```
+---------------+
! d i g i t a l !   I N T E R O F F I C E   M E M O R A N D U M
+---------------+
```

TO: Dolphin CPU List

cc: Peter Hurley
Arnold Miller

DATE: 28 Mar 78
FROM: Dan Murphy
DEPT: LCG S.E.
EXT: 6356
LOC: MR1-2/E37
FILE: CLKMEM

SUBJ: CLOCKS FOR DOLPHIN

The following is a discussion of some of the aspects of clock handling in TOPS20 presented in the hope of stimulating some ideas about how to design hardware to do it more efficiently.

BASIC CLOCK UNITS

The monitor keeps clocks, including elapsed times, runtimes, times of day, etc., in two basic units-- 1 millisecond and 10 microseconds. The 1 ms. clock is used for a wide variety of functions, including all accumulated job runtimes. The 10 microsecond clock is used where greater precision is necessary, particularly for measuring incremental job runtimes.

In principle, the high resolution clock would be sufficient. In practice, there are several reasons to retain the 1 ms clock:

1. A 36-bit word can hold a maximum of about 94 hours in 10 usec units; hence a double-precision integer must be used for accumulated times. With 1 ms units, over 1 year can be represented in a single word.

2. It is less convenient to read the high precision clock accurately since an IO instruction is necessary to read it followed by a full divide to convert to the desired units.

3. Most times available to user programs via monitor calls are specified in 1 ms units, and again, conversion to higher precision units is not generally worthwhile.

READING CLOCKS

The current value of the high precision clock is always obtained from the hardware. The RDTIME instruction gives microsecond units in a double word with a binary point following bit 23 of the low-order word. A divide is then used to convert this to the desired quantity, the divisor being ^D10B23 for 10 usec units, and ^D1000B23 for 1 ms. units.

(The KS10 uses a different format doubleword for the RDTIME instruction. In this format, the low order bit of the doubleword is counted at exactly the rate of the clock crystal. Originally, this crystal was specified as 4.096 MHz so that the overflow of a 12-bit counter could be used to generate 1 ms interrupts. This created a problem for TOPS20 however, since there is no divisor which will convert that to 10 usec units. At TOPS20 request, it was agreed that the crystal would be changed to 4.1 MHz for production machines. There has been some difficulty getting this decision implemented however, and at present the loadtest machine's clock appears to lose about 3.5 seconds per hour.)

The millisecond clock is generally read directly from a core location. Originally (on the KA10 and KI10), this cell was maintained by being AOS'd on each 1 millisecond interrupt. The 1 ms clock must be perfectly synchronized with the high precision clock however, so that approach does not work on the KL10. Until recently, the 1 ms clock was maintained on the KL10 by dividing down the master clock and storing it on each 1 ms interrupt. This, however, costs noticable overhead, particularly on the KS10. Therefore, as of release 3, the 1 ms clock is not updated periodically but rather only on explicit request. The most important routines which read this clock were modified to request an update, and others were left to just get the last updated value. Since the clock is updated on every process context switch, we assume the value is sufficiently accurate. Even this approach is not entirely satisfactory. It may well be that several clock update requests occur within one millisecond thus causing more overhead in the short run than formerly. Also, some clock values are not truly updated and this may be a source of bugs that we haven't discovered. Putting in more update requests would further increase the overhead.

This suggests that the most useful thing would be to have the desired units available directly from the hardware and readable efficiently. As described, two time base clocks would be sufficient for TOPS20, one reporting 1 millisecond units, and the other reporting 10 usec units. There might well also be a third clock reporting the highest precision units available. A clock frequency would be chosen of which 10 usec and 1 ms are integral multiples, and a count-down register would be provided for each clock to cuase the clock to count at the specified rate. Once initialized, the clocks would be synchronized so that UCLOCK/100=MCLOCK always.

RUNTIME MEASUREMENTS

Another facet of clock handling that could possibly be improved is incremental runtime measurement. At present, it takes probably 20 usec to compute the incremental runtime of the current process, i.e. about twice the basic clock rate. The following is the actual code used:

```
        GETHRT: NOSKED                  ;PREVENT SCHEDULING DURING THIS
                JSP T4,MTIME            ;READ CLOCK FROM HARDWARE
                SUB T1,FKT0             ;COMPUTE DIFFERENCE
                CAIGE T1,0              ;HANDLE POSSIBLE OVERFLOW
                ADD T1,BASOVV
                ADD T1,RUNT2            ;ADD FRACTIONAL MS BASE
                MOVE T2,FKRT            ;GET UNITS MS BASE
                IMULI T2,^D100          ;CONVERT TO HP UNITS
                ADD T1,T2               ;SUM OF ACCUMULATED AND INCREMENTAL RT
                OKSKED                  ;ALLOW SCHEDULING
                RETURN
```

The above is the sequence necessary to determine the current runtime of a process in high precision units. The equivalent routine for 1 ms units is similar since the basic clock is kept in high precision units. NOSKED and OKSKED are subroutine calls which prevent rescheduling out of the sequence, since inconsistent results would occur if the scheduler updated the base values during the sequence. The computation being performed is:
        (ACCRT + (STTIM - CURTIM)),
where ACCRT is the process accumulated runtime when last started, STTIM is the time of day when the process was started, and CURTIM is the current time of day. These variables all require more than 36 bits of precision, so the computations are done with millisecond and fractional millisecond units.

This entire sequence could be eliminated if there were a clock maintained by the hardware which could be loaded with the process runtime at the start of process execution and which would count at the desired rate (10 usec) while the process ran. It should also be possible to turn this clock off and on without changing its accumulated value. This allows implementation of a policy of not charging for certain monitor functions (e.g. page fault handling) if so desired by a system administrator. This clock could also be conditioned off depending on certain processor states, e.g. PI in progress, etc. In fact, this is very much like the accounting meters on the KL10, but it counts only real (execution) time. An important fact to keep in mind is that, whether or not a system administrator chooses to use EBOX/MBOX accounting, the monitor must still maintain process runtimes accurately and efficiently; and most sites do not in fact use other than straight time accounting. We need to be able to maintain ordinary runtime accounting as efficiently as we can maintain EBOX/MBOX accounting.

SUMMARY

In summary, time base handling would be significantly improved by the following hardware support:

1. Multiple time base clocks running at different frequencies as selected by the monitor. Specifically, two clocks running at 1 KHz and 100 KHz respectively.

2. A process runtime clock, preferably running at 100 KHz, which would be loaded automatically on context switch and which could be conditioned to exclude overhead functions.

The present interval timer functions are also used and appear to be sufficient.

EBOX/MBOX ACCOUNTING

The KL10 implemented EBOX/MBOX accounting as an attempt to produce more reproducible charging data. A number of problems have appeared in connection with this however.

1. The measure is not in fact completely reproducible. Instructions restarted because of PI servicing will cause both meters to read higher than if no interrupts had occurred.

2. These measures cannot be used as a substitute for actual runtime or as a pseudo-runtime. TOPS10 attempted to do this and fell into many problems. If these measures are to be used for charging, it must be made clear that they are usage units not related to time.

3. The measure is not reproducible on a different model of machine (particularly if, as on the KS10, the machine does not implement the feature). Nor is EBOX accounting necessarily constant over microcode releases.

I believe that the first priority should be to provide ordinary runtime accounting which is as repeatable as possible. This can be done by providing a process runtime clock as described above which can be conditioned to not count during overhead functions. This, of course, will not be constant over different machines, but the system administrator can adjust the charging rate as desired to reflect different processor speed.

A truly constant charging scheme would assign a basic charge to each machine instruction and each canonical operation, such as an indirect cycle. The microcode store would have to hold the number of charge units for each chargable entity, and accumulate the total as used. Variable-length operations such as divide would have to be examined to determine whether to assess a constant charge or a variable charge based on the actual work required. The more such variable charges are included, the more difficult it becomes to maintain constant charging over various machine models and speeds.

I DO NOT FAVOR ATTEMPTING TO DO THIS. The complexity necessary to fully achieve the result is probably not worthwhile. Further, there are an infinite number of combinations of charging rules, and any one combination is unlikely to satisfy more than the one customer who proposed it. Ultimately, the cost of owning or operating a computer is a function of time--lease payments or amortization schedule, operator wages, even electric power cost is a function of how long the machine is turned on, not how much work it does. Hence, any charging scheme not a function of time is arbitrary and unlikely to be generally acceptable. Many service bureaus charge on the basis of clearly identifyable service delivered, such as number of employees in the payroll or number of lines in the output report. This is the proper way to implement constant charging, and can even be made to work on machines from different vendors.

Hence, I conclude that:

1. We should attempt to provide efficient and reproducible runtime accounting;

2. The EBOX/MBOX meters are of limited utility and are not a substitute for runtime accounting. The MBOX meter has some value in helping a user to understand and optimize his program. If the cost is small, these meters should be retained on Dolphin.

3. Fully general constant accounting is quite difficult and of questionable value. Nothing beyond the EBOX/MBOX meters appears warranted.