset, it cancels the frame.

In summary, a HEADER RECEIVER module sends 111 to cancel a frame if that frame contains a matching code in either the S or R field, and its DATA SENDER, DATA RECEIVER, or other onboard logic has signalled that it should cancel such a frame.

When a frame is cancelled, this fact is captured by the FRAME MONITOR which is paired with the HEADER SENDER that initiated the frame, and is reported to its onboard logic.

4.1.3 The Jam Bit Subframe

This "subframe" consists of a single bit. It can be regarded as a subframe or as "the bit after a frame". None of the modules which send other subframes ever drive this bit to one (Low), but HEADER RECEIVERS, DATA SENDERS, DATA RECEIVERS, and FRAME MONITORS all sample it to be sure it is a zero. If the Jam Bit is ever sampled by these modules as a one, they ignore the preceding frame transmission.

The only way a Jam Bit can ever be one is if one or more FRAME MONITOR(S) detect a start bit in the middle of a frame. This can only occur if some other FRAME MONITOR has become desynchronized due to system noise. In this case, the FRAME MONITOR(S) which detect the condition drive a string of 512 one bits on SERDAT*. Since all serial bus frames are shorter than 512 bits, this is bound to affect the Jam Bit seen by all serial bus modules.

4.2 THE FRAME STATUS SUBFRAME

While the cancellation of a frame can be determined from the Frame Type field, other problems can arise when a frame is sent. Such problems are also reported to onboard logic by the FRAME MONITOR which is paired with the HEADER SENDER that initiated the frame. The Frame Status subframe is included at the end of every frame except a Cancelled frame, and is used to diagnose problems (exceptional conditions) that might have arisen during the frame's transmission.

The Frame Status subframe is composed of one 3-bit field:

bit 2 MSB	bit 1	bit 0 LSB

Whenever a HEADER RECEIVER receives a Header subframe with an S (or R) selection code that matches its own, it makes its S SELECT (or R SELECT) output True. Depending on whether the HEADER RECEIVER is part of a TYPE 2, 3, 4, or 5 module group, these outputs may or may not enable an on-board DATA SENDER (or DATA RECEIVER).

Since the HEADER RECEIVER has no way of knowing which type of module group it is used in, it samples the Frame Type subframe. If a DATA SENDER is selected it will send one of the values 001 through 110 in the Frame Type. If the HEADER RECEIVER samples 001-110, it passes over the Data subframe and samples these three Frame Status bits and the following Jam Detect bit. (In this case the HEADER RECEIVER is simply acting as a "tracker" to reflect how DATA SENDER(S) and DATA RECEIVER(S) on other boards responded to the frame.) If the Frame Status is 011 and the Jam Detect bit is zero, the HEADER RECEIVER pulses its S STROBE or R STROBE output to set or clear an on-board "tracking" flip-flop.

If the HEADER RECEIVER samples 000 in the Frame Type, it knows that no DATA SENDER was selected by the frame. It samples the MSB of the (immediately following) Frame Status field and then sends a either a 10 or 01 in the next 2 bits to indicate that it was selected by the S or R field, respectively. It then samples the Jam Detect bit. If the MSB of the Frame Type and the Jam Detect bits are both zero, all is well, and the HEADER RECEIVER pulses its S STROBE or R STROBE output (respectively) to an onboard flip-flop.

If two or more DATA SENDERS are selected by the S field code of the Header subframe, the serial bus protocol requires that they agree about how much data is to be sent. As each DATA SENDER sends its code in the Frame Type field to signal how much data it has to send, it also samples SERDAT* to detect a conflict with other DATA SENDERS. If it samples a "one" in any of the 3 bits while it is sending a "zero", it stops sending any further bits in the Frame Type subframe or Data subframe. However, in this case it does sample all the bits in the Frame Type. If the sampled value is not 111 (i.e. not a cancellation), it uses the value it samples to determine the length of the Data subframe, passes over the data, and thereafter sends 110 during the Frame Status subframe to signify a conflict among selected DATA SENDERS as to size of the Data subframe.

If the DATA SENDER samples the same value in the Frame Type that it is sending, it sends its data. It then samples the MSB of the Frame Status, which indicates a size problem on the part of other DATA SENDERS or RECEIVERS. If the MSB is zero it sends 10 in the next two bits of the Frame Status to indicate that it survived any arbitration in the Data subframe.

Suppose that a DATA SENDER is selected by the S field code of the Header subframe and a DATA RECEIVER is selected by its R code. Let's further suppose that the DATA RECEIVER is only capable of receiving Data subframes up to a certain length (less than 32 bytes).

The DATA RECEIVER samples the Frame Type subframe. If the Frame Type code indicates the DATA SENDER is trying to send more data than it can handle, the DATA RECEIVER counts bits across the size of data indicated by the Frame Type, and then sends 101 in the Frame Status subframe to signal the problem.

If a selected DATA RECEIVER samples 000 in the Frame Type field, there is no DATA SENDER to send data to it. It handles this situation by sending 100 in the Frame Status subframe.

Otherwise (for Frame Types 001-110) the DATA RECEIVER samples the number of bytes of data indicated by the Frame Type. It then samples the MSB of the Frame Status subframe. If the MSB is "zero", it sends 01 in the next 2 bits of the Frame Status to show that it was selected. It then samples the Jam Detect bit. If the MSB of the Frame Status and the Jam Detect bit were both zero, it presents the data it sampled to its onboard logic. The R STROBE output of the paired HEADER RECEIVER indicates the arrival of data to onboard logic.

In summary, the Frame Status value must be interpreted differently for a Data Transfer frame vs. other types of frames. In each case there are 8 possible status values.

STATUS Value	Interpretation
FRAME TYPE 000	(Not a Data Transfer frame)
000	No HEADER RECEIVER selected by either S or R
001	HEADER RECEIVER selected by R only
010	HEADER RECEIVER selected by S only
011	HEADER RECEIVERS selected by S and R
100	RESERVED: should not occur
101	DATA RECEIVER selected by R, nothing selected by S
110	RESERVED: should not occur
111	DATA RECEIVER selected by R, HEADER RECEIVER (only) selected by S
FRAME TYPE 001-110	(Data Transfer frame)
000	No DATA RECEIVER selected by R, all DATA SENDERS selected by S lost Data field arbitration
001	DATA RECEIVER selected by R, all DATA SENDERS selected by S lost Data field arbitration
010	No DATA RECEIVER selected by R, DATA SENDERS selected by S completed the Data subframe normally
011	Data Transfer completed correctly
100	RESERVED: should not occur
101	DATA RECEIVER(S) unable to handle data size
110	Data size conflict among DATA SENDERS
111	(Both 101 and 110 conditions)

4.3 THE DATA SUBFRAME

The length of a Data subframe is specified by the selected DATA SENDER(S) in the Frame Type subframe. There may be 1, 2, 4, 8, 16, or 32 bytes in a Data subframe. As described earlier, there may be one or more DATA SENDER(S) sending in this subframe, depending on how many HEADER RECEIVERS are configured to respond to the S-field code. If several DATA SENDERS are selected, the Data subframe will either contain the logical OR of their data, or the largest value among them. (This is determined by whether the DATA SENDERS' arbitration was disabled or enabled by the Sender Arbitration Enable bit sent during the Header subframe.)

******* THIS PAGE INTENTIONALLY LEFT BLANK *******

5. SERIAL BUS LINES

The serial bus includes three lines: Serial Clock (SERCLK), Serial Data (SERDAT*), and System Reset (SYSRESET*).

SYSRESET* is used to initialize all modules on the serial bus.

SERCLK is driven by a high-current totem-pole driver from the SERIAL CLOCK module, of which there is one per system.

SERDAT* is a wired-OR line which can be driven with an open-collector driver by any module. A "one" on SERDAT* results when one or more modules drive it Low. A "zero" results when no module is driving SERDAT*, so that the backplane terminating resistors pull the signal High.

5.1 SIGNAL TIMING

When the serial bus is used with a system bus, there are no prescribed timing constraints between its signals and any of the other signals on the backplane. The various modules on the serial bus change and sample the SERDAT* signal level when transitions occur on SERCLK. The AC timing characteristics of SERCLK and SERDAT* are shown in Table 5-1 and Figure 5-1.

5.2 BIT TRANSMISSION CODING

Each "bit cycle" of SERCLK is used to send one bit on SERDAT*, and includes four transitions designated C1, S1, C2, and S2. There are three types of bits that can be sent during each bit cycle: a "one bit", a "zero bit", and a "start bit". These bits can be distinguished by serial bus modules by sampling the level of SERDAT* on both the S1 and S2 transitions.

TYPE OF BIT	LEVEL SAMPLED AT S1	LEVEL SAMPLED AT S2
one	Low	Low
zero	High	High
start	High	Low

If sufficient noise is induced onto the serial bus, the modules may be desynchronized (i.e. they may lose track of where frames begin and end). This condition must be detected and the modules resynchronized before frames are "garbled" and misinterpreted. When desynchronization occurs, one or more of the HEADER SENDERS send a start bit in the middle of a frame.

FRAME MONITORS check for a start bit in every bit cycle of SERCLK. When a FRAME MONITOR detects a start bit when no frame is in progress, it tracks the transmission to the end of the frame. If it samples another "High/Low" bit before it has finished counting out the frame, then it "jams" the frame transmission by holding SERDAT* Low for at least 512 bit cycles. This ensures that the frame which was in progress won't be accepted as a valid frame.

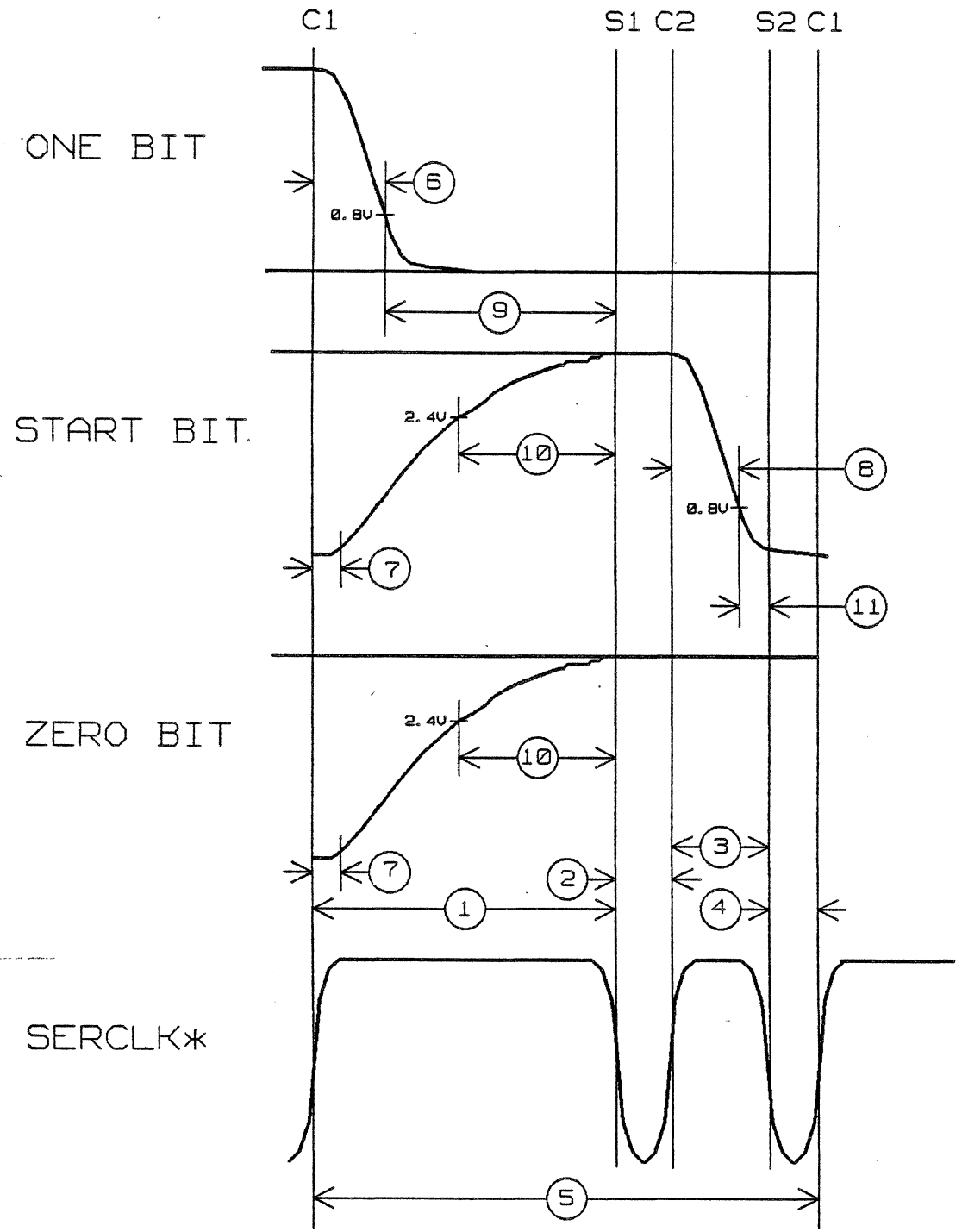

FIGURE 5-1. PRELIMINARY SERIAL BUS TIMING

PARAMETER	DESCRIPTION	(SEE NOTES A, B) MIN	NOM	MAX	NOTES
1	C1 to S1	175	187.5		C
2	S1 to C2	25	31.25		C
3	C2 to S2	50	62.5		C
4	S2 to C1	25	31.25		C
5	C1 to C1 (SERCLK Cycle)		312.5		C
6	C1 to SERDAT* Low	0		TBS	D
7	C1 to SERDAT* Released	0		TBS	D
8	C2 to SERDAT* Low	0		TBS	D
9	SERDAT* Low to S1	TBS			E
10	SERDAT* High to S1	TBS			E
11	SERDAT* Low to S2	TBS			E

NOTES:

A All times given are in nanoseconds.

B NOM and MAX columns apply to 3.2 Mbit/sec rate

C The SERIAL CLOCK module must guarantee this timing between two of its outgoing SERCLK transitions.

D Each serial bus module must guarantee this timing between its incoming and outgoing transitions.

E Each serial bus module is guaranteed this time between the specified incoming transitions.

TABLE 5-1. PRELIMINARY SERIAL BUS TIMING

A desynchronized HEADER SENDER will not consider itself to have sent a start bit at the same time that a "one bit" is being sent by some other module. While it may drive SERDAT* Low at the C2 edge of SERCLK during a "one", such a start bit will not be visible on SERDAT*. The HEADER SENDER is required to sample the level of SERDAT* on the S1 transition of SERCLK in the bit cycle that it sends the start bit. If SERDAT* is High at S1 then it assumes the start bit was sent successfully and sends the rest of the Header subframe. If it was Low the HEADER SENDER tries again on the next bit cycle.

EDGE DESIGNATION	USE
C1	All serial bus modules change SERDAT* on this transition when sending "one" bits or "zero" bits.
S1	All serial bus modules sample SERDAT* on this transition.
C2	HEADER SENDER modules change SERDAT on this transition from a High to a Low when sending a "start bit". Other modules maintain the SERDAT* levels established at C1.
S2	HEADER RECEIVER modules sample SERDAT* on this transition when they are waiting for a new frame to begin (i.e. when they are looking for a "start bit"). FRAME MONITOR modules sample SERDAT* on every S2 transition, first to detect the beginning of a frame and then to detect a start bit that is sent while a frame is in progress.

The C1, S1, C2, and S2 transitions are shown in Figure 5-1.

NOTE: If a board or IC is designed for the serial bus which does not include a FRAME MONITOR module, it is recommended, but not required, that it detect start bits within a frame and "jam" SERDAT* as described for a FRAME MONITOR. This provides greater security and protection against the effects of signal noise that might be sensed at one point on the serial bus, but not at another.

Onboard outputs from serial bus modules should be changed on the C1 transition of SERCLK, and onboard inputs to serial bus modules should be sampled on the S1 transition. Designers of serial bus hardware must guarantee that there will be no metastability problems when sampling inputs from onboard logic. This can be done by specifying setup and hold times that onboard logic must meet with respect to the S1 transition of SERCLK or, preferably, by providing synchronization logic for each input. In the latter case, it is helpful to specify a setup time to (some edge of) SERCLK, which will guarantee that the new state of an input will be recognized and acted upon.

6. SERIAL BUS MODULES

A serial bus subsystem may include the following functional modules:

One SERIAL CLOCK module
HEADER SENDERS
HEADER RECEIVERS
DATA SENDERS
DATA RECEIVERS
FRAME MONITORS

NOTE: These "functional modules" are used as vehicles for discussion of the serial bus protocol, and need not be considered a constraint to logic design. For example, a single IC could provide several of these functions.

The following sections describe the behavior of these modules, including their interaction with on-board signals. It is likely that IC's for the serial bus will be designed with various internal architectures and onboard signals. This specification often describes the details of an onboard interface in general terms. For instance, we may say that "onboard logic reads from a DATA RECEIVER" without specifying exactly how this is accomplished. This is done to avoid placing unnecessary constraints on the logic designs of serial bus interface hardware. In some cases actual signal lines are shown, to improve the explanation of the serial bus. Where this is done, the intent is to make clear what information crosses the boundary between the module and other on board logic. Other methods for conveying this same information across the boundary are also permissible.

It is expected that IC's designed for the serial bus will include some of what is described in this specification as "onboard logic". For example, the functional modules described below have several outputs which produce a pulse which is 1 SERCLK cycle in duration. Real IC's might use these "conceptual pulses" to produce outputs of longer duration.

6.1 SERIAL CLOCK Module

The SERIAL CLOCK module must be located on either the Slot 1 board or the board in the highest-numbered slot. It drives the SERCLK signal with a high-current totem-pole driver. Whenever its SYSRESET* input is True, the SERIAL CLOCK module drives SERDAT* to "one" (Low). After its SYSRESET* input goes False, the SERIAL CLOCK module releases SERDAT* to "zero" (High) on a subsequent C1 edge of SERCLK, in keeping with the timing given in Figure 5-1.

6.2 HEADER SENDER Module

Onboard logic uses a HEADER SENDER module to initiate the transmission of a frame on the serial bus. The HEADER SENDER sends only the "Header subframe" portion of a frame: the remainder of the frame is sent by other modules including HEADER RECEIVERS, DATA SENDERS, and DATA RECEIVERS. Basically, the Header subframe sent by this module serves to "select" one or more HEADER RECEIVERS on the serial bus.

A HEADER SENDER must always be paired with a FRAME MONITOR module. Even though a HEADER SENDER often occupies the same board with a HEADER RECEIVER (etc.), these modules only communicate with each other over the serial bus. Because of this, the HEADER SENDER initiates frames involving its on-board HEADER RECEIVER in exactly the same way that it does with off-board HEADER RECEIVERS.

6.2.1 HEADER SENDER signals

As shown in Figure 6-1, a HEADER SENDER takes the SYSRESET* and SERCLK signals from the serial bus as inputs, the SERDAT* signal as a bidirectional I/O, and the FRAME IN PROGRESS signal from its onboard FRAME MONITOR as an input. It also has the following additional onboard signals:

Priority Port — A set of input lines and associated control signals, through which the 3 bit Message Priority value for subsequent frame transmission(s) is loaded into the HEADER SENDER. This port may consist of dedicated input lines, or a bus interface to an internal Priority register.

Code 1,2 Ports — Two similar sets of signals, through which two 10 bit selection codes for subsequent frame transmissions are loaded.

Sender Arb Port — A similar set of signals, through which the DATA SENDER Arbitration Enable bit for subsequent frame transmissions is loaded into the HEADER SENDER.

NOTE: The Priority, Code 1, Code 2, and Sender Arb Ports will often be implemented as registers that can be loaded from the same "bus interface", which may be connected to an onboard processor or onboard system bus interface.

SEND12, SEND21 — Control inputs which cause the HEADER SENDER to send a "Header subframe". SEND12 makes the HEADER SENDER send the value from Code 1 Port in the "S field" of the Header subframe, and the value from Code 2 Port in the "R field". SEND21 makes the HEADER SENDER send the value from Code 2 Port in the "S field" and the value from Code 1 Port in the "R field".

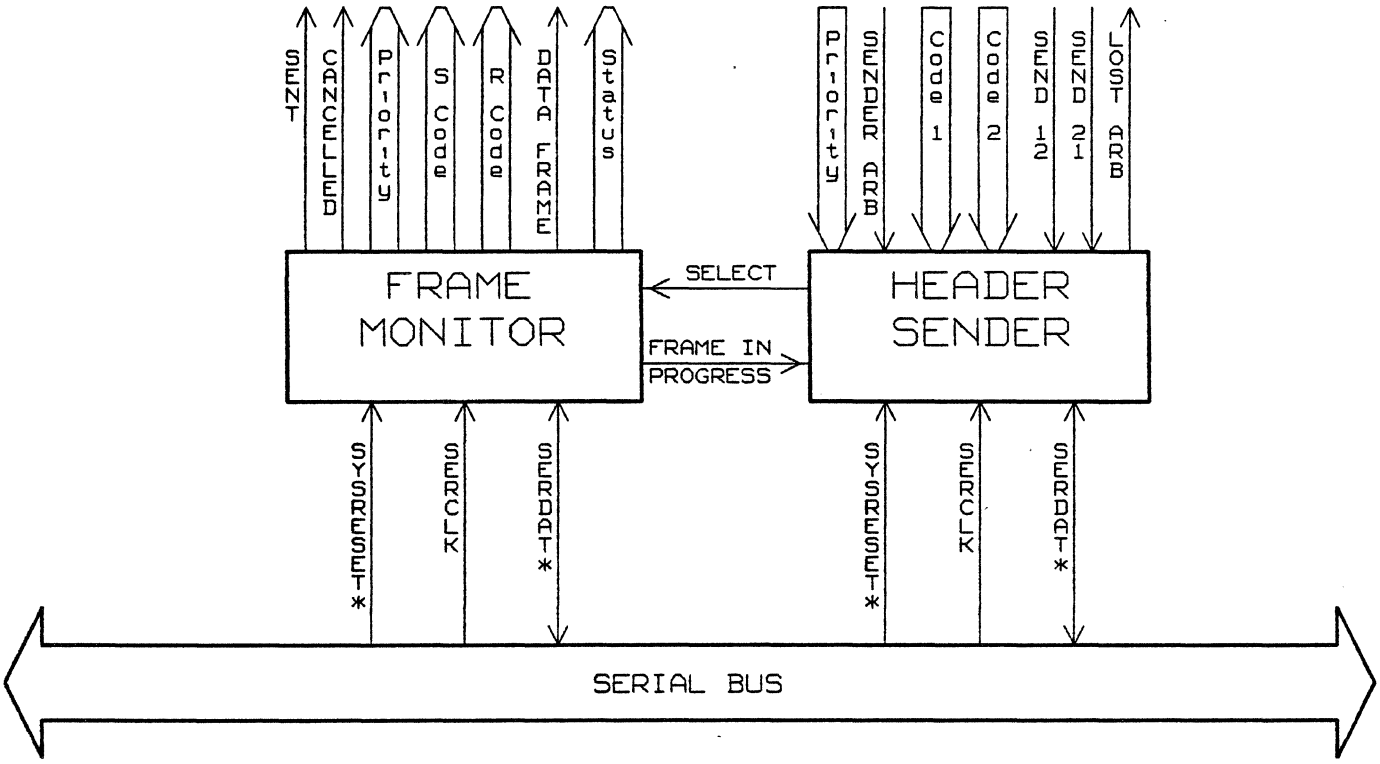

FIGURE 6-1. HEADER SENDER with FRAME MONITOR

NOTE: Implementors of serial bus hardware may be tempted to make this SEND function a bit in an internal register, or "implied" by loading a Header subframe to be transmitted. However, by including one or more actual SEND inputs, an implementation can be made applicable to "dumb" boards as well as "intelligent" ones. The Header subframe to be transmitted can be loaded from a processor via the system bus, and its actual transmission can be "triggered" later by an onboard signal event.

SELECT — An output to a paired FRAME MONITOR module, indicating that this HEADER SENDER has won the serial bus arbitration and initiated the current frame. This causes the FRAME MONITOR to report the result of frame transmission to onboard logic.

LOST ARB — An alternative output to SELECT, indicating that the HEADER SENDER has tried to send a Header subframe, but has lost the arbitration for use of the serial bus to another HEADER SENDER.

NOTE 1: When a HEADER SENDER loses the serial bus arbitration, onboard logic can increase the Priority value for the subsequent retry (e.g. by one, perhaps up to a defined upper limit). This strategy, plus reduction of the Priority value when a frame is Cancelled (see 6.6.1), can ensure that each HEADER SENDER on the serial bus gets a fair share of access to the bus.

NOTE 2: Information other than LOST ARB, about the results of sending a frame, is available from the outputs of the paired FRAME MONITOR.

6.2.2 HEADER SENDER Initialization

Whenever SYSRESET* is Low, a HEADER SENDER initializes itself as follows:

1) It makes its LOST ARB output False.

2) It releases SERDAT*.

3) It discards any Header subframe which was awaiting transmission when SYSRESET* went Low.

4) If any of the Priority, Code 1, Code 2, or Sender Arb Ports are implemented as loadable registers, it clears those registers to zero.

When SYSRESET* goes High, a HEADER SENDER enters "idle state".

6.2.3 HEADER SENDER Operation

The state diagram of a HEADER SENDER is shown in Figure 6-2. In idle state, the HEADER SENDER samples SEND12, SEND21, and SERDAT* on each S1 edge of SERCLK, and samples FRAME IN PROGRESS and SERDAT* on each S2 edge of SERCLK.

1a) IF it samples SEND12 or SEND21 True and SERDAT* "zero" (High) on an S1 edge, and samples FRAME IN PROGRESS False and SERDAT* "zero" (High) on the following S2 edge (i.e. if it samples a "zero" on SERDAT*),

THEN it drives SERDAT* to "one" (Low) at the C2 edge of the next bit cycle, generating a start bit. The HEADER SENDER then proceeds to step 2) below.

1b) IF it samples SEND12 or SEND21 True, and SERDAT* "zero" (High) on an S1 edge, and samples FRAME IN PROGRESS False, SERDAT* "one" (Low) on the following S2 edge,

(i.e., if another HEADER SENDER is sending a Start bit, one bit in advance of when this module would have sent one in case 1a). This can occur only if on-board logic made SEND12 or SEND21 True during the previous SERCLK cycle.)

THEN

IF the logic of this HEADER SENDER is fast enough to send the most significant bit of its Header subframe on the following bit cycle, (i.e. if it can drive SERDAT* within the time specified from C1),

THEN it proceeds to step 2, just as if the other HEADER SENDER'S start bit was its own. (The bus arbitration will then decide which HEADER SENDER must retire from the bus and try again.)

ELSE (the HEADER SENDER isn't fast enough to send the first bit of its Header subframe on the next bit cycle) it simply delays one cycle of SERCLK and returns to idle state sampling as described above. At the end of the current frame it will encounter the less demanding case 1a) above.

2) On the next C1 edge of SERCLK following the start bit, the HEADER SENDER places the most significant bit of its Message Priority value on SERDAT*. Note that other HEADER SENDERS in the system may be doing the same thing.

3) On the following S1 edge of SERCLK, the HEADER SENDER samples SERDAT* as an input.

IF it placed a "zero" on SERDAT* at C1, and it samples a "one" (Low) on SERDAT* on the S1 edge,

THEN this HEADER SENDER has lost the arbitration for use of the serial bus. In this case, it makes its LOST ARB output True for one SERCLK cycle and returns to idle state.

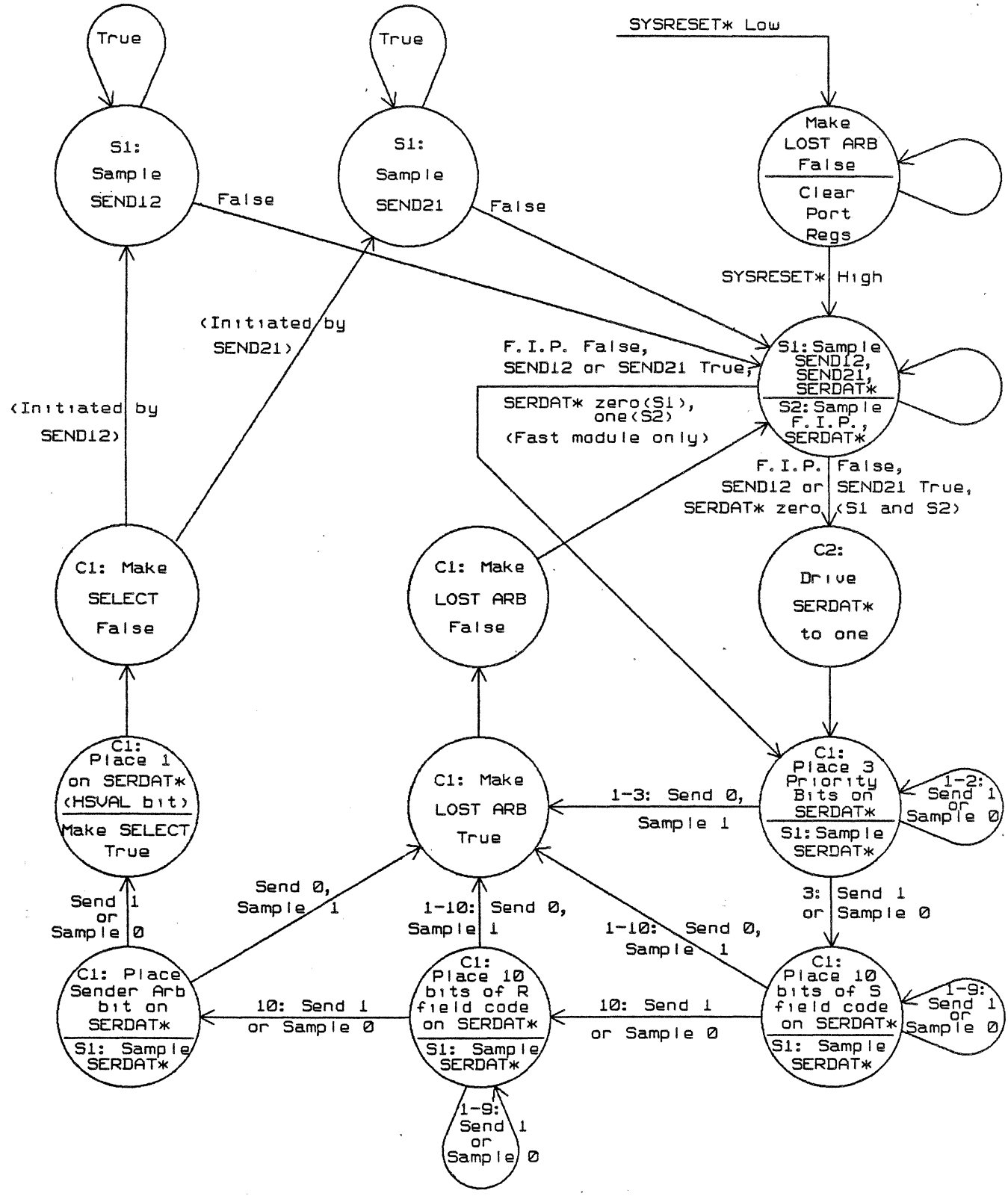

Figure 6-2. HEADER SENDER State Diagram

4) The process described in 2-3) is repeated for each of the remaining bits in the Header subframe. This includes the remaining 2 bits of the Message Priority field, the S and R fields which are 10 bits each, the "sender arbitration enable bit", and the HSVAL bit. The HEADER SENDER always sends the HSVAL bit "one".

5) On the same C1 edge on which it sends the HSVAL bit, the HEADER SENDER makes its SELECT output True. This output indicates that this HEADER SENDER has won the serial bus arbitration and sent its Header subframe successfully. SELECT True makes the paired FRAME MONITOR report the results of the overall frame transmission to onboard logic.

> NOTE: The HSVAL bit guards against the possibility that all HEADER SENDERS dropped out of the arbitration due to noise on the bus. If the HSVAL bit of any message frame is "zero", the frame is ignored by all HEADER RECEIVERS.

6) After sending HSVAL and making SELECT True, on the next SERCLK C1 edge it makes SELECT False. On subsequent S1 edges it samples SEND12 and SEND21, until the signal which initiated the operation (step 1) is made False by onboard logic. It then returns to idle state.

Note that this arbitration method guarantees that the Header subframe from at least one HEADER SENDER is always sent correctly (barring an "all drop out" error due to some externally induced noise glitch). If one considers the contending Header subframes as binary numbers, the subframe which is successfully sent is the one with the highest binary value.

But, if two or more HEADER SENDERS set out to send exactly the same Header subframe, they will not know of each other's presence, and each will proceed as if it had sent the subframe alone. This is fine for some frames. It is not appropriate for other frames (e.g. those which set a "semaphore"). When onboard logic causes a HEADER SENDER to initiate a Semaphore Set frame, it must ensure that the R field contains a code that is not used by by any other HEADER SENDER on the serial bus. If this requirement is met, the normal bus arbitration ensures that all but one HEADER SENDER (the one with the highest "Requester number") will retire from the bus before the Header subframe completes.

6.3 HEADER RECEIVER Module

A HEADER RECEIVER module may stand alone or may be paired with a DATA SENDER and/or a DATA RECEIVER. Its primary function is to compare the S and R fields in each Header subframe on the serial bus against its selection code value, and signal its paired modules and/or onboard logic if it finds a match.

6.3.1 HEADER RECEIVER Signals

As shown in Figure 6-3, a HEADER RECEIVER takes the SYSRESET* and SERCLK signals from the serial bus as inputs and the SERDAT* signal as a bidirectional input/output. It also has a number of additional inputs from, and outputs to onboard logic, as follows:

Code Port — A set of parallel input lines and associated control signals, through which a 10-bit selection code value can be loaded into the HEADER RECEIVER. This may consist of dedicated inputs, or a "bus" interface to an internal selection code register.

ENABLE S,R — Two inputs from onboard logic which indicate whether to accept a frame with this module's selection code in the S or R field, respectively. If the HEADER RECEIVER matches its code in the S (or R) field, and the ENABLE S (ENABLE R) input is False, it "cancels" the frame.

S,R SELECT — Two outputs to a paired DATA SENDER and DATA RECEIVER, respectively, which indicate that the HEADER RECEIVER has detected a Header subframe on the serial bus which includes the selection code from Code Port in the S or R field, respectively. These outputs are made True for 1 SERCLK period.

S,R STROBE — Two outputs to onboard logic, which indicate the same information as S SELECT and R SELECT, but are made True at the end of the frame, and only if the frame is completely successful.

MAX/OR — An output to a paired DATA SENDER, which reflects the state of the "Sender Arbitration" bit in a Header subframe.

6.3.2 HEADER RECEIVER Initialization

Whenever SYSRESET* is Low, a HEADER RECEIVER initializes itself as follows:

1) It makes its S SELECT, R SELECT, S STROBE, and R STROBE outputs False.

2) It releases SERDAT*, and ignores it as an input.

3) If the Code Port is implemented as a loadable register, it clears the register to zero.

When SYSRESET* is released, the HEADER RECEIVER enters "idle state".

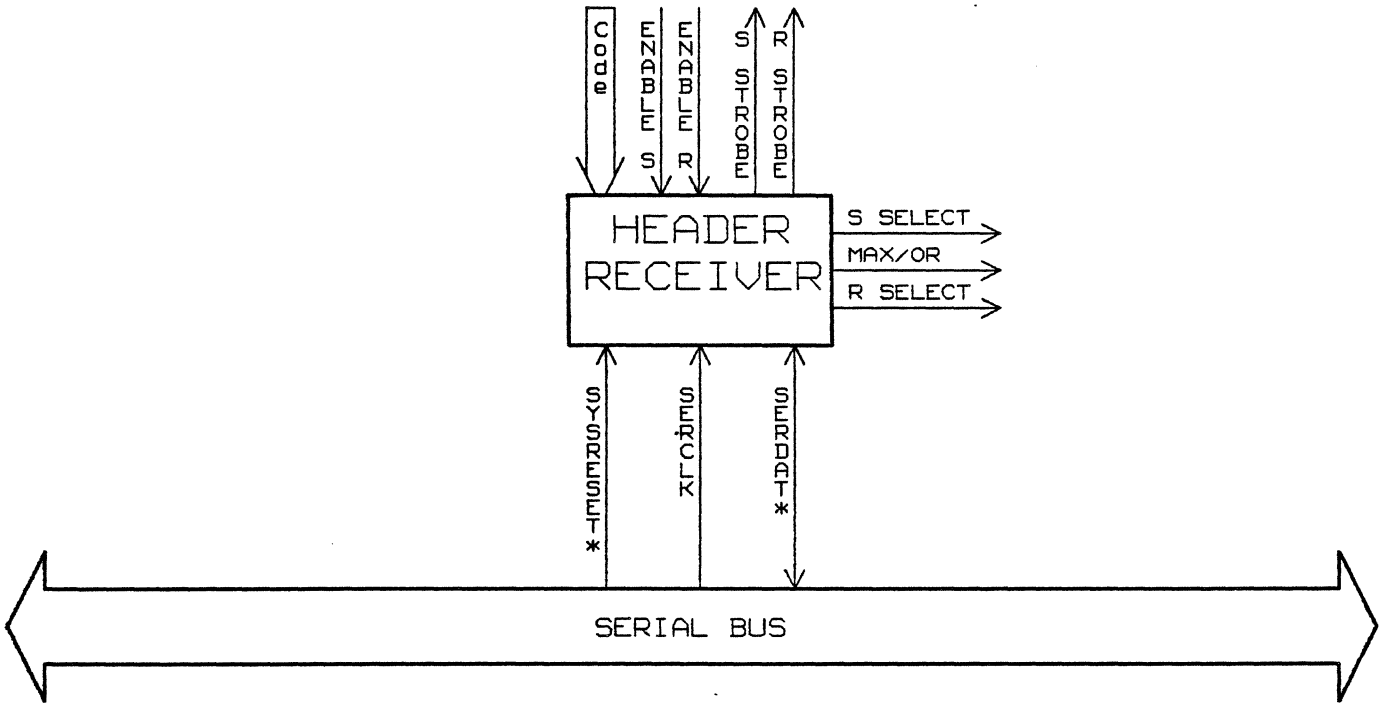

FIGURE 6-3. Signals used by a HEADER RECEIVER

6.3.3 Programming the Selection Code

If the selection code is implemented as a loadable register, the HEADER RECEIVER must be programmed with the desired value after initialization. For the purposes of this specification, this is considered part of system setup before any frames are sent on the serial bus. Reprogramming a HEADER RECEIVER with a new selection code, while the serial bus is in operation, is allowed but not covered by this specification.

6.3.4 HEADER RECEIVER Operation

The state diagram of a HEADER RECEIVER is shown in Figure 6-4. Starting from idle state, this module proceeds as follows:

1) It samples SERDAT* repeatedly on the S1 and S2 edges of SERCLK, until it samples a High on S1 and a Low on S2. This combination indicates a start bit on the serial bus.

   NOTE: On the first C1 edge after returning from step 15 or 22 below, a HEADER RECEIVER makes its S STROBE and R STROBE outputs False.

2) It then counts off three bit times on SERCLK, thus ignoring the Message Priority field of the frame.

3) The HEADER RECEIVER then samples the next 10 bits on SERDAT*, on S1 edges of SERCLK. As it samples these bits, it compares each to the corresponding bit of its selection code from Code Port. If all 10 bits are equal it makes an internal signal called "S Match" True, if not it makes S Match False.

4) The HEADER RECEIVER then samples the next 10 bits on SERDAT* as it did in step 3), except that it makes an internal signal called "R Match" True or False.

5) It then samples the next bit on SERDAT* on the S1 edge of SERCLK. This is the "Sender Arb Enable" bit. If the bit is "one" it makes it MAX/OR output True, otherwise it makes MAX/OR False.

6) On the next C1 edge of SERCLK, if S Match is True, the HEADER RECEIVER makes its S SELECT output True. On the same edge, if R Match is True it makes its R SELECT output True.

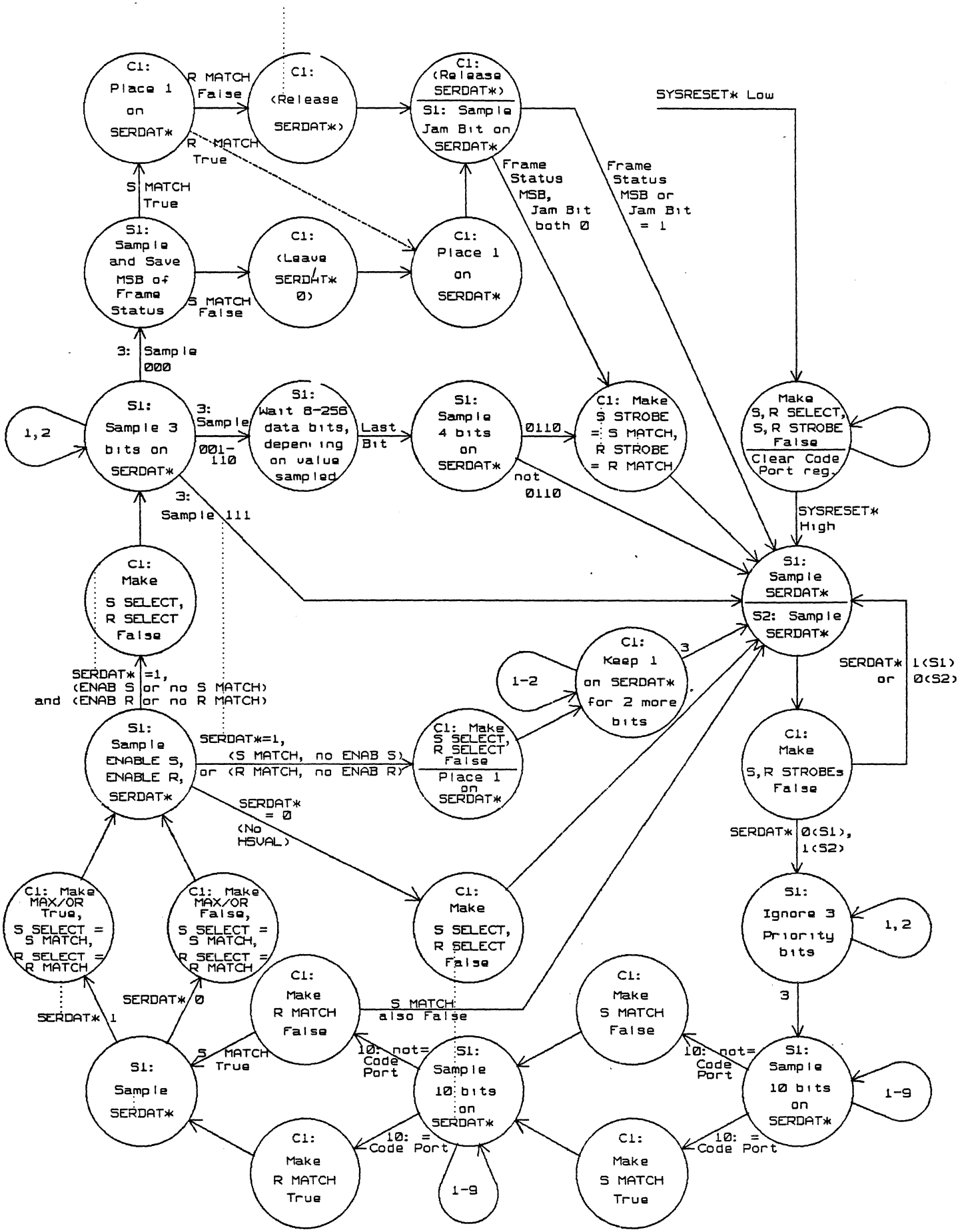

Figure 6-4. HEADER RECEIVER State Diagram

7) On the next S1 edge of SERCLK, the HEADER RECEIVER samples its ENABLE S and ENABLE R inputs, and the (HSVAL) bit on SERDAT*.

Depending on what it samples in this and previous steps, the HEADER RECEIVER proceeds as follows:

HSVAL	S Match	R Match	ENAB S	ENAB R	Action
x	False	False	x	x	Return to step 1 (idle state)
zero	<-not both False->		x	x	Step 8
one	True	x	False	x	Step 9
one	x	True	x	False	Step 9
one	<---- any other combination ---->				Step 10

NOTE: An "x" in the table above indicates that this item does not matter to the HEADER RECEIVER in this case.

8) (The HSVAL bit was zero.) The HEADER RECEIVER simply makes S SELECT and R SELECT False on the next C1 edge and returns to idle state.

9) (In this case the HEADER RECEIVER has matched the S field in the Header and sampled the ENABLE S input False, or it has matched the R field and sampled the ENABLE R input False.) In either case, on the next C1 edge it makes S SELECT and R SELECT False and drives a one (Low) on SERDAT*. It maintains SERDAT* one for the following 2 C1 edges. This "cancels" the frame on the serial bus. On the C1 edge after that, it releases SERDAT* and returns to idle state.

10) (In this case the HEADER RECEIVER has matched the S or the R field, set its S SELECT and/or its R SELECT output True, and sampled the corresponding ENABLE input True.) It now makes S SELECT and R SELECT False on the next C1 edge, and then samples SERDAT* on the following 3 S1 edges of SERCLK. This is the Frame Type code.

11) If the Frame Type is 111, the frame has been cancelled by another HEADER RECEIVER, and this module returns to idle state.

12) If the Frame Type code is 000, the HEADER RECEIVER proceeds to step 17). Otherwise (the Frame Type code is 001-110) this is a Data Transfer frame. The HEADER RECEIVER in this case acts as a "tracker" module wherein its S STROBE (or R STROBE) output signals a successful data transfer from (or to) its code.

13) The HEADER RECEIVER counts off the necessary number of SERCLK cycles to ignore the Data subframe on SERDAT*. It determines the number of cycles to delay from the Frame Type as follows:

Frame Type	SERCLK Count
001	8
010	16
011	32
100	64
101	128
110	256

14) After the above number of SERCLK cycles, the HEADER RECEIVER samples SERDAT* on each of the next 4 S1 edges of SERCLK. These 4 bits are the Frame Status field and the Jam Detect bit.

15) If the 4 sampled bits are exactly 0110, then the serial bus data transfer is successful. On the following C1 edge the HEADER RECEIVER makes the S STROBE output True if S Match is True, makes the R STROBE output True if R Match is True, and returns to idle state.

16) If the 4 sampled bits are anything other than 0110, the serial bus data transfer was not successful. The HEADER RECEIVER does not signal on S or R STROBE, but simply returns to idle state.

17) (The Frame Type was 000, and the frame contains no data.) The HEADER RECEIVER samples the MSB of the Frame Status field on the next S1 edge, and retains what it sampled for later.

18) On the next C1 edge, if S Match is True, the HEADER RECEIVER drives a "one" on SERDAT* to show that it is present. (If S Match is False it does not drive SERDAT*.)

19) On the next C1 edge, if R Match is True, the HEADER RECEIVER drives a "one" on SERDAT* to show that it is present. (If R Match is False it releases SERDAT*.)

20) On the next C1 edge, the HEADER RECEIVER releases SERDAT* and then samples it on the next S1 edge. This is the Jam Detect bit.

21) If the Jam Detect bit is "one" or the previously sampled MSB of the Frame Status field was "one", the HEADER RECEIVER does not drive its S or R STROBE outputs, but simply returns to idle state.

22) (The Jam Detect bit and the previously sampled MSB of the Frame Status are both "zero".) If S Match is True, the HEADER RECEIVER drives its S STROBE output True on the following C1 edge. On the same edge, if R Match is True, it drives its R STROBE output True. In either case it returns to idle state.

6.4 DATA SENDER Module

A DATA SENDER module must be paired with a HEADER RECEIVER. Its function is to take data from on-board logic, and send it on the serial bus when it is signalled by its HEADER RECEIVER.

6.4.1 DATA SENDER Signals

As shown in Figure 6-5, a DATA SENDER takes the SYSRESET* and SERCLK signals from the serial bus as inputs, and the SERDAT* signal as a bidirectional I/O. It also has a number of inputs and outputs with onboard logic, as follows:

Data Port — A set of parallel input lines and associated control signals, through which onboard logic provides data to be sent on the serial bus. This may consist of dedicated inputs or a bus interface to an internal Data register.

> NOTE: Frames on the serial bus can include 1, 2, 4, 8, 16, or 32 bytes of data. Real DATA SENDER implementations may be limited to sending less than 32 bytes. This module signals how many bytes are being sent before it sends data. This specification does not cover the details of how data is loaded via Data Port, nor how the number of bytes loaded is determined by the DATA SENDER.

S SELECT — An input from the paired HEADER RECEIVER, which signals when data is to be sent.

MAX/OR — An input from the paired HEADER RECEIVER, which signals how data is to be sent. If this signal is True, the DATA SENDER enables its serial bus arbitration logic while sending data.

DSENT — An output to onboard logic, which signals that the DATA SENDER has successfully sent data on the serial bus.

> NOTE: For frames which select a single DATA SENDER, and frames without Data field arbitration (MAX/OR False), DSENT signals identically to the S STROBE output of the paired HEADER RECEIVER. However, when a frame selects multiple DATA SENDERS for their largest value, all of their paired HEADER RECEIVERS will signal on S STROBE, but only the "winning" DATA SENDER will signal on DSENT.

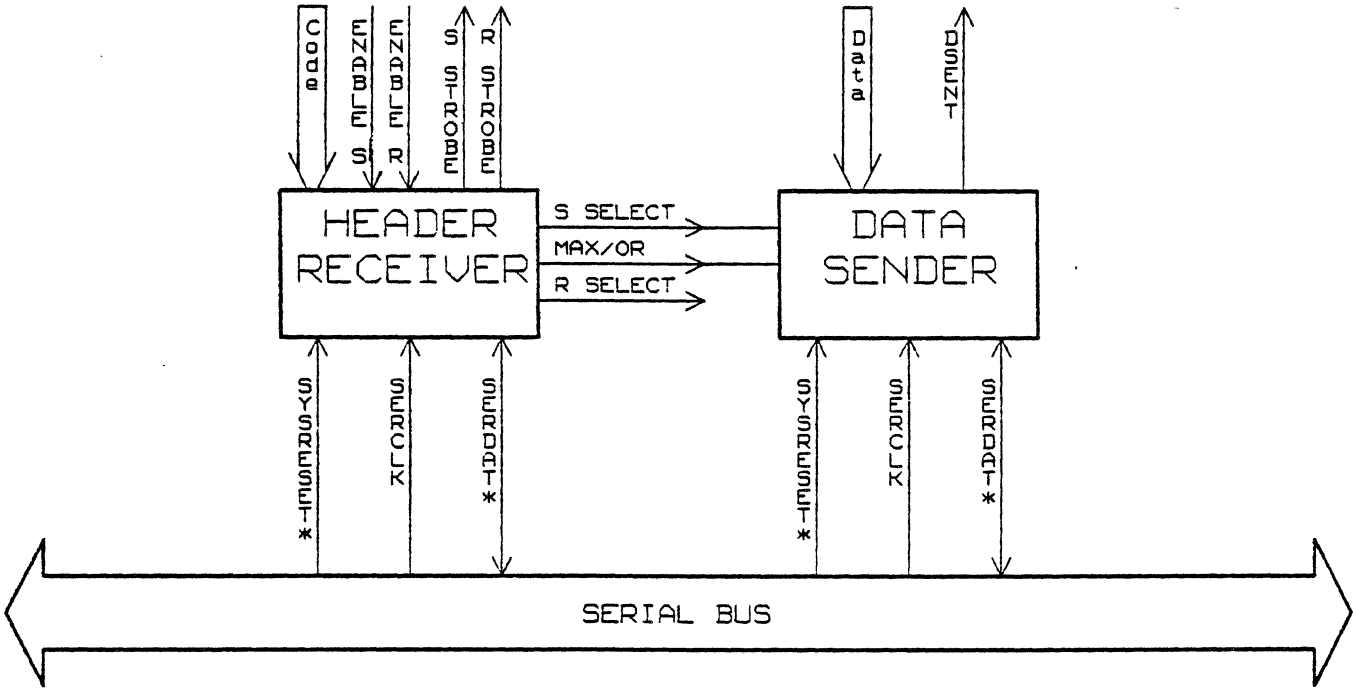

FIGURE 6-5. DATA SENDER with HEADER RECEIVER

6.4.2 DATA SENDER Initialization

Whenever SYSRESET* is Low, a DATA SENDER initializes itself as follows:

1) It makes its DSENT output False.

2) It releases SERDAT*, and ignores it as an input.

3) If Data Port is implemented as a loadable register, it clears that register to zero.

When SYSRESET* is released, the DATA SENDER enters idle state, described below.

6.4.3 Programming the Data Port

DATA SENDERS can be used in two ways. In one approach, the DATA SENDER is always ready to send data, and sends "whatever data is available" whenever a frame which selects it appears on the serial bus. Onboard logic with such a DATA SENDER presents new data at Data Port "whenever the data to be sent changes". In this case the DATA SENDER design should "double buffer" Data Port to assure that when it sends data in a frame, it does not send a mixture of "old" and "new" data.

In other applications, each time onboard logic presents new data at Data Port, the DATA SENDER sends it once and only once. Onboard logic with such a DATA SENDER presents new data whenever it is available and previously loaded data has been sent. If a frame which selects such a DATA SENDER appears on the serial bus, and onboard logic "has not provided new data", it "cancels" the frame.

This difference is determined by the way in which the onboard signals are connected and driven. The following description of a DATA SENDER allows for either mode. In the first case, the ENABLE S input of the paired HEADER RECEIVER is permanently set to True, and the DSENT output of the DATA SENDER is not used by onboard logic. In the second case, onboard logic makes ENABLE S True after it has loaded data, and False again when the DATA SENDER pulses the DSENT output.

6.4.4 DATA SENDER Operation

The state diagram of a DATA SENDER is shown in Figure 6-6. Starting from idle state, the DATA SENDER proceeds as follows:

1) In idle state the DATA SENDER samples its S SELECT input and the SERDAT* line on every S1 transition of SERCLK. When it samples S SELECT True and SERDAT* "one" (Low), it proceeds to step 2.

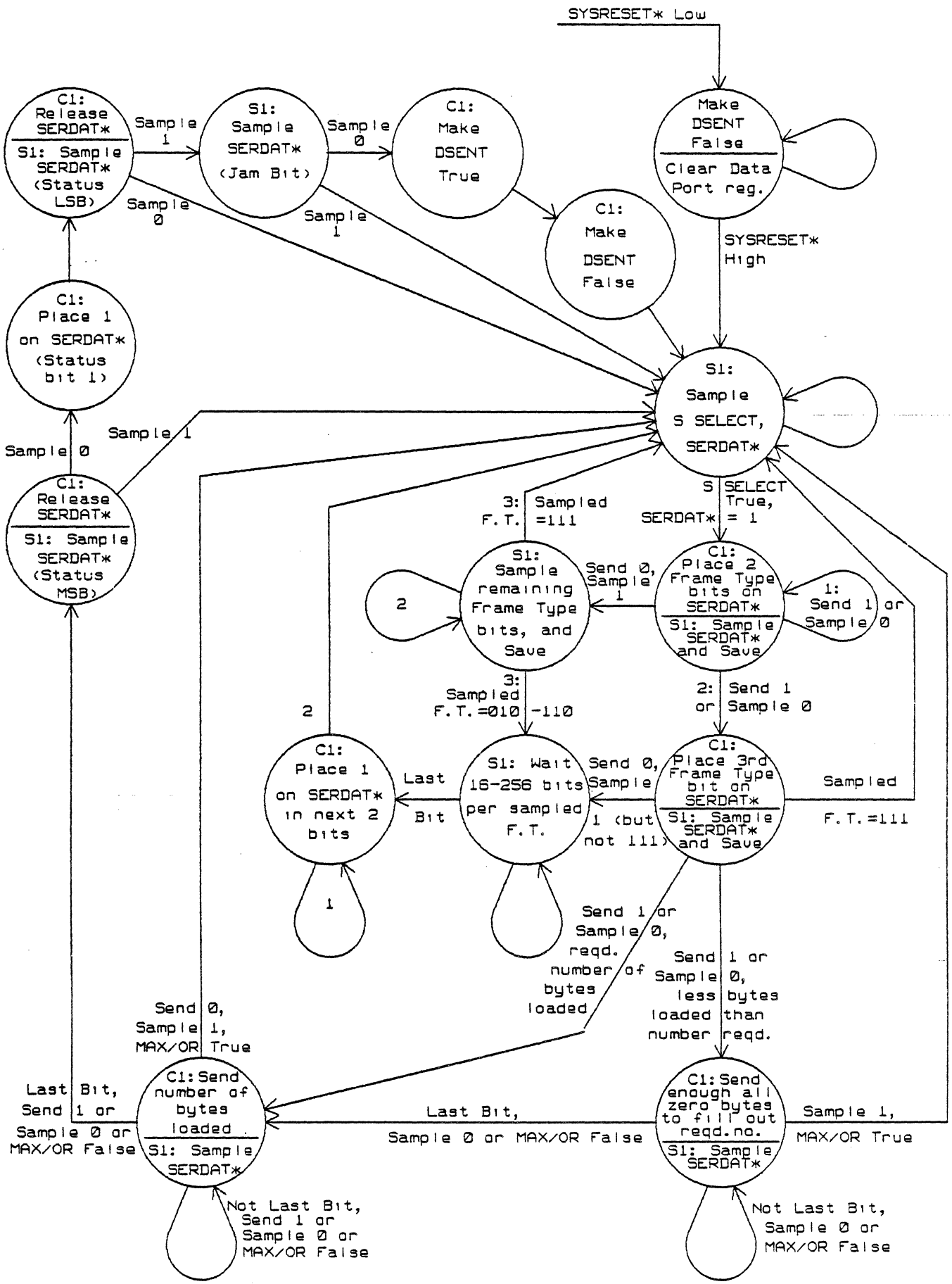

Figure 6-6. DATA SENDER State Diagram

2) This description assumes that the DATA SENDER "knows" the number of bytes provided by onboard logic, the last time new data was presented at Data Port. It further assumes that the DATA SENDER has converted this to a 3-bit Frame Type code as follows:

Number of Bytes	3-bit Frame Type
1	001
2	010
3-4	011
5-8	100
9-16	101
17-32	110

3) At the next C1 edge of SERCLK, the DATA SENDER sends the most significant bit of its 3-bit Frame Type on SERDAT*.

4) At the next S1 edge on SERCLK, it samples the state of SERDAT*. If it has sent a "one" or if it samples a "zero", the DATA SENDER proceeds to send the second and third bits of its code on subsequent C1 edges.

5) If on any of these three bits it samples a "one" (Low) on SERDAT* and it sent a "zero", the DATA SENDER stops sending any remaining bits of its 3-bit code on subsequent C1 edges, but it does continue to sample any remaining bits on S1 edges.

6) After the third S1 edge, if the 3 bits sampled are 111, the frame has been cancelled by a HEADER RECEIVER. In this case the DATA SENDER simply returns to idle state.

7) If the 3 bits sampled are not 111, but are not the same as the code the DATA SENDER set out to send, we have a "Sender/Sender Size Conflict". In this case the DATA SENDER uses the sampled code to determine the number of data bytes in the frame, counts off the corresponding number of bits on SERCLK, and then sends 110 on SERDAT* at the next three C1 edges (i.e. in the Frame Status field). (It does not sample nor arbitrate in the Status field.) It then returns to idle state.

8) If the 3 bits sampled bits match the DATA SENDER's Frame Type, it begins to send data on SERDAT*. If the number of bytes from onboard logic is less than the maximum number in the "Number of Bytes" column shown in Step 2, the DATA SENDER first sends the number of all-zero bytes necessary to fill out the maximum number of bytes (i.e. the bytes are sent "right justified").

After the all-zero bytes, or immediately if onboard logic provided the "right" number of bytes, it sends bytes from Data Port, starting with the most significant bit of the most significant (leftmost) byte, and ending with the least significant bit of the least significant (rightmost) byte.

9) As the DATA SENDER sends each bit of data on each C1 edge of SERCLK, it samples SERDAT* on the following S1 edge. If it samples a "one" after it sent a "zero", and if its MAX/OR input is True, then it immediately returns to idle state. If it samples a "zero", if it sent a "one", or if the MAX/OR input is False, it continues sending data bits.

10) After the DATA SENDER has sent and sampled the last data bit, it releases SERDAT* on the C1 edge of SERCLK and samples it at the next S1 edge. This is bit 2 of the Frame Status field. If it samples a one it returns to idle state.

11) (Bit 2 of the Frame Status is zero.) The DATA SENDER then drives a "one" on SERDAT* at the next C1 edge, to show that it survived any Data field arbitration. At the next C1 edge it releases SERDAT*, and then samples it at the next two S1 edges of SERCLK. These are bit 0 of the Frame Status field and the Jam Detect bit.

12) If bit 0 of the Frame Status is zero, or the Jam Detect bit is one, the DATA SENDER simply returns to idle state.

13) If these 2 bits are 10, the DATA SENDER makes its DSENT output True, to signal to its onboard logic that it has sent data successfully. On the following C1 edge it makes DSENT False again, and returns to idle state.

6.5 DATA RECEIVER Module

A DATA RECEIVER module must be paired with a HEADER RECEIVER. Its function is to take data sent by DATA SENDERS on the serial bus when it is signalled by its HEADER RECEIVER, and present the data to onboard logic.

6.5.1 DATA RECEIVER Signals

As shown in Figure 6-7, a DATA RECEIVER takes the SYSRESET* and SERCLK signals from the serial bus as inputs and the SERDAT* signal as a bidirectional I/O. It also has a number of inputs and outputs with onboard logic, as follows:

Data Port — A set of parallel output lines and associated control signals, through which data from the serial bus is presented to onboard logic.

> NOTE: Frames on the serial bus can include 1, 2, 4, 8, 16, or 32 bytes of data. Real DATA RECEIVER implementations may be limited to receiving less than 32 bytes. If a frame selects a DATA RECEIVER, and it contains more data than the DATA RECEIVER can accept, the module signals the problem on the serial bus, in the Frame Status field. This specification does not cover the details of how onboard logic reads received data from Data Port.

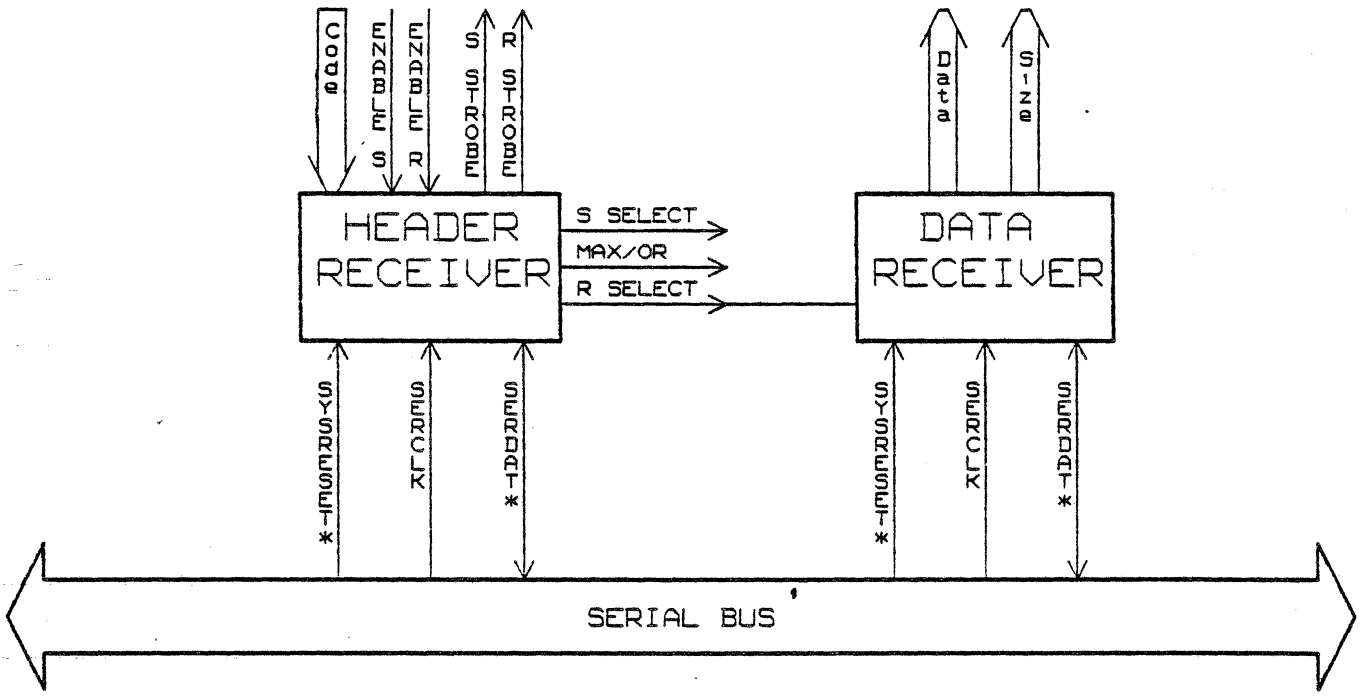

FIGURE 6-7. DATA RECEIVER with HEADER RECEIVER

Data Size Port — Three parallel outputs and associated control signals, through which the number of received bytes is presented to onboard logic. These outputs are encoded as described elsewhere in this specification for the Frame Type subframe.

NOTE: Data Port and Data Size Port may be implemented by dedicated signals, or by a common "bus interface".

R SELECT — An input from the paired HEADER RECEIVER, which selects the DATA RECEIVER to capture data from the serial bus.

NOTE: The R STROBE output of the paired HEADER RECEIVER signals onboard logic when data from the serial bus is available on Data Port.

6.5.2 DATA RECEIVER Initialization

Whenever SYSRESET* is Low, a DATA RECEIVER initializes itself as follows:

1) It releases SERDAT*, and ignores it as an input.

2) If Data Port and/or Data Size Port is implemented as a loadable register, it clears the register(s) to zero.

When SYSRESET* is released, the DATA RECEIVER enters idle state, described below.

6.5.3 Reading Data from the Data Port

In a system DATA RECEIVERS can be used in two ways. In one approach the DATA RECEIVER is always ready to receive data, and changes the data available on Data Port whenever it is selected by a frame on the serial bus. Onboard logic with such a DATA RECEIVER can use the data on Data Port "whenever it's needed". In this case, if onboard logic uses multiple "read cycles" to read out the data, and new data arrives on the serial bus while it is doing so, onboard logic may read out a mixture of old and new data. Real DATA RECEIVER designs must deal with this problem. One way to handle it is to signal onboard logic if the problem arises.

In other applications, each time new data arrives on the serial bus, onboard logic reads it from Data Port once and only once. In this scheme, when data arrives from the serial bus, onboard logic reads it out of the DATA RECEIVER as soon as possible. If another Data Transfer frame selects the DATA RECEIVER before the previous data has been read, the module "cancels" the frame.

This distinction is made by onboard logic and does not affect the DATA RECEIVER module described here. The following description of a DATA RECEIVER allows for either mode. In the first case, the ENABLE R input of the paired HEADER RECEIVER is permanently set to True, and its R STROBE output is not used by onboard logic. In the second case, onboard logic makes ENABLE R False when the HEADER RECEIVER pulses R STROBE, and makes ENABLE R True again after it has read out the data.

6.5.4 DATA RECEIVER Operation

The state diagram of a DATA RECEIVER is shown in Figure 6-8. Starting from idle state, the DATA RECEIVER proceeds as follows:

1) In idle state the DATA RECEIVER samples its R SELECT input and the SERDAT* line on every S1 transition of SERCLK, until it samples R SELECT True and SERDAT* "one" (Low).

2) On the next 3 S1 edges of SERCLK, the DATA RECEIVER samples SERDAT*. The sampled 3-bit code is the "Frame Type". The DATA RECEIVER interprets the Frame Type code as follows:

Frame Type	Interpretation
000	No DATA SENDER selected
001	1 byte of data in the frame
010	2 bytes " " " " "
011	4 " " " " "
100	8 " " " " "
101	16 " " " " "
110	32 " " " " "
111	Cancelled frame

3) If the Frame Type is 111, this is a Cancelled frame, and the DATA RECEIVER simply returns to idle state in this case.

4) If the Frame Type code is 000, no DATA SENDER in the system is selected to send data. In this case, the DATA RECEIVER sends 101 on SERDAT* on the next 3 C1 edges of SERCLK. On the next C1 edge it releases SERDAT* and returns to idle state.

5) If the Frame Type indicates more data than the DATA RECEIVER is designed to handle, it waits out the indicated number of bytes, and then sends 101 on SERDAT* on the next 3 C1 edges of SERCLK. On the next C1 edge it releases SERDAT* and returns to idle state.

6) If the sampled Frame Type code indicates a length the DATA RECEIVER can handle, it samples the indicated number of bytes from SERDAT*, on following S1 edges of SERCLK. It samples the data into a Data Holding register, starting with the most significant bit of the most significant (leftmost) byte, and ending with the least significant bit of the least significant (rightmost) byte.

> NOTE: If a DATA RECEIVER implementation is designed to provide data to onboard logic "once and only once" as described in 6.5.3, it can omit the Data Holding register and sample data directly into the Data register which is available via Data Port. If there are subsequent problems in the Status subframe, the R STROBE output of the paired HEADER RECEIVER is simply not pulsed to signal onboard logic of new data.

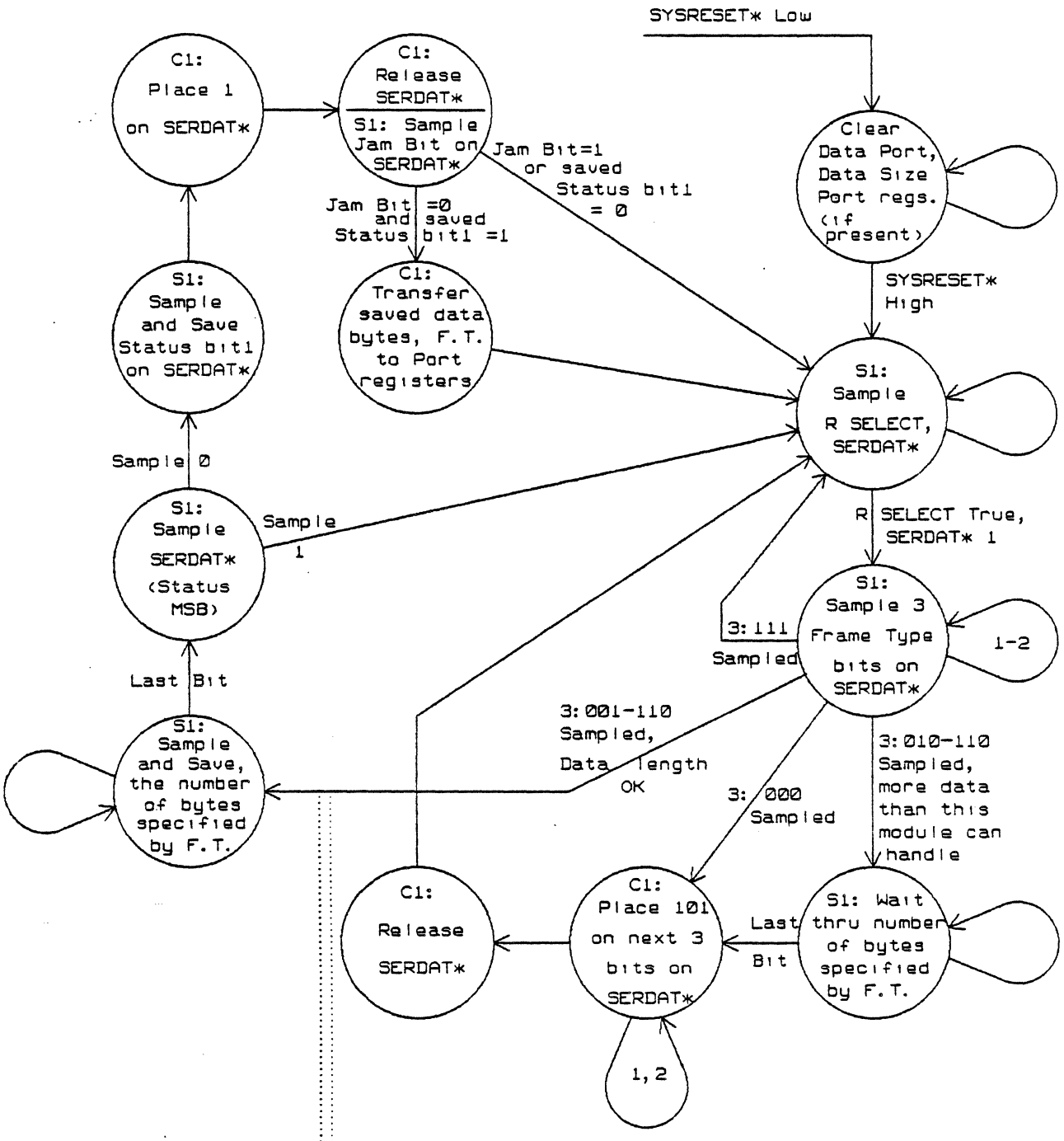

Figure 6-8. DATA RECEIVER State Diagram

7) After the DATA RECEIVER has sampled the last data bit, it samples SERDAT* on the next S1 edge of SERCLK. This is bit 2 of the Frame Status field. If it samples a one on SERDAT* it returns to idle state.

8) (Bit 2 of the Frame Status is zero.) On the next S1 edge of SERCLK the DATA RECEIVER again samples SERDAT* (bit 1 of the Frame Status field), and saves the sampled value.

9) On the next C1 edge of SERDAT*, the DATA RECEIVER drives a one on SERDAT* to show that it is present. This is bit 0 of the Frame Status field.

10) On the next C1 edge of SERCLK the DATA RECEIVER releases SERDAT*, and then samples it on the following S1 edge of SERCLK. This is the Jam Detect bit.

11) If bit 1 of the Frame Status is zero and/or the Jam Detect bit is one, there is a problem with the frame. In this case the DATA RECEIVER simply returns to idle state.

12) (Bit 1 of the Frame Status is one and the Jam Detect bit is zero.) The serial bus data transfer is successful. On the following C1 clock the DATA RECEIVER transfers the received data from the Data Holding register to the Data register which is available via Data Port, transfers the sampled Frame Type code to the register which is available via Data Size Port, and returns to idle state. The R STROBE output of the paired HEADER RECEIVER signals the arrival of new data to onboard logic.

6.6 FRAME MONITOR Module

The FRAME MONITOR "closes the loop" of serial bus modules we have been describing, by reporting the result of a frame transmission back to the onboard logic with the HEADER SENDER which initiated the frame. A FRAME MONITOR is normally paired with a HEADER SENDER, but could stand alone to monitor all serial bus traffic. (Since the SELECT output of a HEADER SENDER and the S and R SELECT outputs of a HEADER RECEIVER have the same timing, a FRAME MONITOR could be paired with a HEADER RECEIVER as well, but this combination has no obvious use.)

The FRAME MONITOR provides two other important functions. First, it "tracks" the transmission of every frame on the serial bus, and informs its paired HEADER SENDER when the bus is free for a new frame.

Second, it watches for Start bits at all times: if it sees a Start bit on SERDAT* while it is tracking a frame, an error caused by noise on the serial bus has caused one or more of the other FRAME MONITORS in the system to be "out of synchronization" with this one. In this case, it "jams" the serial bus by sending a string of 512 "ones" (Low) on SERDAT*. This invalidates the current frame: it is ignored by all serial bus modules. It also brings all FRAME MONITORS and HEADER SENDERS on the serial bus back into synchronization.

6.6.1 FRAME MONITOR Signals

As shown in Figure 6-1, a FRAME MONITOR takes the SYSRESET* and SERCLK signals from the serial bus as inputs and the SERDAT* signal as a bidirectional I/O. It also has a number of inputs and outputs with onboard logic, as follows:

Priority Port — Three parallel outputs and associated control signals, whereby the Priority field of serial bus frames is reported to onboard logic.

S,R Code Ports — Two sets of 10 parallel outputs and associated control signals, whereby the S and R Selection Codes of serial bus frames are reported to onboard logic.

Data Frame Port — An output indicating whether the frame being reported to onboard logic was a Data Transfer frame. This information is needed to interpret the information from the Status port.

Status Port — Three parallel outputs and associated control signals, whereby the Frame Status field of serial bus frames is reported to onboard logic.

> NOTE: The above ports may be dedicated outputs, or a common bus interface.

SENT — An output which indicates that a frame initiated by the paired HEADER SENDER has been completed, and that the results of the frame transmission are available on the above five ports. This output is made True only if the paired HEADER SENDER has signalled that it initiated the frame, by means of the SELECT signal. It is not made True for a Cancelled frame, nor for a "jammed" frame.

CANCELLED — An alternative output to SENT. It indicates that a frame initiated by the paired HEADER SENDER has been cancelled by a HEADER RECEIVER which is "not ready". Only the Priority and S and R Code ports are valid for a Cancelled frame. This output is not made True if the paired HEADER SENDER has not signalled on SELECT, nor for a "jammed" frame.

> NOTE: When a frame is cancelled, onboard logic can reduce the Priority value for subsequent retries (e.g. to 000), to allow other HEADER SENDERS to send frames on the serial bus. This strategy, plus increasing the Priority value when a HEADER SENDER loses serial bus arbitration (see 6.2.1), can ensure that each HEADER SENDER gets a fair share of access to the bus.

SELECT — An input from the paired HEADER SENDER, used to signal that the HEADER SENDER has won the arbitration for the serial bus and initiated the frame. This input enables the SENT and CANCELLED outputs for the frame. For a FRAME MONITOR which is not paired with a HEADER SENDER, this input should be permanently True.

FRAME IN PROGRESS	An output to a paired HEADER SENDER. When this output is False, the serial bus is free and the HEADER SENDER can initiate a frame.

6.6.2 FRAME MONITOR Initialization

Whenever SYSRESET* is Low, a FRAME MONITOR initializes itself as follows:

1) It makes its SENT and CANCELLED outputs False.

2) It makes its FRAME IN PROGRESS output True.

3) It releases SERDAT*, and ignores it as an input.

When SYSRESET* is released, the FRAME MONITOR makes FRAME IN PROGRESS False and enters idle state, described below.

6.6.3 FRAME MONITOR Operation

The state diagram of a FRAME MONITOR is shown in Figure 6-9. Starting from idle state, this module proceeds as follows:

1) It samples SERDAT* on both the S1 and S2 transitions in every SERCLK cycle, until it samples a "zero" on S1 and a "one" on S2. This indicates a start bit.

   NOTE: On the first C1 edge after returning from step 6 or 10 below, the FRAME MONITOR makes its SENT and CANCELLED outputs False.

2) After the C1 edge after sampling a start bit, it makes its FRAME IN PROGRESS output True. The timing requirement on this is "relaxed" in the sense that the signal need only meet setup to the next S2 edge at the paired HEADER SENDER.

3) Starting on the S1 edge after the S2 edge on which it found a start bit, it samples successive bits on SERDAT* on both the S1 and S2 edges, and disposes of them as follows:

   3 bits after start: sampled into Priority Port
   Next 10 bits: sampled into S Code Port
   Next 10 bits: sampled into R Code Port
   Next bit: discarded/ignored

4) For each of the bits sampled in step 3 and in subsequent steps until it returns to idle state, if the FRAME MONITOR samples SERDAT* High on S1 and Low on S2, it drives SERDAT* to "one" (Low) on 512 consecutive C1 edges of SERCLK. On the next C1 edge thereafter, it releases SERDAT* and returns to idle state.

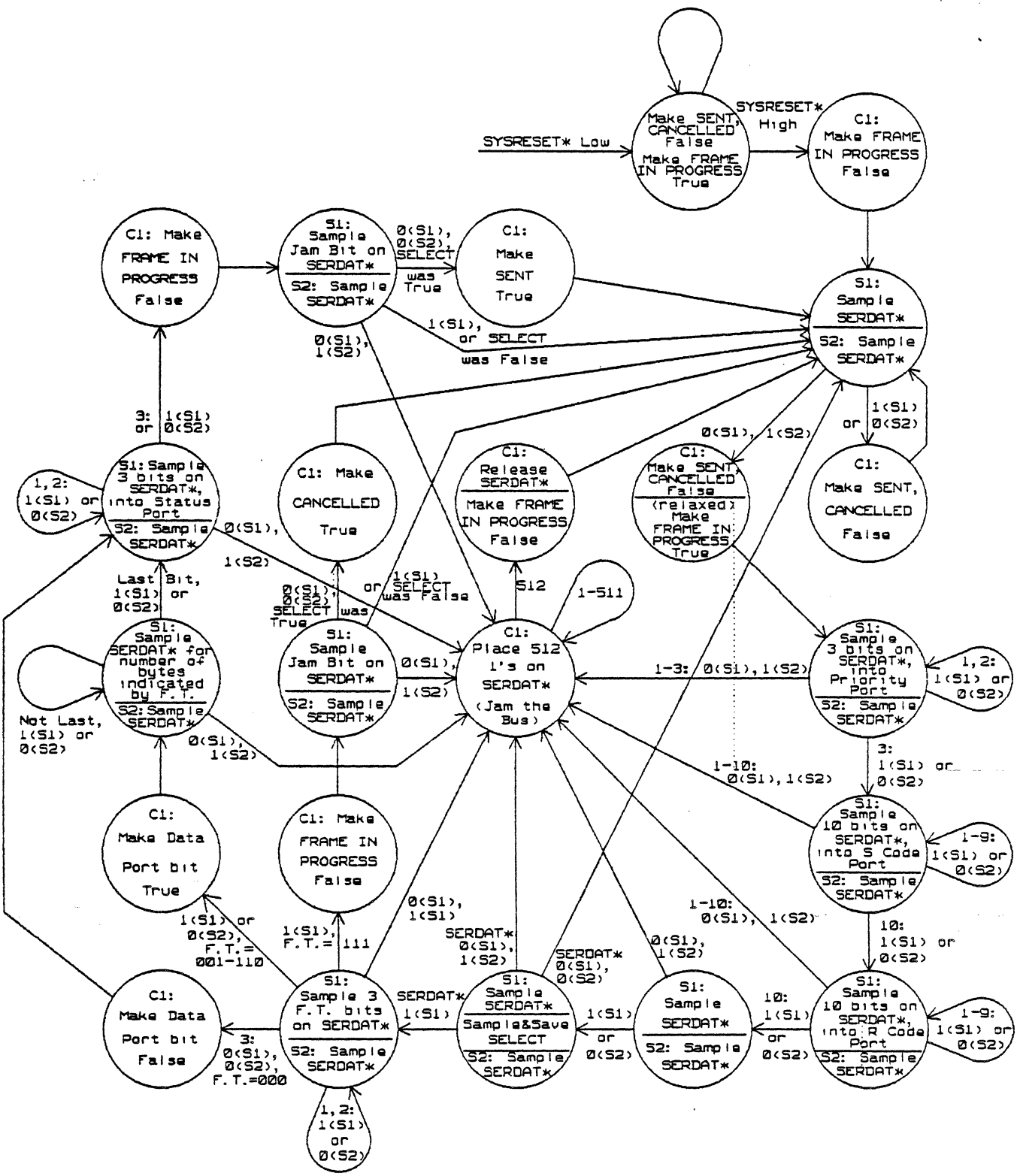

Figure 6-9. FRAME MONITOR State Diagram

5) The FRAME MONITOR then samples its SELECT input on the next S1 edge and retains the result for later. At the same time it samples the next bit on SERDAT* after those in step 3, the HSVAL bit. If it samples SERDAT* "zero" at S1 it makes FRAME IN PROGRESS False at the following C1 edge, and (unless it samples a "one" at S2) it returns to idle state.

6) (HSVAL was "one".) The FRAME MONITOR then samples the next 3 bits on SERDAT*, which are the Frame Type.

7) If the Frame Type is 111, it makes FRAME IN PROGRESS False on the following C1 edge. On the following S1 and S2 edges it samples SERDAT* (this is the Jam Bit). If the Jam Bit is "zero" and SELECT was sampled True in step 5), it makes CANCELLED True on the following C1 edge. Unless it sampled a start bit it then returns to idle state to sample the bit following the Jam Bit. (In idle state it will make CANCELLED False on the next C1 edge.)

8) If the Frame Type is 001-110, the FRAME MONITOR translates this code into the number of Data bits in the frame, as described elsewhere in this specification. On the next C1 edge it makes the register bit for Data Frame Port True, and then samples SERDAT* for that number of Data bits. In the Data bits the FRAME MONITOR samples for Start bits as described in 4), but otherwise discards the data. It then proceeds to step 10).

9) If the Frame Type is 000, on the next C1 edge the FRAME MONITOR makes the register bit for the Data Frame Port False.

10) On the next 3 S1 and S2 edges after the Data bits (if present, else after the Frame Type bits), the FRAME MONITOR samples these three bits into the Status port.

11) On the C1 edge after the last Status bit, the FRAME MONITOR makes FRAME IN PROGRESS False. On the following S1 and S2 edges it samples SERDAT* (this is the Jam Bit). If the Jam Bit is "zero" and SELECT was sampled True in step 5), it makes SENT True on the following C1 edge. Unless it sampled a start bit it then returns to idle state to sample the bit following the Jam Bit. (In idle state it will make SENT False on the next C1 edge.)

7. SERIAL BUS EXAMPLES

The basic serial bus modules defined in this specification provide a set of hardware building blocks that can be configured in many different ways for different purposes. This section shows some examples of serial bus module configurations, and how they can be applied to typical system needs.

7.1 EXAMPLE 1: A VIRTUAL SIGNAL LINE

The simplest path for communication within any system is a single conductor with a driver at one end and a receiver at the other. When an entire system is designed on a single printed circuit board, the designer is free to interconnect the logic with as many signal lines as are required.

Standardized backplane buses, however, provide much more limited options. It may not always be possible to fit all of the logic for a given function on a single board. When a design spans more than one board it usually requires signal lines between the boards. Sometimes this is accomplished by using some "reserved" lines on the backplane. Sometimes cables between the two boards provide the conductors. Both of these solutions create their own set of problems.

In this situation the hardware engineer is faced with a simply stated problem: how can he design hardware to do a specific job within the constraints of a particular board size and backplane bus? An analogy can be drawn between this problem and one which software designers face. Sometimes the amount of physical memory in a computer system is insufficient to accommodate all the required programs and data. To solve this memory limitation problem, "virtual memory" concepts were invented. Virtual memory puts the programmer into an "idealized" machine which has a very large amount of main storage. Such systems allow software designers to structure a program in a reliable and understandable way, without being forced to accommodate arbitrary memory sizing constraints. Although a virtual memory system may run slower than one fully populated with memory, the benefits are often viewed as being well worth the costs.

To bring this analogy back to the hardware world, the logic designer would like to be able to create a number of "virtual" signal lines between boards, without being constrained by arbitrary things like the number of reserved lines on a backplane, the number of wires in a cable, etc. Ideally this should be done without using any special dedicated lines on the backplane, since this can create incompatibilities with other board designs that use these lines for different purposes.

The serial bus allows the creation of such "virtual signal lines" between boards. Figure 7-1 shows how a TYPE 1 module group and a TYPE 2 module group do this. The SEND12 and SEND21 inputs to the HEADER SENDER at the top of the figure are driven by a single signal line named BINARY INPUT. Whenever this line goes True the SEND12 line goes True, causing the HEADER SENDER to send a "Flip-flop Set frame" with a selection code of 37 in the S field. This frame causes the HEADER RECEIVER in the lower part of the figure to pulse its S STROBE line True, setting the flip-flop.

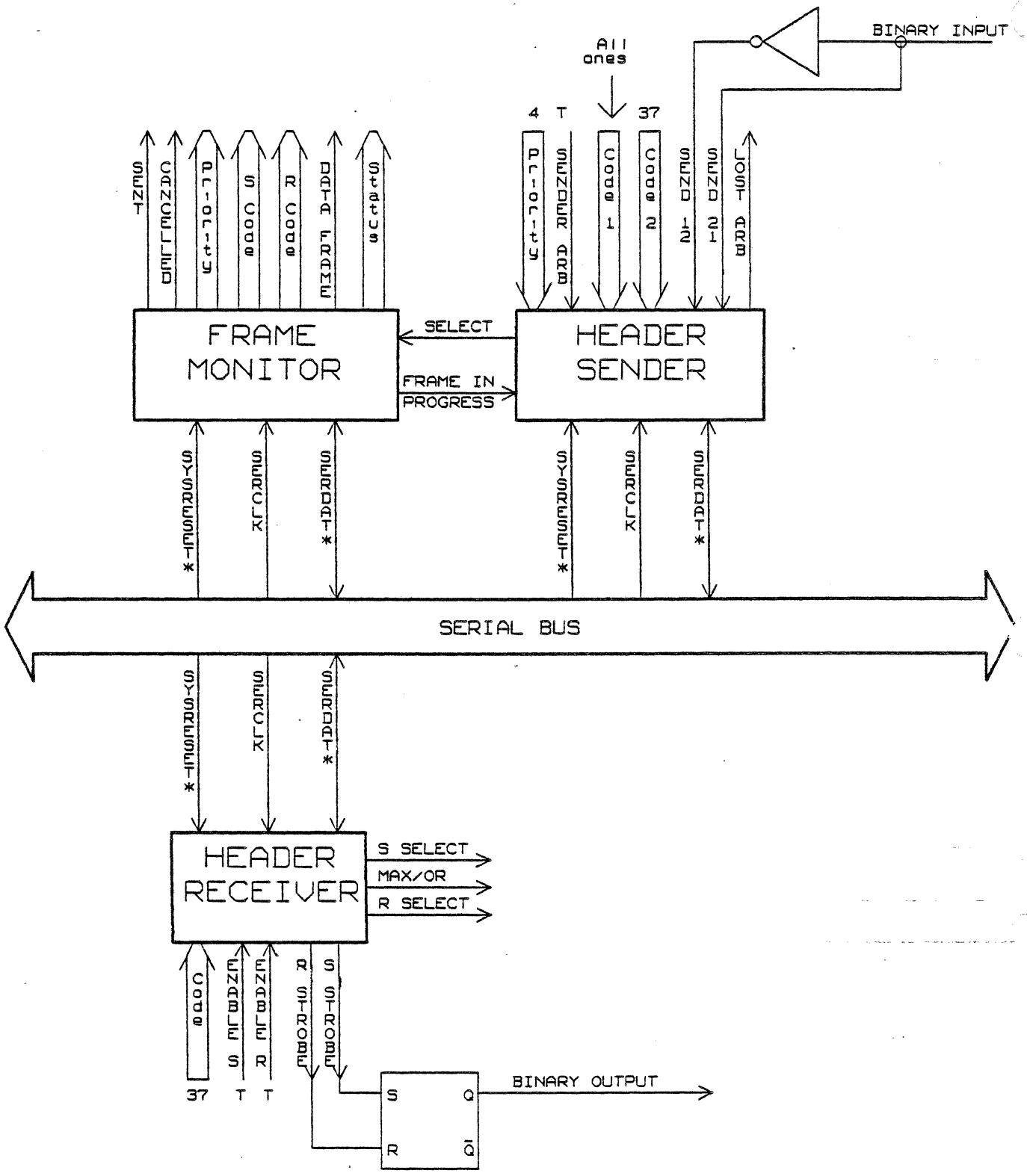

FIGURE 7-1. EXAMPLE 1: VIRTUAL SIGNAL LINE

When the BINARY INPUT signal at the top of the figure is returned to False, SEND21 goes True and the HEADER SENDER sends a "Flip-flop Reset frame". This frame causes the HEADER RECEIVER to pulse its R STROBE line True, resetting the flip-flop.

Thus, this arrangement of modules provides a "virtual signal line". Given that there are 1023 available selection codes on the serial bus ("all ones" is not used), we could theoretically create that many virtual signal lines. (Keep in mind that these "modules" are not individual integrated circuits. They are just the atomic units from which serial bus IC's are built. A single IC will usually provide a number of these modules.)

The "propagation delay" of serial bus virtual signals is greater than that of real lines, and is subject to variations with serial bus traffic. However, as in the case of virtual memory systems, the flexibility they offer will often outweigh these speed disadvantages.

6.2 EXAMPLE 2: A SEMAPHORE

As we discussed earlier, multiprocessing systems require the use of control structures such as "semaphores", to govern access to shared system resources. Figure 7-2 shows how serial bus modules can be configured to provide "intelligent semaphores".

The modules at the top of the diagram are on one board and those at the bottom are on another. The TYPE 2 module groups on the right side of the figure are the semaphores. Because these module groups are configured to respond to the same selection codes, they always stay synchronized. (If a Set frame sets one of them it also sets the other, etc.)

Notice that the output from each flip-flop is fed back to the S ENABLE input of its HEADER RECEIVER. If these flip-flops are set, their outputs make the S ENABLE inputs to the HEADER RECEIVERS False. In this state the HEADER RECEIVERS cancel any Set frames directed to them, accepting only Reset frames.

The TYPE 1 module groups at the left side of the diagram set and reset the semaphores. In each group, its SEND12 and SEND21 inputs are driven from a common signal named "RESOURCE REQUEST". In order to understand how these modules work together, let's go through two typical sequences.

When the on-board logic drives RESOURCE REQUEST True the flip-flops might be either set or reset. Let's take the case where they are set first.

If the two flip-flops are set, and the on-board logic of the top board drives RESOURCE REQUEST True, then the False signal from the "Q bar" output of the flip-flop prevents the HEADER RECEIVER'S SEND21 input from going True. This prevents the HEADER SENDER from sending an unnecessary Set frame over the serial bus. (It would just be rejected by the HEADER RECEIVERS anyway.)

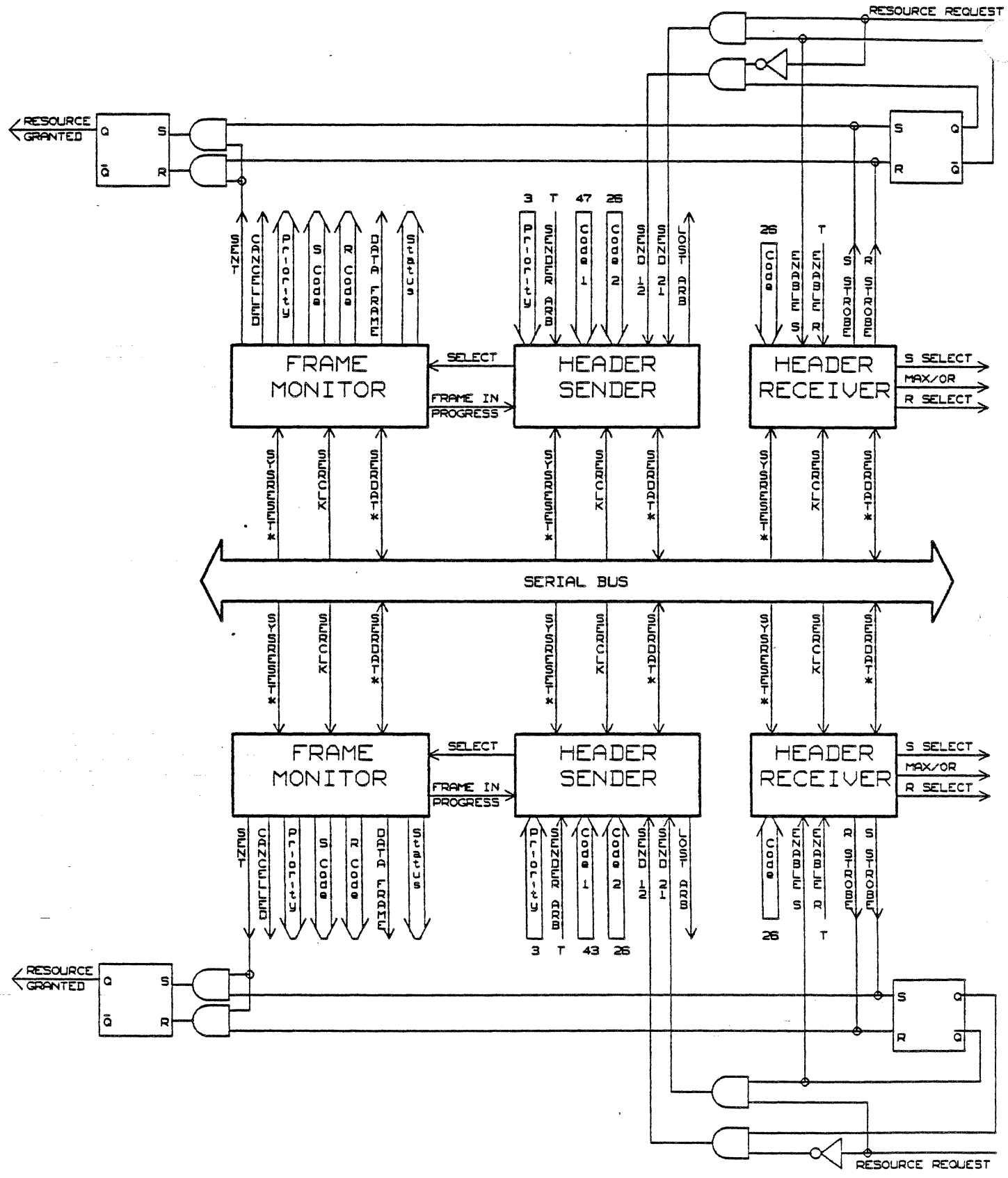

FIGURE 7-2. EXAMPLE 2: A SEMAPHORE

Suppose that while the top board is waiting to send a Set frame, the on-board logic of the bottom board drives its RESOURCE REQUEST line True also.

When the HEADER SENDER that originally set the flip-flops (not shown) finally sends a Reset frame, the "Q bar" outputs of both flip-flops enable the SEND21 inputs of the upper and lower HEADER SENDERS simultaneously. As a result, they both begin to send Semaphore Set frames. Since their semaphore select codes are the same in the S field, both are still sending at the end of that field. However, since the HEADER SENDER at the bottom of the figure has the smaller requester code in its R field, it retires from the bus. This leaves the top HEADER SENDER to finish its frame, setting the flip-flops on both boards.

Since the flip-flops on BOTH boards are set, this fact doesn't tell the on-board logic which board has been granted the resource. The FRAME MONITOR on the left side of each board samples all frames on the bus, and asserts SENT if its HEADER SENDER wins the arbitration and signals this fact on the SELECT line. In the current case the top FRAME MONITOR pulses its SENT output True.

The diagram shows how the SENT pulse is used to control a flip-flop which generates a RESOURCE GRANTED signal to the on-board logic. (The setting of this flip-flop may be used to generate an interrupt to an onboard processor.) If there are several semaphores in the system, decoding on the S and R code outputs of the FRAME MONITOR can be used to route the SENT pulse to one of a number of such flip-flops.

When the onboard processor on the upper board finishes using the corresponding resource, it makes the RESOURCE REQUEST signal False. This makes the SEND12 input of the upper HEADER SENDER True, and it sends a Reset frame. This clears both of the semaphores plus the upper RESOURCE GRANTED flip-flop. The HEADER SENDER on the lower board is then enabled to send a Semaphore Set frame. When this occurs, the lower RESOURCE GRANTED signal signals its onboard processor that it now controls the resource.

7.3 EXAMPLE 3: A VIRTUAL BUS

Most multiprocessing architectures today are the result of trade-offs based on the high cost and physical size of backplane interface logic. Thus most multiprocessing systems today are based on a single backplane bus instead of several.

But, if we look at the logical structure of these systems we find that each processor may need to communicate with every other processor. One could envision a system that provides for this kind of communication with a complex structure of point-to-point communication paths, but given the cost of the hardware, it is usually impractical to build such a structure of buses.

Clearly, the hardware structure of a typical single-backplane system is very different from the type of message traffic needed in a multiprocessor configuration. This difference in structure can sometimes be hidden by the software which handles the message traffic, but it can result in awkward constraints which the user must live with, perhaps without ever really understanding why.

In section 7.1 we talked about how the serial bus could be used to provide a very large number of virtual signal lines. These lines could be configured to connect any board to any other. This logical structure is much more like the one we've just discussed. While limited, these virtual signal lines provide the kind of point-to-point paths needed in multiprocessing systems.

But the serial bus can do more than just provide virtual signal lines between boards. It can also provide "virtual buses". Figure 7-3 shows a module configuration that permits this. The TYPE 3 and TYPE 1 module groups at the top of the figure are on one board and the TYPE 4 module group at the bottom is on another board.

The TYPE 1 module group contains a HEADER SENDER which directs a Data Transfer frame from the top board to the bottom whenever its SEND12 input is driven True. It does this by sending a Header subframe that selects the DATA SENDER on the top board and the DATA RECEIVER on the bottom board. If we look at the signals from the on-board logic on the top board and the the output signals that go to the on-board logic of the bottom board, in each case we see a parallel bus plus a "strobe" line. This "virtual bus" has a much slower "propagation time" than a real bus and its delay varies with the level of bus traffic, but the fact that the serial bus can provide up to 1023 "virtual buses" makes it very useful.

Of course, once we can simulate a simple bus and a strobe we can create more elaborate structures. For example, we could have one such virtual bus going from board A to board B, and a second virtual bus in the other direction. This module configuration can be used for "stimulus-and-response" operations. The simplest example of such an operation is board A reading a memory location on board B.

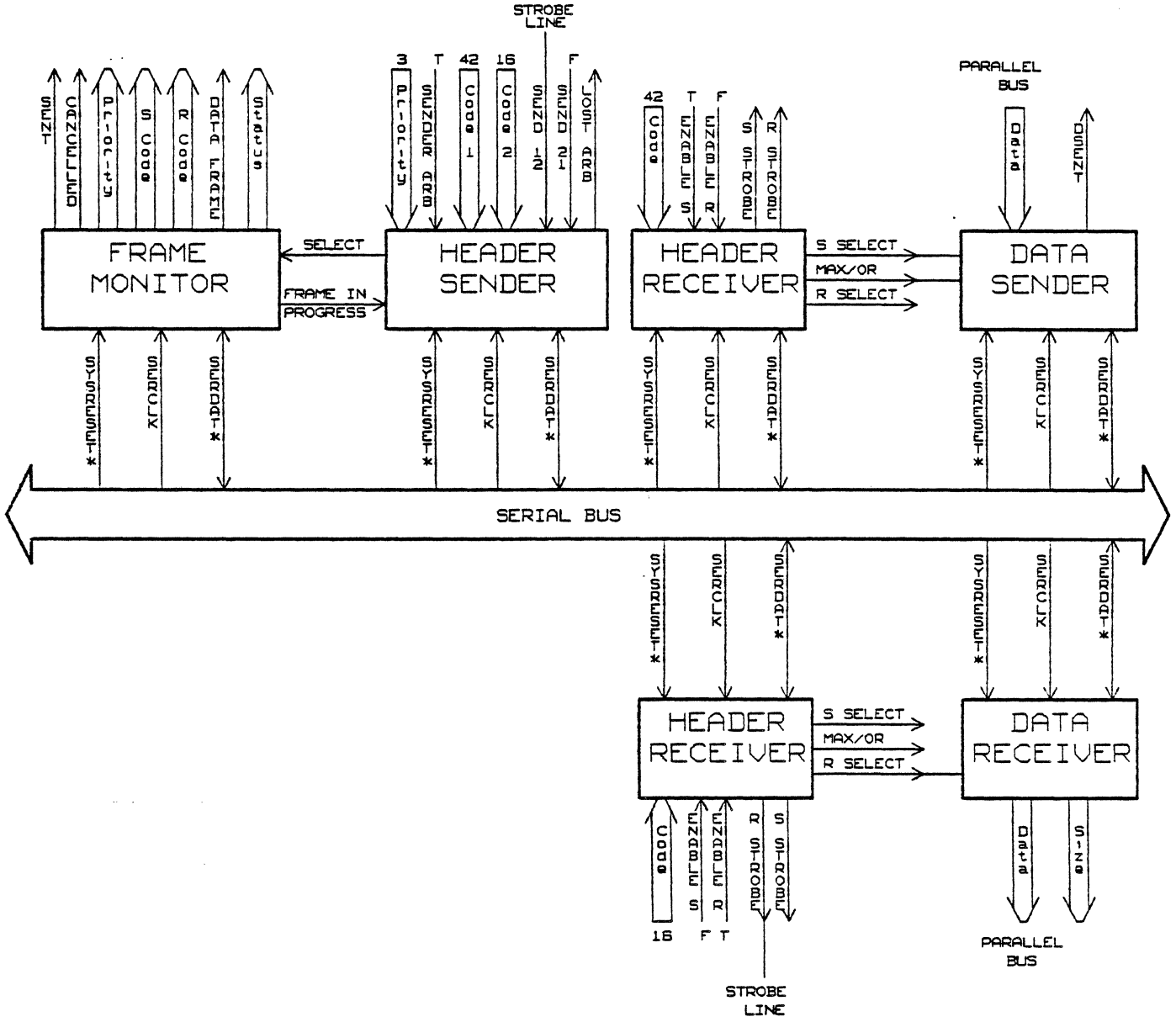

FIGURE 7-3. EXAMPLE 3: A VIRTUAL BUS