

AD-951302

(2)



**TECHNICAL REPORT RD-GC-88-38**

**A DESCRIPTION OF THE COMPUTER SIMULATION  
OF A NEW BUS ARBITRATION SCHEME**

G. Patton Bradford  
Guidance & Control Directorate  
Research, Development, & Engineering Center

DECEMBER 1988

DTIC  
ELECTE  
FEB 14 1989  
S D D

AD-A203 930



**U.S. ARMY MISSILE COMMAND**

*Redstone Arsenal, Alabama 35898-5000*

Approved for public release; distribution unlimited.

UNCLASSIFIED  
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188 Exp. Date: Jun 30, 1986	
1a. REPORT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>		1b. RESTRICTIVE MARKINGS			
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		Approved for public release; distribution unlimited.			
4. PERFORMING ORGANIZATION REPORT NUMBER(S)  RD-GC-88-38		5. MONITORING ORGANIZATION REPORT NUMBER(S)			
6a. NAME OF PERFORMING ORGANIZATION Guidance & Control Directorate RD&E Center		6b. OFFICE SYMBOL (If applicable) AMSMT-RD-GC-S	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State, and ZIP Code) Commander, US Army Missile Command ATTN: AMSMT-RD-GC-S Redstone Arsenal, AL 35898-5254		7b. ADDRESS (City, State, and ZIP Code)			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION  Same		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification)  A DESCRIPTION OF THE COMPUTER SIMULATION OF A NEW BUS ARBITRATION SCHEME (U)					
12. PERSONAL AUTHOR(S) G. Patton Bradford					
13a. TYPE OF REPORT Final Technical		13b. TIME COVERED FROM Mar 88 to May 88	14. DATE OF REPORT (Year, Month, Day) DECEMBER 1988		15. PAGE COUNT 30
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Computer bus, computer architecture, arbitration		
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>The object of bus arbitration schemes is to provide fair access to the bus by the various elements of the computer system. Arbitration must allow for each element to have access to the bus while, at the same time, preventing any single element from monopolizing the bus. The method of bus arbitration presented here appears to meet these goals.</p> <p>The arbitration scheme described in this report places the various elements of the system on a rotating system of priorities. After the various priorities are set initially (0...N-1, where N is the number of system elements) all the priorities are incremented by one on a modulo N basis. In this way, each processor will eventually have top priority and be able to take possession of the bus if needed.</p> <p>In a simulation of this scheme using a random mix of bus demands, the average wait of any processor for the bus was very short, generally less than 10 bus cycles ("bus cycle" defined in the body of the report.) For simulations in which the bus demand was high (a probability (cont'd))</p>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> OTC USERS			21. ABSTRACT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>		
22a. NAME OF RESPONSIBLE INDIVIDUAL G. Patton Bradford		22b. TELEPHONE (Include Area Code) (205) 876-7081		22c. OFFICE SYMBOL AMSMT-RD-GC-S	

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.  
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

DD 1473 (cont'd)

Item 19. Abstract

of requesting the bus of 0.75 and a probability of retaining the bus of 0.75), the average delay was seven cycles.

The simulation program itself is described in detail and the results of various combinations of probabilities are presented.

#### ACKNOWLEDGEMENTS

I must be honest and state right from the beginning that this arbitration scheme is not mine; I did not think of it, and I can in no way lay claim to it. This scheme is the idea of Mr. James Holeman of Systems Dynamics, Inc. During some discussions on the design of an experimental digital signal processing board, Jim presented this idea as a simple, reliable, and robust method for arbitrating access to a common bus between various system boards. This simulation appears to bear out some of his hopes for the design in that it provides for short average delays as well as a relative immunity to system failure due to the failure of system components.

Thanks, Jim. I appreciate it.

TABLE OF CONTENTS

	<u>Page</u>
I. INTRODUCTION.....	1
II. ARBITRATION ALGORITHM DESCRIPTION.....	2
III. BUS SIMULATION PROGRAM.....	3
IV. SIMULATION RESULTS.....	4
V. CONCLUSIONS.....	5
APPENDIX A. Source Code Listing.....	A-1
APPENDIX B. Results of Runs.....	B-1

## I. INTRODUCTION

One problem that continues to face the designer of computer systems in which various system elements must share common resources is the arbitration of access to these resources. In most systems, this comes down to the sharing of a common bus that connects these resources to each other. The question is: how can access to this resource, the bus, be handled in a "fair" manner? That is, how can one arrange for every potential user of the bus to have adequate access to the bus while preventing any one user from monopolizing it? Also, how can this be done in a way that is the least taxing in terms of hardware and software?

The method put forth in this paper for handling this problem is a conceptually simple solution involving a rotating priority system. In this way, all boards at some point would have exclusive, and in some cases non-interruptible, access to the bus. Because the priority rotates, no one individual user of the bus can monopolize it. This scheme is described in the next section.

Following this discussion of the arbitration algorithm is an extensive treatment of the program used to simulate the action of the arbitration scheme. Finally, there is a discussion of the results of a number of runs using different values for the probabilities of requesting the bus and of completing a bus access.

## II. ARBITRATION ALGORITHM DESCRIPTION

The arbitration scheme itself is conceptually very simple. The  $N$  processors, or bus users, are initially assigned priorities of 0 to  $N-1$ , each processor receiving a different priority. While no accesses are made to the bus, or, more accurately, no processor releases the bus, the priorities remain the same. However, when a processor obtains and then releases the bus, all processors have their priorities incremented by one on a modulo  $N$  basis; that is, the processor with priority 0 goes to priority 1, priority  $N-2$  goes to  $N-1$ , and priority  $N-1$  goes to 0. In this way, the highest priority processor, the one with the greatest probability of having had access to the bus now receives the lowest priority. A "fairer" method would have the processor releasing the bus receive the lowest priority; however, this would require some method for doing a wholesale reassignment of priorities, whereas the method described requires only incrementing counters on each processor.

However, once a processor receives the bus, it does not necessarily have non-interruptible access to it; the processor may lose the bus to a processor having higher priority. If this happens, the executing process is blocked, the processor relinquishes the bus, the priorities of all the processors are updated, and the requesting processor with the highest priority is granted the bus. The blocked processor now has a pending bus request that will be answered when its priority again exceeds that of all other requesting processors.

### III. BUS SIMULATION PROGRAM

In order to develop an understanding as to how this system would perform over time, a program was written that will simulate the performance of this type of scheme over a number of iterations. To allow for a degree of variability, different parameters were set up in the form of probabilities: in particular, the probability of a processor failing, of requesting the bus, and of the processor completing its bus access at any particular time. In this manner, different bus loads could be simulated. The basic structure of the simulation is described below. (See Appendix A for the source code listing.)

The processors are represented in the program as a relatively large data structure containing various flags, probabilities, counters, and value-holders. The flags for each process are: ACTIVE, which shows if the processor is healthy; BLOCKED, which is set if the processor has the bus but has been interrupted; WAITING, which is set if the processor has a bus request which has not been met; and RUNNING, which indicates that the processor has run or is running since the last time update. The probabilities contained in each process structure are those of failure (PROB\_FAIL), making a bus request (PROB\_REQUEST), and of completing a bus request on any given cycle (PROB\_COMPLETE). The counters contain information on the cycles used in the present block, wait, or bus access (PRESENT\_BLOCK, PRESENT\_WAIT, and PRESENT\_RUN), the total number of cycles spent in blocks, waits, or bus accesses (TOTAL\_BLOCK, TOTAL\_WAIT, NUM\_BLOCKS, NUM\_WAITS, and NUM\_RUNS). The longest of any individual block, wait, or run is also tracked (LONGEST\_BLOCK, LONGEST\_WAIT, LONGEST\_RUN). Finally, and perhaps most important to this simulation, is PRIORITY, which contains the processor's present priority.

To begin the simulation each processor structure has its values initialized through the procedure INIT(). The ACTIVE flag is set, the BLOCKED, WAITING, and RUNNING flags are cleared, the counters are cleared, and using the procedure SET\_PRIORITY(), the priority of each processor is randomly set to an integer value between 0 and N-1 (since in this simulation the processors cannot run or request the bus concurrently, they must instead be polled sequentially.) This random setting of the priorities of the different processors helps to nullify any effects that might occur due to having the processors arranged sequentially by priority. After this, the various probabilities are set. The program allows for this to be done randomly with a default value or with the user specifying the probability for each processor individually. These options allow for the tailoring of the bus loading to one's preference.

The main program itself consists mainly of two large loops. The outer loop determines how many times polling of the processors occurs and the inner loop determines how each processor will act when it is polled. At the beginning of the outer loop are print statements that report the status of the system every thousand iterations. At the end of the outer loop is a call to the UPDATE\_TIME() procedure which updates the counters within each processor structure, incrementing as necessary the number of cycles in the RUN, WAIT, or BLOCK states. The inner loop is where the actual computations that determine processor state are performed. The routine will check the referenced processor's current state and then, based on this state and the probabilities associated with this processor, the next state of the processor is determined.

At the beginning of this inner loop, the processor under consideration may be in one of two main states, either healthy and functioning (ACTIVE=1) or "dead" (ACTIVE=0), having failed at some point in the past. If the processor is dead, then the program continues on to the next processor. Figures 1 through 3 are flow charts of this portion of the program.

On the other hand, if the processor has not failed, it will be in one of four states as it enters processing. It will either be the present bus master, it will be blocked (it had the bus at one time but was preempted before it could complete the transaction), it will be waiting (the processor has requested the bus, but has not yet received it), or it could have no pending requests. Each of these states requires that different actions be taken and each have different alternatives to choose. Each of these alternatives is chosen through the use of the probabilities assigned to each processor and the actions of a random number generator. Any time the value received from the random number generator is less than the value of the assigned probability, the action associated with that probability will be performed.

The first state to be looked at is that of the processor being the present bus master. This is perhaps the simplest of all the states to process. The first check is to see if the processor fails at this point. If it does, then its ACTIVE flag is cleared, BUS\_MASTER and PRIORITY are set to FREE, and UPDATE\_PRIORITY() is called. If the processor remains active, then RUN\_PROCESS() is called to see if the processor will complete its access this cycle. The program then continues on to the next processor.

RUN\_PROCESS() is a very simple routine itself. If the PROB\_COMPLETE associated with this process is greater than some random number, then the bus is freed again. UPDATE\_PRIORITIES() is then called.

With the next three states, BLOCKED, WAITING, and NO\_PENDING\_REQUESTS, again the possibility of processor failure is checked first, and, with these three, the processing is the same. If the processor does fail, ACTIVE is cleared, LONGEST\_WAIT, LONGEST\_BLOCK, TOTAL\_WAIT, and TOTAL\_BLOCK are all updated with respect to PRESENT\_WAIT and PRESENT\_BLOCK. The flags WAITING and BLOCKED are then cleared.

The next check made on processors in one of these three states is whether or not it will make a request for the bus. For the BLOCKED and WAITING states this is automatic. For the NO\_PENDING\_REQUESTS state, this is checked through its PROB\_REQUEST value. If there is no request, the program goes on to the next processor.

Next, having made a request for the bus, the priority of the requester is compared to that of the present bus master, that is, to the value stored in the global variable PRIORITY. If the priority of the requester is the lesser of the two values, then the program continues on with the next processor; otherwise, the requesting processor becomes the new bus master. If the bus is free, then this is accomplished by writing the processor's number into BUS\_MASTER, its present priority into PRIORITY, and its BLOCKED and WAITING flags cleared. After this, RUN\_PROCESS() is called to see if it will complete its access in this cycle. The program then continues on to the next processor.

If, on the other hand, the bus is not free when the requester receives the bus, the old bus master must be blocked. This is accomplished by setting the bus master's `BLOCKED` flag, incrementing its `NUM_BLOCKS`, and calling `UPDATE_PRIORITIES()`. The requester is then given control of the bus as in the previous paragraph.

After the program has stepped through all `N` of the processors, it is considered to have completed one `BUS CYCLE`. At the end of each bus cycle, the routine `UPDATE_TIME()` is called in order to keep track of the amount of time spent by each processor, either on the bus, waiting for initial access to the bus, or in a blocked state. The routine looks at each processor data structure, looking at what flags are set, and updating the appropriate times. First, it looks at the `RUNNING` flag. If it is set and the processor is the present bus master, then `PRESENT_RUN` is incremented and `LONGEST_RUN` is updated. If it is not the bus master, then `PRESENT_RUN` is cleared. In any case, the `RUNNING` flag is cleared. Next, `UPDATE_TIME()` checks the `BLOCKED` and `WAITING` flags and updates either `PRESENT_WAIT` or `PRESENT_BLOCK`, respectively. Finally, the routine sets the `RUNNING` flag of the `PRESENT_BUS` master.

At this point, there may exist some confusion over why `UPDATE_TIME()` clears and then sets the `RUNNING` flag while the other flags are left alone. The reason is that, with the structure of the simulation, once a processor is put on `WAIT` or `BLOCK`, it remains there until the next bus cycle. On the other hand, if it is running, it may be blocked any time during the present cycle or some future cycle. In order to keep track of the amount of time some processor had the bus, it is assumed that the processor keeps the bus for one entire cycle if it had the bus for any portion of that cycle. For this reason, when a processor is blocked, the `RUNNING` flag is not reset until `UPDATE_TIME()` is called at the end of the cycle. In this way, all the processors that ran during the cycle get credit for that cycle, and by setting the bus master's `RUNNING` flag again, it will get credit the next time the update routine is called at the end of the next cycle.

After the outer loop has run the required number of iterations (10,000 in this simulation), the results are tallied. The total amount of time spent running in a `WAIT` or in a `BLOCK` is tallied, as well as the total number of times processors were put in a `RUN`, `WAIT`, or `BLOCK`. From these, average times may be computed. Also reported are the longest times in any state.

#### IV. SIMULATION RESULTS

Appendix B gives the results of a number of runs. The first three use random values for the probabilities of failure, request, and completion. (The intermediate results from the first run are presented in order to give the reader an idea of what is displayed during a run.) As can be seen from these, although the total times and longest times spend in a WAIT or BLOCK state have considerable variation from one run to another, the average time in a WAIT or BLOCK remains consistently small, generally less than 10 cycles. In addition, the total number of cycles in which the bus was active remained relatively constant.

The rest of the runs presented represent situations that range from a very heavily loaded bus (a high probability of requesting the bus along with a low probability of completing an access) to a very lightly loaded bus (a low probability of request with a high probability of completion.) As can be seen on the heavy loading run, although the number of cycles the bus is in use increases by about half over the random loading, the amount of time in a BLOCK or WAIT more than doubles. However, the average time spent in a BLOCK or WAIT is still under 10 cycles - the same as with the random case. The next case, high probability for both request and completion, is perhaps the ideal bus loading situation. Bus utilization is more than doubled over any other case. In addition, the average WAIT and BLOCK is down to one.

The case in which both the probabilities are low (a request is not likely, but if there is one, it is not likely to complete on any given cycle) is about average when compared to the other cases. The average delays are in the same range as all the others, under 10 cycles, and because there are fewer requests, the bus is used less often than the higher request situations, even the one with the high blocking rate. The final situation, with low request probability and high completion probability, has the lowest rate of blocking and waiting, though not much lower than the high request/high complete. Again, because the request rate is low, bus utilization is dropped.

#### V. CONCLUSIONS

This system of bus arbitration using a rotating priority scheme appears to be an effective and efficient method of handling bus arbitration. Average delays are consistently low in all cases tested; failure of individual system elements appears to have little effect on overall system performance; and the conceptual simplicity of the system should allow for relatively efficient implementations in hardware.

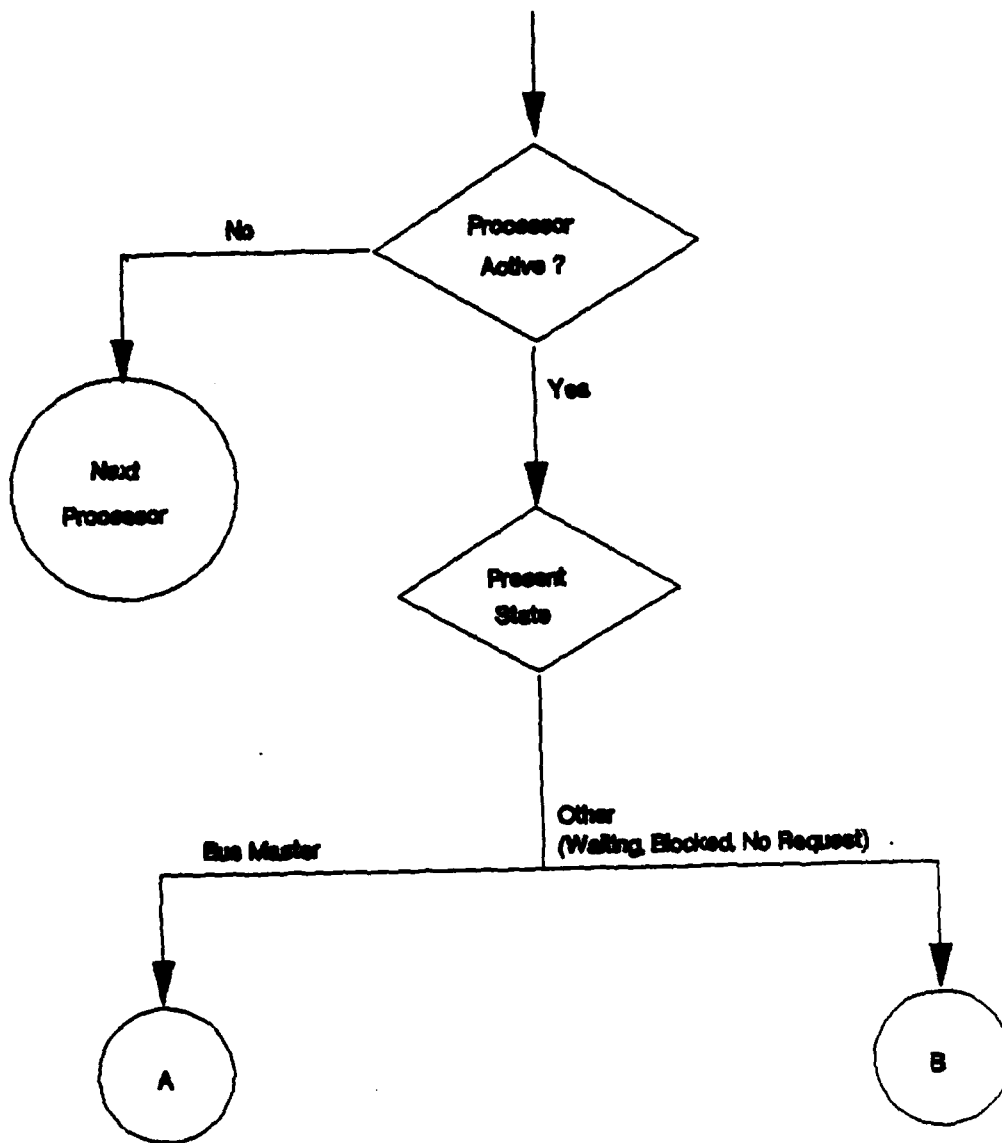


Figure 1. Flowchart for inner processing loop.

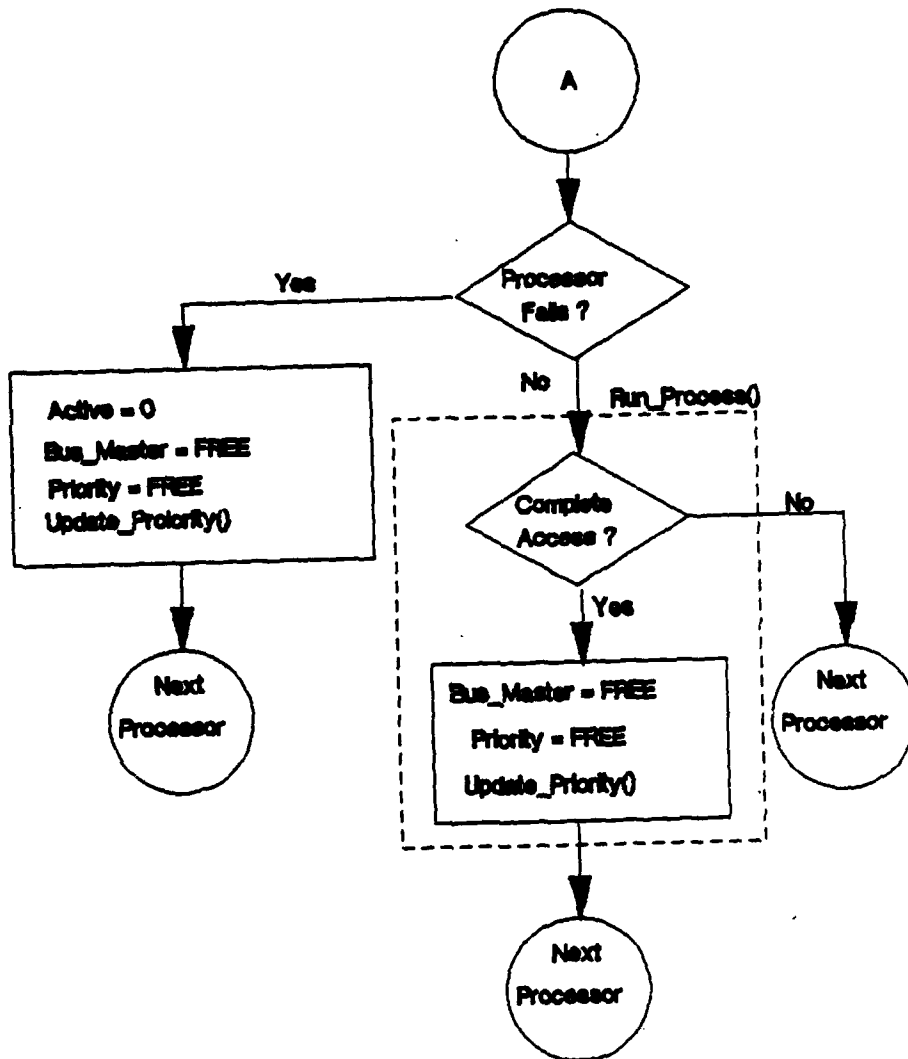


Figure 2. Flowchart of bus master processing loop.

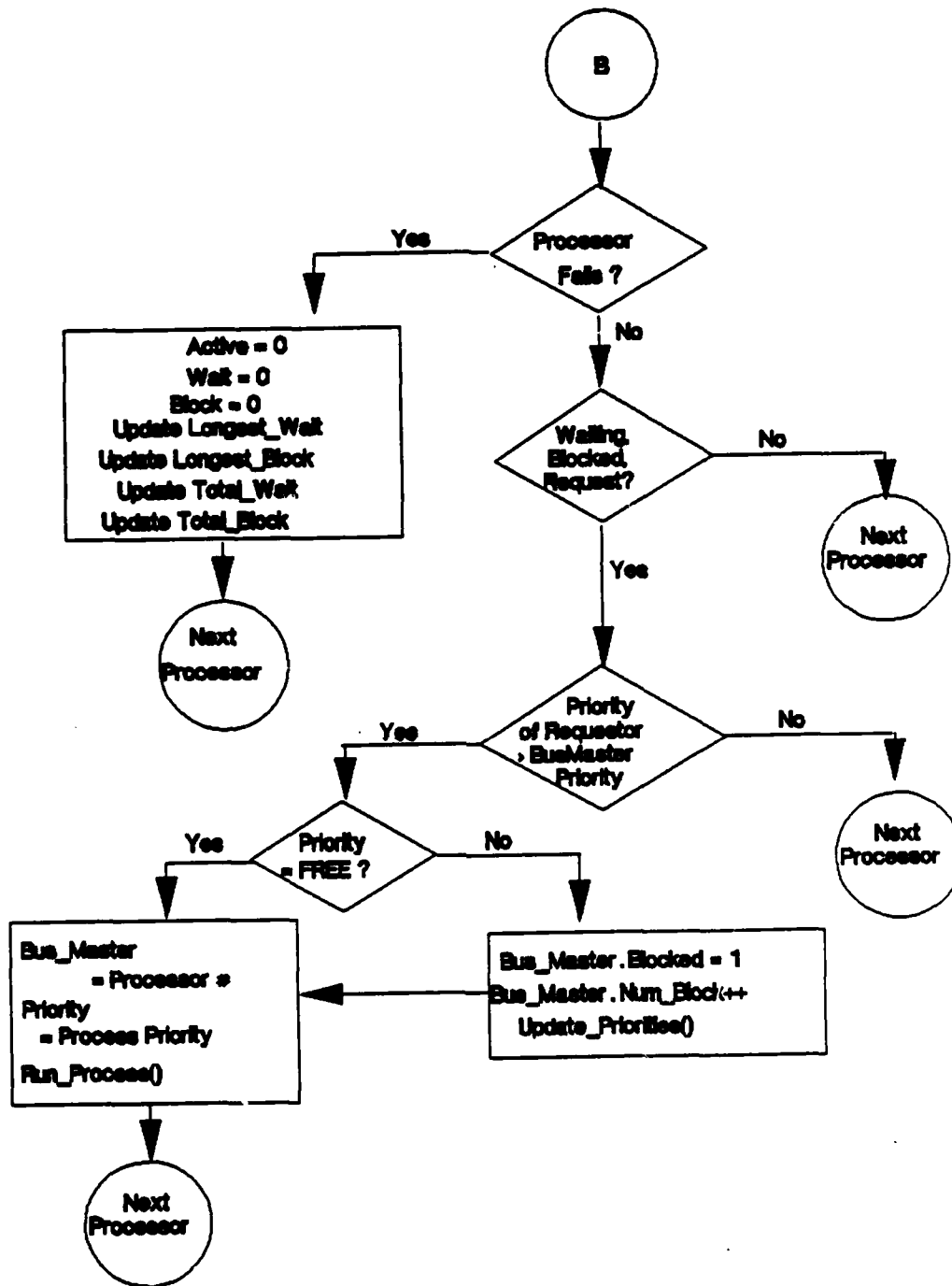


Figure 3. Flowchart of other (waiting, blocked, no request) processing loop.

APPENDIX A  
SOURCE CODE LISTING

```
Line# Source Line Microsoft C Compiler Version 4.00
.....
/*
/* busim.c - Program to simulate the rotating priority bus arbitration
/* scheme. The program will run the main cycle (all ten processors being
/* polled once) 10,000 times. It will then report on the maximum and
/* average delays encountered.
.....
/* #include "include.h" */
#include <stdio.h>
#include <dos.h>
.....
/*
/* global.h - Defines such things as the structure for the processes
.....
#define NUMPROCESSORS 10
#define MAX - 1
#define MAXLOOPS 10000
struct Process
char Active; /* flag showing whether or not the processor */
char Blocked; /* has failed to be processed but was interrupted */
char Waiting; /* has requested bus but has not received it */
char Running; /* has run during this cycle */
short int Priority; /* Present priority of processor */
float ProbFail; /* The probability that this processor will fail */
float ProbRequest; /* The probability that a bus request will be
/* made on some cycle */
float ProbComplete; /* The probability that, if the processor has
/* the bus, it will complete this cycle */
int PresentWait; /* The number of cycles waiting on present request */
int LongestWait; /* The longest wait on a bus request */
int TotalWait; /* Total number of cycles in a wait state */
int NumWait; /* The number of times processor made to wait */
int PresentBlock; /* The number of cycles blocked */
int LongestBlock; /* The longest time blocked */
int TotalBlock; /* Total number of cycles in a blocked state */
int NumBlock; /* The number of times executing process is
/* blocked */
int PresentRun; /* The number of non-interrupted cycles possessing
/* the bus */
int LongestRun; /* The longest time spent in any single run */
.....
```

```
Line# Source Line Microsoft C Compiler Version 4.00
.....
int TotalRun; /* The total amount of time this processor has
/* had the bus */
int NumRuns; /* the number of times the processor has requested
/* and received the bus */
} ProcArray[NUMPROCESSORS];
long Seed; /* The seed value for the random number generator */
short Priority = 0; /* Present owner of the bus */
short Priority = 0; /* Priority of owner of the bus */
int TotalWait = 0; /* sum of totalwait */
int TotalBlock = 0; /* sum of totalblock */
int TotalRun = 0; /* sum of totalrun */
int NumWait = 0; /* sum of numwait */
int NumBlock = 0; /* sum of numblock */
int NumRuns = 0; /* sum of numruns */
int LongestWait = 0; /* longest of longestwait */
int LongestBlock = 0; /* longest of longestblock */
int LongestRun = 0; /* longest of longestrun */
int LargestWait = 0; /* largest number of waits */
int LargestBlock = 0; /* largest number of blocks */
int LargestRun = 0; /* largest number of runs */
float AvgWait; /* average time in a wait state */
float AvgBlock; /* average time in a block state */
float AvgRun; /* average time in a bus possession */
.....
/* End of global.h */
.....
/*
/* Random() - function to produce a Random number x, 0 <= x < 1.
.....
```

```

float Random()
/* Routine to produce the random numbers */
{
    Seed = (25173L * Seed + 13849L) % 65536;
    return (((float)Seed)/((float)65536));
} /* End Random() */

.....
/*
UpdateTime() - Update the wait times on all the processors
.....

```

PAGE 3  
07-11-83  
14:22:08

Microsoft C Compiler Version 4.00

```

Line# Source Line
void UpdateTime()
{
    short int i;
    for (i=0; i<=NUMPROCESSORS; i++)
        /* Update the time on any processor that has run this cycle */
        {
            if (ProcArray[i].Running)
                {
                    if (BusMaster == 1)
                        ProcArray[i].PresentRun++;
                    if (ProcArray[i].PresentRun > ProcArray[i].LongestRun)
                        ProcArray[i].LongestRun = ProcArray[i].PresentRun;
                }
            else /* Clear PresentRun if processor does not presently
                have the bus */
                ProcArray[i].PresentRun = 0;

            /* Update longestrun and totalrun for all the processors.
            Clear all of the Running flags. */
            if (ProcArray[i].LongestRun)
                ProcArray[i].LongestRun = 1;
            ProcArray[i].TotalRun++;
            ProcArray[i].Running = 0;
        }

        /* Update the time on each blocked or waiting process */
        if (ProcArray[i].Blocked)
            ProcArray[i].PresentLock++;
        else
            if (ProcArray[i].Waiting)
                ProcArray[i].PresentWait++;
        }
    } /* End of UpdateTime() */
}

```

UpdateTime Local Symbols

Name	Class	Offset	Register
i	auto	-0002	

```

.....
/*
SetPriority() - Do random initialization of priorities
.....

```

PAGE 4  
07-11-83  
14:22:08

Microsoft C Compiler Version 4.00

```

Line# Source Line
void SetPriority()
{
    char IsSet[NUMPROCESSORS];
    int i, x;

    /* Initialize IsSet to 0 */
    for (i=0; i<=NUMPROCESSORS; i++)
        IsSet[i] = 0;

    /* Initialize priorities. Get random number between 0 and
    NUMPROCESSORS. If already in use, choose another. Otherwise,
    assign x to ProcArray[i].Priority and set IsSet[x] */
    for (i=0; i<=NUMPROCESSORS; i++)
        do
            x = ((int)() * Random());
        while (IsSet[x]);
        ProcArray[i].Priority = x;
        IsSet[x] = 1;
    } /* End of SetPriority() */
}

```

SetPriority Local Symbols

Name	Class	Offset	Register
i	auto	-0002	
x	auto	-0001	



```

ProcArray[0].NumUnits = 0;
ProcArray[0].PresentLock = 0;
ProcArray[0].CurrentLock = 0;
ProcArray[0].NumLocks = 0;
}
/* Set the values for the various probabilities */
pick1: /* ProbFail */
printf("Do you wish to set the probabilities of failure?\n");
printf("(R) Randomly\n(D) default value, or\n(I) individually?\n");
scanf("%c", &Answer);
switch ((int)Answer)
{
case 'R':
case 'r':
for(i=0; i<NUMPROCESSORS; i++)
{
ProcArray[i].ProbFail = .0001 * Random();
/* Low probability is required to keep them from
all failing during the first few iterations.
*/
printf("Probability of failure %d: %f\n",i,ProcArray[i].ProbFail);
}
break;
case 'D':
case 'd':
printf("Default value?\n");
scanf("%f", &Value);
for(i=0; i<NUMPROCESSORS; i++)
ProcArray[i].ProbFail = Value;
break;
case 'I':
case 'i':
for(i=0; i<NUMPROCESSORS; i++)
{
printf("Probability of failure for processor %d ",i);
scanf("%f", &Value);
ProcArray[i].ProbFail = Value;
}
printf("\n");
break;
}

```

PAGE 7  
09-03-88  
14:22:08

```

Line# Source Line Microsoft C Compiler Version 4.00
default:
printf("Dumb. Very dumb. Try again.\n");
goto pick1;
}
printf("\n");
/* ProbRequest */
pick2: printf("Do you wish to set the probabilities of making a bus request?\n");
printf("(R) Randomly\n(D) default value, or\n(I) individually?\n");
scanf("%c", &Answer);
switch ((int)Answer)
{
case 'R':
case 'r':
for(i=0; i<NUMPROCESSORS; i++)
{
ProcArray[i].ProbRequest = Random();
printf("Probability of bus request %d: %f\n",i,ProcArray[i].ProbRequest);
}
break;
case 'D':
case 'd':
printf("Default value?\n");
scanf("%f", &Value);
for(i=0; i<NUMPROCESSORS; i++)
ProcArray[i].ProbRequest = Value;
break;
case 'I':
case 'i':
for(i=0; i<NUMPROCESSORS; i++)
{
printf("Probability of request for processor %d ",i);
scanf("%f", &Value);
ProcArray[i].ProbRequest = Value;
}
printf("\n");
break;
default:
printf("Dumb. Very dumb. Try again.\n");
goto pick2;
}
printf("\n");
/* ProbComplete */
pick3: printf("Do you wish to set the probabilities of completing a bus access?\n");
printf("(R) Randomly\n(D) default value, or\n(I) individually?\n");
scanf("%c", &Answer);
switch ((int)Answer)
{
case 'R':
case 'r':

```

```

Line# Source Line
370 for(i=0; i<NUMPROCESSORS; i++)
371 {
372 ProcArray[i].ProbComplete = Random();
373 printf("Probability of completion %d: %f\n",i,ProcArray[i].ProbComplete);
374 }
375 break;
376 case 'd':
377 printf("default value?\n");
378 scanf("%d",&Value);
379 for (i = 0; i < NUMPROCESSORS; i++)
380 ProcArray[i].ProbComplete = Value;
381 break;
382 case 'i':
383 for (i = 0; i < NUMPROCESSORS; i++)
384 {
385 printf("Probability of completion for processor %d: ",i);
386 scanf("%d",&Value);
387 ProcArray[i].ProbComplete = Value;
388 }
389 printf("\n");
390 break;
391 default:
392 printf("Dumb. Very dumb. Try again.\n");
393 goto pick3;
394 }
395 printf("\n");
396 } /* End of Init() */

```

Init Local Symbols

Name	Class	Offset	Register
inregs	auto	-0024	
answert	auto	-0016	
outregs	auto	-0010	
Value	auto	-0004	si

```

377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

```

Line# Source Line
Microsoft C Compiler Version 4.00
(
370 printf("%d iterations\n", i);
371 for (j=0; j<NUMPROCESSORS; j++)
372 printf("Init Cycles: %d \tBlock Cycles: %d \tRun cycles: %d\n",
373 ProcArray[j].TotalInit, ProcArray[j].TotalBlock, ProcArray[j].TotalRun);
374 printf("\n\n");
375 }
376 for (j = 0; j<NUMPROCESSORS; j++)
377 {
378 if (!ProcArray[j].Active)
379 /* Processor is dead. Do nothing */
380 continue;
381 if (j == BusMaster)
382 /*
383 A. Bus
384 Mark as dead
385 Release bus
386 Update Priorities
387 B. Complete access
388 Release bus
389 Update Priorities
390 C. Continue access
391 Update times
392 */
393 {
394 if (Random() < ProcArray[j].ProbFail)
395 /* The processor just failed. Mark as dead. Release
396 bus by setting BusMaster to FREE and Priority to
397 FREE. Update all of the priorities */
398 {
399 printf(" Processor %d failed at time %d\n", j, i);
400 ProcArray[j].Active = 0;
401
402 BusMaster = FREE;
403 Priority = FREE;
404 UpdatePriority();
405 continue;
406 } /* end of Bus Master failure */
407 /* Check to see if process will complete during

```

```

    this bus cycle. Else continue to hold bus. */
    RunProcess(ProcArray[j].ProbComplete);
    continue;
} /* end of processes if processor already has bus */
/* II. Does not have bus.
A. Dies
    1. If blocked or waiting
        Update Total/LongestBlock/Wait

```

PAGE 10  
05-03-88  
14:22:08

Line# Source Line

```

Microsoft C Compiler Version 4.00
    Clear block and end wait
B. Does not die
    1. If blocked, waiting, or makes request
        a. If gets bus
            i. If bus active
                Mark bus master as blocked
                Increment block time
                Get bus
                Update Total/LongestBlock/Wait
                Clear blocked/waiting flags
                Update priority
                Update time
            ii. If bus inactive
                Get bus
                Update time
        b. If not get bus
            If not blocked or waiting,
                mark as blocked.
                update time
    2. No request
        Do nothing.
*/
if (Random() < ProcArray[j].ProbFail)
/* The processor just failed. Mark as dead. */
printf(" Processor %d failed at time %d\n", j, 1);
ProcArray[j].Active = 0;
if (ProcArray[j].Blocked || ProcArray[j].Waiting)
/* If the failed processor was blocked or waiting,
update the Total/LongestBlock/Wait blocks and clear
the Blocked/Waiting flags */
{
    if(ProcArray[j].PresentWait > ProcArray[j].LongestWait)
        ProcArray[j].LongestWait = ProcArray[j].PresentWait;
    if(ProcArray[j].PresentBlock > ProcArray[j].LongestBlock)
        ProcArray[j].LongestBlock = ProcArray[j].PresentBlock;
    ProcArray[j].TotalWait += ProcArray[j].PresentWait;
    ProcArray[j].TotalBlock += ProcArray[j].PresentBlock;
    ProcArray[j].PresentWait = 0;
    ProcArray[j].PresentBlock = 0;
    ProcArray[j].Blocked = 0;
    ProcArray[j].Waiting = 0;
}
/* else do nothing */
continue;
} /* end of failed processor */
/* Else it did not die. Check and see if it is waiting,
blocked, or making a bus request.
if(ProcArray[j].Waiting || ProcArray[j].Blocked ||
Random() < ProcArray[j].ProbRequest)

```

PAGE 11  
05-03-88  
14:22:08

Line# Source Line

```

Microsoft C Compiler Version 4.00
/* Has made request for bus. See if it will get it */
if(ProcArray[j].Priority > Priority)
{
    if (Priority != FREE)
        /* bus already active. Block and replace old
        bus master */
        {
            ProcArray[BusMaster].Blocked++;
            ProcArray[BusMaster].NumBlocks++;
            BusMasterPriority();
            BusMasterPriority();
            ProcArray[j].Running = 1;
            ProcArray[j].NumBlocks++;
            Priority = ProcArray[j].Priority;

            if(ProcArray[j].PresentWait > ProcArray[j].LongestWait)
                ProcArray[j].LongestWait = ProcArray[j].PresentWait;
            if(ProcArray[j].PresentBlock > ProcArray[j].LongestBlock)
                ProcArray[j].LongestBlock = ProcArray[j].PresentBlock;
            ProcArray[j].TotalWait += ProcArray[j].PresentWait;
            ProcArray[j].TotalBlock += ProcArray[j].PresentBlock;
            ProcArray[j].PresentWait = 0;
            ProcArray[j].PresentBlock = 0;
            ProcArray[j].Blocked = 0;
            ProcArray[j].Waiting = 0;
        }
    RunProcess(ProcArray[j].ProbComplete);
}

```

```

/* Bus is free */
{
    BusMaster = i;
    Priority = ProcArray[i].Priority;
    ProcArray[i].Running = 1;
    ProcArray[i].NumRuns++;

    if(ProcArray[i].PresentWait > ProcArray[i].LongestWait)
        ProcArray[i].LongestWait = ProcArray[i].PresentWait;
    if(ProcArray[i].PresentBlock > ProcArray[i].LongestBlock)
        ProcArray[i].LongestBlock = ProcArray[i].PresentBlock;
    ProcArray[i].TotalWait += ProcArray[i].PresentWait;
    ProcArray[i].TotalBlock += ProcArray[i].PresentBlock;
    ProcArray[i].PresentWait = 0;
    ProcArray[i].PresentBlock = 0;
    ProcArray[i].Blocked = 0;
    ProcArray[i].Waiting = 0;
    RunProcess(ProcArray[i].ProcComplete);
    continue;
} /* end else bus is free */
} /* end if get bus */

/* Else will not get bus. Put on wait if not
blocked or waiting already. Update times.
*/

```

PAGE 12  
05-03-88  
14:22:08

Microsoft C Compiler Version 4.00

```

Line# Source Line
{ if (!ProcArray[i].Blocked && !ProcArray[i].Waiting)
    ProcArray[i].Waiting = 1;
    ProcArray[i].NumWaits++;
}
} /* end if process alive */
} /* end processor loop */
UpdateTime();
} /* End main loop */
printf("\n");
for (i = 0; i < NUMPROCESSORS; i++)
    SumWaitTime += (long) ProcArray[i].TotalWait;
    SumLockTime += (long) ProcArray[i].TotalBlock;
    SumRunTime += (long) ProcArray[i].TotalRun;
    SumWaits += (long) ProcArray[i].NumWaits;
    SumLocks += (long) ProcArray[i].NumLocks;
    SumRuns += (long) ProcArray[i].NumRuns;

if(!ProcArray[i].NumWaits)
    printf("Processor %d never waited\n", i);

if(!ProcArray[i].NumLocks)
    printf("Processor %d was never blocked\n", i);

if(!ProcArray[i].NumRuns)
    printf("Processor %d never had the bus\n", i);

if(MaxWaitTime < ProcArray[i].LongestWait)
    MaxWaitTime = ProcArray[i].LongestWait;

if(MaxBlockTime < ProcArray[i].LongestBlock)
    MaxBlockTime = ProcArray[i].LongestBlock;

if(MaxRunTime < ProcArray[i].LongestRun)
    MaxRunTime = ProcArray[i].LongestRun;

if(MaxWaits < ProcArray[i].NumWaits)
    MaxWaits = ProcArray[i].NumWaits;

if(MaxLocks < ProcArray[i].NumLocks)
    MaxLocks = ProcArray[i].NumLocks;

if(MaxRuns < ProcArray[i].NumRuns)
    MaxRuns = ProcArray[i].NumRuns;
}

printf("\nTotal number of cycles blocked: %d\n", SumBlockTime);
printf("\nTotal number of cycles in wait: %d\n", SumWaitTime);
printf("\nTotal number of cycles on bus: %d\n", SumRunTime);
printf("\nLongest time for a block: %d\n", MaxBlockTime);
printf("\nLongest time for a wait: %d\n", MaxWaitTime);
printf("\nLongest time in run: %d\n", MaxRunTime);

```

PAGE 13  
05-03-88  
14:22:08

Microsoft C Compiler Version 4.00

```

Line# Source Line
printf("Average time in a block: %d\n", SumBlockTime/SumNumLocks);
printf("Average time in a wait: %d\n", SumWaitTime/SumNumWaits);
printf("Average time running: %d\n", SumRunTime/SumNumRuns);

MaxNumLocks ?
printf("Max. number of blocks for a processor: %d\n", MaxNumLocks);
! printf("Nobody was ever blocked!\n");

MaxNumWaits ?
printf("Max. number of waits for a processor: %d\n", MaxNumWaits);
! printf("Nobody ever had to wait!\n");

```



**APPENDIX B**

**RESULTS OF RUNS**







Processor 0 was never blocked  
Processor 6 was never blocked

```
Total number of cycles blocked: 16211
Total number of cycles in wait: 1857
Total number of cycles on bus: 16605
Longest time for a block: 70
Longest time for a wait: 70
Longest time in run: 70
Average time in a block: 3
Average time in a wait: 2
Average time running: 2
Max. number of blocks for a processor: 1259
Max. number of waits for a processor: 72
Max. number of runs for a processor: 1778
```

#### D. Run 4: High Probability of Request/Low Probability of Completion

Do you wish to set the probabilities of failure

```
(R)randomly
(D)default value, or
(I)individually?
```

Default value?  
.00001

Do you wish to set the probabilities of making a bus request

```
(R)randomly
(D)default value, or
(I)individually?
```

Default value?  
.8

Do you wish to set the probabilities of completing a bus access

```
(R)randomly
(D)default value, or
(I)individually?
```

Default value?  
.2

```
Total number of cycles blocked: 56845
Total number of cycles in wait: 27538
Total number of cycles on bus: 25271
Longest time for a block: 108
Longest time for a wait: 64
Longest time in run: 50
Average time in a block: 5
Average time in a wait: 7
Average time running: 7
Max. number of blocks for a processor: 1692
Max. number of waits for a processor: 222
Max. number of runs for a processor: 2570
```

#### E. Run 5: High Probability of Request/High Probability of Completion

Do you wish to set the probabilities of failure

```
(R)randomly
(D)default value, or
(I)individually?
```

Default value?  
.00001

Do you wish to set the probabilities of making a bus request

```
(R)randomly
(D)default value, or
(I)individually?
```

Default value?  
.8

Do you wish to set the probabilities of completing a bus access

```
(R)randomly
(D)default value, or
(I)individually?
```

Default value?  
.8

```
Total number of cycles blocked: 11000
Total number of cycles in wait: 16228
Total number of cycles on bus: 58000
Longest time for a block: 8
Longest time for a wait: 8
Longest time in run: 8
Average time in a block: 4
Average time in a wait: 4
Average time running: 4
Max. number of blocks for a processor: 1138
Max. number of waits for a processor: 1206
Max. number of runs for a processor: 6270
```

**F. Run 6: Low Probability of Request/Low Probability of Completion**

Do you wish to set the probabilities of failure  
(0)randomly  
(1)default value, or  
(2)individually?  
0  
Default value?  
.0001

Do you wish to set the probabilities of making a bus request  
(0)randomly  
(1)default value, or  
(2)individually?  
0  
Default value?  
.2

Do you wish to set the probabilities of completing a bus access  
(0)randomly  
(1)default value, or  
(2)individually?  
0  
Default value?  
.2

Total number of cycles blocked: 42550  
Total number of cycles in wait: 21497  
Total number of cycles on bus: 19677  
Longest time for a block: 66  
Longest time for a wait: 66  
Longest time in run: 66  
Average time in a block: 1  
Average time in a wait: 1  
Average time running: 1  
Max. number of blocks for a processor: 1053  
Max. number of waits for a processor: 46  
Max. number of runs for a processor: 1525

**G. Run 7: Low Probability of Request/High Probability of Completion**

Do you wish to set the probabilities of failure  
(0)randomly  
(1)default value, or  
(2)individually?  
0  
Default value?  
.0001

Do you wish to set the probabilities of making a bus request  
(0)randomly  
(1)default value, or  
(2)individually?  
0  
Default value?  
.2

Do you wish to set the probabilities of completing a bus access  
(0)randomly  
(1)default value, or  
(2)individually?  
0  
Default value?  
.8

Total number of cycles blocked: 1068  
Total number of cycles in wait: 5205  
Total number of cycles on bus: 22729  
Longest time for a block: 1  
Longest time for a wait: 1  
Longest time in run: 1  
Average time in a block: 0  
Average time in a wait: 0  
Average time running: 1  
Max. number of blocks for a processor: 286  
Max. number of waits for a processor: 319  
Max. number of runs for a processor: 2119

DISTRIBUTION

	<u>No. Copies</u>
US Army Materiel System Analysis Activity ATTN: AMXSU-MP Aberdeen Proving Ground, MD 21005	1
IIT Research Institute ATTN: GACIAC 10 W. 35th Street Chicago, IL 60616	1
AMSMI-RD, Dr. McCorkle	1
Dr. Rhoades	1
-RD-AS-SS, Mr. Sims	1
-RD-BA	1
-RD-GC, Mr. Sliz	1
-RD-GC-S	5
-RD-CS-R	15
-RD-CS-T	1
-GC-IP	1
AMCPM-ML, Mr. Yarbrough	1
-ML-TM, Mr. Crosswhite	1
-ML-TM-R, Mr. Hill	1