XEROX
BUSINESS SYSTEMS
Systems Development Department

To:	Bob Metcalfe	Date:	September 14, 1978
From:	Roy Ogus and Victor Schwartz	Org:	SDD/SD System Architecture
Subject:	**D0 RS-232-C microcode status**	Filed:	[Iris]<Ogus>memo>RS232Status.memo

This memo discusses the current state of D0 RS-232-C microcode. It describes what has been done up until now for the Oak project and what difficulties have been encountered. The additional functions still required to realize the full RS-232-C controller for Pilot/Star1 are mentioned, and some recommendations are made for their implementation.

What has been done for Oak

Background. As described in a previous memo ([Iris]<Ogus>memo>RS232.memo), the only hardware existing in the D0 for RS-232-C transmission is a set of hardware timer modes which provide a mechanism for timing the bits received or transmitted through the RS-232-C port. The microcode must do all the serialization and deserialization of data, as well as control the other RS-232-C signals directly. It is thus intended that the RS-232-C microcode be driven by the timer microcode task (which is also servicing other timers such as the memory refresh timer, etc.). Hardware wakeups are provided only at timer expirations. This means that the microcode driven by the timers cannot "TASK" (i.e offer the machine to other tasks) before it has completed the servicing of the particular timer. If it did, there would not be a subsequent wakeup to allow it to resume service. This limits the amount of processing per timer service to about 12 microinstructions to prevent hogging of the machine bandwidth. It was thus realized that it would not be feasible for the timer task to handle the large amount of work specified for the RS-232-C microcode in the *Pilot Design Specification.* A simpler byte interface between the software and the microcode was proposed, and an asynchronous mode of operation was planned as the first implementation of such an interface.

Description of asynchronous mode. A detailed description of this interface can be found in another memo ([Iris]<Ogus>memo>RS232Int.memo), which is being updated continually to track the state of the microcode. The microcode consists of three parts which all run under under the D0 timer task. They are the low-frequency *poller*, the *receiver*, and the *transmitter*. The poller runs at a frequency of about 100 polls per second. It provides the interface between the software driver and the microcode, handling the RS-232-C control signals transmitted to or received from the port, as well as controlling the receiver and transmitter. The receiver and transmitter carry out the (de)serialization and timing of bits to and from the RS-232-C port, and perform the (de)encapsulation of the characters (start, stop, and parity bits). An Interim Controller Status Block (ICSB) is maintained by the software for communication between the driver and microcode. Associated with the ICSB are two ring buffers which are used for the buffering of characters between the software and microcode.

Statistics. The above three pieces of microcode require about 200 microinstructions (excluding the SendBreak and BreakDetect functions, which are not yet implemented). 20 R-registers are needed, and the ICSB occupies 12 words. Currently, two 128-word ring buffers are used. The worst case timer service is 18 microinstructions for the transmitter and receiver, including the timer dispatch overhead; a typical data bit service will be 12 to 14 microinstructions, which amounts to about 2.4% of the processor cycles at 9600 bps (assuming a 95 ns cycle time). This does not include conditional branches (which extend an instruction time if the condition is true), and aborted instructions due to memory conflicts. The actual usage of the bandwidth will be somewhat higher.

Status. The microcode for the asynchronous mode has been completed except for the implementation of the SendBreak and BreakDetect functions. A test (micro)program has been written to test the code, which provides the same interface to the RS-232-C code as the Mesa software will. Using the test program, the RS-232-C code runs correctly for line speeds that use only a single D0 timer ($>$ 1200 bps), and has been tested in loopback mode through a loopback connector on the D0. Line speeds that use the double timer mode ($\leq$ 1200 bps) do not function correctly due to a bug in the D0 timer hardware. This bug has been fixed in one D0 (EM 009) and the test program runs correctly on this machine for all line speeds (110 bps through 9600 bps). The RS-232-C microcode has been integrated into the total Mesa microcode, and the RS-232-C Mesa software has correctly executed on the D0 in loopback mode for line speeds in the range 2400 bps through 9600 bps. Its as expected, though not yet verified, that the software will execute correctly on EM 009 for *all* line speeds. Current work involves making the necessary microcode changes to improve the overall software-microcode performance.

Problems encountered. Most of the difficulties experienced were related to the handling of the hardware timers. One problem concerned the computation of the values to be loaded into the timers. The word to be loaded into a timer contains three fields, viz. the timer slot, the timer state, and the data value. The first two can be fixed at assembly time for the various timers, but the data value is a function of the RS-232-C line speed, which is a variable, its value being determined at runtime. Since it takes at least 4 instructions to compute a timer value, and since we potentially need 6 timer values, it was decided to precompute all the possible timer words and have the software pass the appropriate precomputed constant(s) to the microcode through the ICSB. This results in 6 words in the ICSB, and 6 R-registers being needed for timer values, since we need input bit-time values, input half-bit-time values, and an output bit-time value. Slow line speeds require a double timer each, thus the 6 values. The alternative of passing a single value of the line speed and computing the timer values dynamically is not feasible due to the amount of computation needed.

A second problem was the fact that some line speeds require the loading of a double timer to time a bit (speeds $\leq$ 1200 bps), whereas others ($>$ 1200 bps) need only a single timer. This posed a problem as for the microcode, since *different* state and slot (and data) values are loaded into the timers depending on whether the timer is to be double or single. In addition, two slots need be loaded for a double timer, and there is a requirement that at least 4 instructions intervene between successive timer slot loads. The solution chosen to handle all the above constraints was to have the software set or clear a "slowLine" bit in the ICSB, indicating to the microcode whether a single timer is to be used. This increased the number of microinstructions since the bit has to be tested each time a timer is to be loaded. In addition, there were cases where the only way to insure that there were 4 intermediate instructions was to insert NOPs, again adding to the number of microinstructions.

The RS-232-C hardware timer modes on the D0 were largely untested, and great deal of time was spent tracking down apparent microcode problems, only to finally conclude that the timers themselves had bugs. This indeed was the case, and after the hardware problems were fixed, the remaining bugs in the microcode disappeared.

Considerable time and effort was spent in reducing both the number of instructions required to service a timer wakeup, as well as the number of R-registers needed. In many cases it was found to be difficult to keep the timer service within the 12 instruction constraint, given the amount of work that had to be done. The entire state of the receiver and transmitter has to be stored in R-registers, and a great deal of register sharing was done to save R-registers. Nevertheless, 20 R-registers are still needed - 6 of which are used to store the timer constants.

To summarize, a significant amount of D0 resources is required to implement the receiver and transmitter bit handlers. About 70% of the 200 instructions are needed for these functions; 20 R-registers, and a 12-word ICSB in a dedicated portion of main memory (e.g. the I/O Page), are required as well. If it becomes apparent that 128-word ring buffers are too small to prevent input overrun or excessive wakeups of the Mesa driver, then larger buffers will have to be used, further increasing the main memory and R-register usage. There are additional functions which are either still needed in the microcode, or would inprove performance if implemented in microcode. The SendBreak and BreakDetect functions are examples of the former category. Other reasonable functions for microcode implementation include CRC computations, as well as naked notifies at appropriate times. Implementing these functions would, of course, increase the amount of processing required for timer services.

What remains to be done for Pilot/Starl

The microcode functions still required for the implementation of Pilot RS-232-C controllers can be divided into those functions *essential* for controller realization, and those which would be more desirably implemented in microcode than software from a performance point of view.

To complete the functions for the *asynchronous* mode of operation, the SendBreak and BreakDetect functions need to be implemented. As mentioned above, incorporating the CRC computation and the naked notifies in the microcode would improve the overall performance.

In order to realize the Pilot RS-232-C controller, two *synchronous* modes of operation need to be implemented in the microcode. These modes are the synchronous *byte-oriented* mode, and the synchronous *bit-oriented* mode (e.g. HDLC). The former can be implemented as a byte-interface similar to the asynchronous interface. It would also be enhanced by the addition of the CRC computation and naked notify capability. The bit-oriented synchronous mode could also be realized as byte-interface, but would be greatly improved by incorporating a frame interface. Since the bit-oriented mode requires zero-bit insertion to be performed, and cannot allow intra-frame idling, a frame interface might be the only way to realize the full 9600 bps speed.

To realize the full RS-232-C controller in microcode as specified in the *Pilot Design Specification*, the function of IOCB chaining should be implemented. This function is presently performed by software, but its incorporation into the microcode (if possible) would improve system performance.

Approaches to completing the RS-232-C controller

In order to realize the full RS-232-C controller, three basic approaches can be followed. Each approach will implement a different amount of the RS-232-C controller functions in microcode.

a) Timer task microcode only.

The first approach is to follow the current strategy and attempt to implement as much as possible of the remaining functions using microcode which runs under the timer task only. As mentioned above, the current implementation pushes the amount of processing per timer service to the limit. It is clear that not much more processing could be added using this approach.

The SendBreak function for the asynchronous mode can be implemented by a simple extension to the current microcode; the BreakDetect can similarly be realized.

It appears to be feasible to implement a byte-oriented synchronous mode interface in a similar fashion to the asynchronous mode. This would provide a byte interface to the software, and would have to incorporate the functions of character synchronization, and SYN character generation during output idling. It does not seem feasible to incorporate the CRC computations in the timer-task-only approach, since this would involve too much extra processing, and requires knowledge of the frame. Naked notifies could possibly be incorporated into the poller function. The synchronous modes do not need double timers, since the appropriate timer modes generate wakeups based on modem clock transitions rather than expiration of a time period.

It may be possible to realize the bit-oriented synchronous mode as a byte-interface to the software as well. This mode requires the added function of zero-bit-insertion/stripping and, in addition, cannot tolerate intra-frame idling. It might thus prove difficult to attain the full speed of 9600 bps using a byte interface. A frame interface is really needed. Such an interface has at least three advantages: first, the delineation of the logical unit of transfer between the software and microcode, eliminating Mesa polling; second, the simplification of CRC computation by microcode; and third, a solution to the intra-frame idling problem.

One technique to increase the amount of processing per timer service is to set timer wakeups to occur twice (or *n* times) every bit. This would allow an extra processing cycle(s) to occur in between each bit. This scheme has the disadvantage of extra overhead due to two (or *n*) timer loads and wakeups per bit (4 or 2*n*, if slow speed, asynchronous), and in the synchronous mode the interbit timer loads will require different timer values from the normal timer loads, necessitating extra constants to be stored.

b) Timer task plus other task microcode.

In order to put substantially more processing into the microcode, execution by a task other than the timer task will be needed. Using some other task has the problem that no hardware wakeups exist to wake up the task after it has TASKed. Task 0, which is used to run the Mesa emulator, always has its wakeup request asserted. Thus, task 0 could be notified by the timer task at appropriate times to continue with the RS-232-C processing. The notified microcode would first save the state of the emulator (the TPC and any emulator registers used), and later restore this state when the RS-232-C processing has completed. This would be a way to use task 0 "behind the emulator's back" to obtain more processing for the RS-232-C functions. This scheme suffers from several disadvantages. The communication between the timer task and task 0 is cumbersome. To communicate through R-registers the stack would have to be used, otherwise the communication could take place through main memory, again increasing the amount of fixed memory needed. The number of R-registers available for RS-232-C processing at the task 0 level is minimal since most of them are used by the emulator. In addition, there are several instructions of overhead associated with the notify to task 0 which count as part of the timer task wakeup processing.

Another scheme to alleviate the above R-register shortage is to use a third task in conjunction with task 0 as described above. Task 0 would continually notify the third task, which could carry out the needed RS-232-C processing and continually update its TPC appropriately. It would rely on task 0 for its "wakeups". This scheme is even more complicated, and suffers from the same disadvantages as using task 0 alone, except that more R-registers are available.

c) RS-232-C controller hardware.

A third approach would be to add hardware to the D0 which could handle the bit timing to and from the RS-232-C port and present a byte interface to the microcode. LSI chips exist on the market for these functions; in particular the Zilog Z80-SIO chip handles asynchronous, byte-synchronous, as well as bit-synchronous protocols. CRC computations are performed, zero insertion provided, and two full-duplex channels are available, as well as numerous other functions. Hardware wakeups could be provided for the D0, so that the RS-232-C microcode could easily implement the full Pilot RS-232-C controller functions. It is understood that there is space on the floppy-disk controller board, and the amount of hardware required should be reasonably small.

The hardware approach would have several advantages. The micro-instructions which are now needed for bit timing would be eliminated, thus freeing up space in the control store for other RS-232-C functions. The hardware wakeups would remove the constraints imposed on the RS-232-C microcode by the timer task. It is expected that the overall performance could be improved. The full Pilot RS-232-C controller could be implemented in microcode in a straightforward manner.

Discussion of approaches

An important parameter which should be used to evaluate the three approaches is performance. However, it is still too early to get an accurate feel for the performance of the current microcode implementation, and it might be too late to change anything once enough performance data is collected.

If approach (a) is chosen then what could realistically be implemented is a byte-interface for each mode, without CRC computations, and possibly with naked notifies. For the high speed bit-synchronous mode, large ring buffers would be needed to prevent output data underrun, since intra-frame idling does not exist in the protocol. The rest of the Pilot RS-232-C controller functions would have to be implemented in software.

If approach (b) is followed, then more controller functions could be moved to the microcode, perhaps even to approach the full Pilot controller. This approach will, however, require a great deal of microcode (perhaps 2 pages per mode) and R-registers, as well as being very complicated. The microcode development time would be significant. Of approaches (a) and (b), it would seem that (a) should be chosen and a byte interface implemented, unless, as a result thereof, it becomes impossible to achieve the 9600 bps data rate. If no extra hardware is provided in the D0 then approach (b) will be the only viable one which could implement more than a byte-interface between the microcode and software.

The hardware approach would easily allow the full Pilot RS-232-C controller to be implemented with a reasonable amount of microcode.

Recommendation

Approach (b) should be avoided, since it presents some very difficult interfaces which will almost certainly cause development and maintenance problems.

Approach (a) may be workable, although it suffers in three major ways: first, the tricks which will be necessary to get the required features into the microcode will delay and complicate a successful implementation; second, the extra burden placed on the Mesa driver (e.g. for various polling functions, and for CRC computation) may result in excessive use of processor bandwidth, and/or failure to perform at the advertised speed of 9600 bps; and third, future expansion to handle post-Star1 functions will be increasingly difficult to accommodate, and will, in all likelihood, lead us to approach (b) with the resulting additional complexity.

In summary, based on the experience gained by progressing to the current state of the microcode, it is clear that approach (a) will result in a relatively tedious and intricate development, and perhaps may not perform adequately when complete. *For these reasons, we recommend approach (c).* Future communication controller implementation can re-assess the appropriate hardware/microcode/software tradeoffs. But for this implementation, the hardware cost involved appears to be worth the risk-reduction it will buy for the microcode and software development.

cc: Bob Belleville, Don Charnley, Ron Crane, Bill Danielson, Bob Garner, Carol Hankins, Pitts Jarvis, Rich Johnsson, Howard Kakita, Bill Kennedy, Robert Kierr, Brian Rosen, Dick Snow, Chuck Thacker, Jim White