NISTIR 89-4190

GRAMPS MULTIPROCESSOR OPERATING SYSTEM

Peter Mansbach

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Robot Systems Division
Sensory Intelligence Group
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

Lee Morgan, Deputy Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Raymond G. Kammer, Acting Director

NISTIR 89-4190

GRAMPS MULTIPROCESSOR OPERATING SYSTEM

Peter Mansbach

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Robot Systems Division
Sensory Intelligence Group
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

October 1989
Issued January 1990

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

Lee Mercer, Deputy Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Raymond G. Kammer, Acting Director

1/9/90

OVERVIEW OF THE 'GRAMPS' MULTIPROCESSOR OPERATING SYSTEM

Peter Mansbach
National Institute of Standards and Technology
Bldg. 220/Rm. B-124
Gaithersburg, MD 20899

ABSTRACT

GRAMPS is an operating system designed for use by a number of independent functionally-divided processors which communicate via common (shared) memory. It achieves high speed, simplicity, and ready adaptability to users' individual needs by being primarily a single-tasking system, and adding processor boards when concurrent operation of processes is required. It thus trades software cost for known hardware cost.

Three mechanisms of asynchronous data communication between processors are provided. These include a set of *Unix*-compatible calls, so a program can be debugged in a *Unix* environment and simply re-linked to run on a target board. *GRAMPS* allows individual processors to be stopped and restarted, without interrupting the other processors, and allows single processors to be run in isolation. The operating system includes a fast downloader, a monitor (in *PROM*), and an array of debugging tools. *GRAMPS* provides an extremely fast system for single-tasking multiple-processor applications, and some multi-tasking capability as well. It is currently running on Motorola *680x0* microprocessors, and has also run on Intel *8086*'s.

keywords: asynchronous communication; communications protocol; functionally-divided processes; GRAMPS; multiprocessing; multiprocessor; operating system; real-time; robot vision; vision.

OVERVIEW OF THE 'GRAMPS' MULTIPROCESSOR OPERATING SYSTEM

1. Introduction

GRAMPS--the General Real-time Asynchronous Multi-Processor System--is an operating system designed for use by a number of independent functionally-divided processors which communicate with each other via common (shared) memory. *GRAMPS* is currently running on Motorola *680x0* microprocessors, several of which communicate over a *VME* backplane.[1] An earlier version has also run on Intel *8086* processors, with a *Multibus* backplane.

GRAMPS is very fast and very simple. It can be easily modified and tailored to users' individual needs. It achieves these qualities by being primarily a single-tasking system, thus avoiding the complexity of "time-sharing" or multitasking systems. When concurrent operation of processes is required, additional processor boards are easily added; *GRAMPS* handles the communications. Thus, additional hardware (if necessary) is substituted for more complicated, expensive, and error-prone software.

GRAMPS supports both synchronous and asynchronous transfer of messages or other data between processors. Further, three mechanisms of asynchronous data transfer are provided, depending on the needs and preferences of the user. A subset of these communications calls, including *open, close, read, write, printf, getchar,* and *putchar,* use the *Unix*[2] *I/O* calling formats. Thus a program, which is written and compiled on a *Unix* host, may be linked to the standard *Unix* library and debugged on the host. The same program may be re-linked with the *GRAMPS* library and downloaded to the target. On the host the "interprocessor" communication takes place through files rather than common memory.

GRAMPS allows individual processors to be stopped and restarted, without interrupting the other processors, and allows single processors to be run in isolation.

GRAMPS was originally designed to have each processor run a single task. In this mode the operating system executes only when requested by the user process. This permits the individual processes to achieve their full speed potential. Also, the absence of multi-tasking allows great simplicity in the operating system, compared to "time-sharing" systems, and in particular in the writing of input/output device drivers, and in the use of interrupts.

Functions have since been added to allow simple multi-tasking operation. System calls to activate and de-activate tasks are provided, and the on-board timer is programmed to interrupt the running task at a time specified by the scheduler. Thus the user can now perform efficient context-switching, subject to a few constraints. Since simple priority scheduling is often insufficient to satisfy the needs of real-time applications, the user himself can specify and program the exact scheduling algorithm required.

Numerous checks and tools are provided for aid in debugging. These may be turned off to permit still faster execution and smaller code.

Most of the *GRAMPS* code is written in *C*. Some of the basic routines--the message-passing primitives, terminal *I/O*, block moves, etc.--are in assembly language, as are portions of the *PROM*.

2. Background

The *GRAMPS* operating system was developed by the author at the National Bureau of Standards. This work was done in support of a robot vision system which was initially constructed with five Intel *8086* microprocessors in a *Multibus* backplane [1,2,3]. Memory was provided that was accessible to all the processors via the *Multibus*.

It was desired to run the different stages of vision processing concurrently on separate processors, and to pass data from one stage to another asynchronously. Data transfer would take place via common

[1] Commercial products are identified in order adequately to describe the equipment. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the equipment identified is necessarily the best available for the purpose.

memory, by having the processor that generated the information write it into common memory, and having each processor that needed that information read it when required. Mutual exclusion, to prevent one stage from reading data while another was still writing, and vice versa, was to be provided by the operating system.

Commercial operating systems available at the time did not support multiprocessor data passing. (We use the term *multiprocessor* to refer to systems with two or more processors, as distinct from *multi-tasking*, where several tasks may execute on the same *CPU*.) Since data-passing between processors was a central requirement for our vision system, it seemed appropriate to write a simple operating system of our own. In addition, for a research project with unanticipated requirements there is a substantial advantage to writing one's own operating system, and thereby having both the source code and the expertise to modify it readily available. The system is then easily adapted to the needs of the project.

We adopted the following design requirements for the *GRAMPS* operating system:

Support for multiple processors
Efficient data passing via shared memory, often large blocks of data
Support for both synchronous and asynchronous communication
Unix-like *I/O* calls
Ability to restart one process while others continue uninterrupted
Ability to run single processes without others present
Specific identifying messages for all errors in system code
Dynamic common memory allocation
Task-switching capability

2.1. Other Multiprocessor Systems

Since that time a number of other systems have become available with, or have added, multiprocessor capabilities. These include *GEM* [4], *Harmony* [5], *Meglos* [6], *MTOS* [7], *POPEYE* [8], *pSOS* [9], *VRTX* [10], and *VxWorks* [11].

GEM (developed at Ohio State) envisions a process architecture quite different from ours, with processes being *dynamically* assigned to processors, and containing "micro-processes" which can be turned on and off by other micro-processes. *Harmony* (National Research Council, Canada) is in some ways similar to *GRAMPS*, but has more highly developed multi-tasking facilities. However, the *Harmony* system blocks when sending or receiving messages, which may lose a fair amount of potential processing time in an application like ours that is not multi-tasking. *Meglos* (developed at Bell Laboratories) does not use shared memory, but instead bases its communications on the elaborate and very fast *S/NET*, also developed by Bell Labs.

MTOS (Industrial Programming Inc.), *pSOS* (Software Components Group), and *VRTX* (Hunter & Ready) are all systems initially designed to be multi-tasking on a single processor, which have now been extended to allow multiprocessor operations as well. *MTOS* additionally offers dynamic load-balancing among the processors. *POPEYE* (Carnegie-Mellon) is a complete vision system, including a *68000*-based operating system with many features similar to *GRAMPS*. However, there is no mention in the cited reference of communication from one processor board to another. And, finally, *VxWorks* (Wind River Systems) runs using *VRTX* as the operating system for each board, and uses the *Unix socket* formalism to implement interprocessor communications.

3. Major Components of the Operating System

The full *GRAMPS* operating system consists of five physically distinct parts: **kernel, downloader, PROM, SYS** (system process), **and debugging tools.**

3.1. Kernel

The major part of the system code is the *kernel*, which is a collection of subroutines resident in the *GRAMPS* run-time library. These subroutines are included by the linker, and thus become part of each

2 *Unix* is a trademark of AT&T/Bell Laboratories.

processor's program code. The kernel includes the functions dealing with common memory communication with other processes, terminal *I/O*, dynamic memory allocation, process initialization, etc. These are discussed further below.

Note that the *C* language promises that initialized variables start out with the values specified by the programmer, but these values may change during program execution. Thus, to allow a program to be restarted, the *GRAMPS* kernel saves these initial values the first time the program is executed after being downloaded, and subsequently restores them each time the process is restarted. It is not necessary to redownload the program to restart it.

3.2. Downloader

The *downloader* is resident on the host computer, and sends fully compiled and linked programs to the target processor boards for execution there. These programs are sent first to an interface board on the target computer. From there the destination processor copies them to its on-board memory under control of its *PROM*. Clearly the choice of host computer is not important to the rest of the *GRAMPS* system, and can be changed to suit the application.

The downloader performs some alterations to the object code before sending it to the target board. In particular, it finds the symbol table produced by the compiler, compresses it and appends it to the code, for use by the *GRAMPS tools*. It also determines the highest program address, and inserts that, the starting address, and the symbol table address into a table used by the startup subroutines and the *PROM*. The code itself is sent in a compacted format less than half as long as the common *Motorola S*-format, and thus taking less than half the time to download.

The *download* program is independent of the device hardware, except that it calls different device drivers for different hardware. It has been run using standard serial RS-232 non-login lines, and alternatively using 16-bit parallel *I/O* ports. This last is particularly fast. The *download* program could also use a network link, such as an *ethernet* line, or could load over the *VME*bus if the host and target processors share the same backplane.

3.3. PROM

The **PROM** is semi-permanent code "burned in" to read-only memory, one copy per processor board. It includes a wakeup sequence for the board, the "receiving downloader" for receiving data from (or sending it to) the host or development computer, monitor functions for directly examining registers and memory, and other basic debugging aids. An onboard timer chip, if one exists, can be selected to automatically time sections of code. A help command reviews the available monitor commands and formats.

In addition to the usual monitor commands *(display, substitute, go, go to breakpoint, single step, move, fill, find, scan, compare, reset,* etc.), we have added a *go next n* command, that goes through the next breakpoint *(n-1)* times before finally stopping on the *n*th time.

Another nice feature, in the *PROM*'s wakeup procedure, is that it does not automatically wait for input from the terminal. Rather it cyclically polls the terminal, the parallel port, the serial link to the host, and a pre-defined *PROM* message buffer. Thus a user on the host computer may, by activating the host's *download* program, cause the *interface* board's processor to enter it's "receiving downloader" program, either for serial or parallel transmission. It is not necessary to first enter a command from the interface board's terminal; in fact, it is not necessary to have a terminal there at all. When the program has been transferred to the interface board, the host's download program will then instruct the interface board to store a command in the *target* board's *PROM* message buffer. The target board, polling on this buffer, will receive the command, enter a subroutine in *its PROM* that copies the code from the interface board's common memory into the target board's local memory, and (if so commanded) start executing the program. This is entirely automatic, and in response to a single *download* command typed on the host.

3.4. SYS

The system process, *SYS*, consists primarily of initialization functions, including setting up the message buffer directory in common memory, writing out a table of system-wide parameters (such as the

camera calibration parameters used in the vision system), and zeroing buffer areas for easier debugging. During execution, other processes' status can by monitored by *SYS*, which then reports to the operator on exceptional situations such as the death of a process. Currently, the *SYS* initialization code is included in one of the other processes, and the monitoring functions are not performed.

3.5. Debugging Tools

The debugging *tools* are provided to assist in program development. These provide user-friendly displays of the state of the common memory buffers and related system flags, allow variables to be examined and/or changed by name, obtain a stack trace (listing the sequence of nested subroutine calls), remove or restore named subroutines or execute them in isolation, and set debug flags which release additional output during program execution.

4. Protocol for Interprocessor Communications

Central to the *GRAMPS* operating system are the inter-processor communications, which are implemented as follows.

Data are passed from one processor on the bus to another via common memory (memory accessible to both processors). The data are first written into common memory by the one processor, and then read by the other. It is the principal task of the operating system's communications programs to assure first that the data is not read until the writer is finished writing, and then that the memory is not written to (by some other processor) until the reading is complete.

To achieve this, each separate data buffer in common memory is accessed by the user as though it were a *Unix* file, which must be explicitly *opened* and *closed*. (We use the word "file" interchangeably with "buffer", to emphasize the similarity with *Unix* file manipulation.) There is one important rule which the system enforces: *only one user may have access to a particular buffer at any one time*. This is generally a reasonable requirement: if one person is still writing a buffer, the other shouldn't be reading it; if he's not still writing, he should relinquish the buffer (*close* it). This requirement is relaxed in a derivative protocol described below in section 4.5.

Buffers are permanently assigned areas in common memory. The buffer names (filenames), their common memory addresses, legitimate users, etc, are specified in an *allfiles* array maintained by the system, and resident in the system buffer *parmfile*. Each process, at initialization, extracts from the *allfiles* array the information concerning those buffers for which it is a legitimate user. This information is stored internally in the process's own on-board *files* array, so that it is available for use during real-time processing without the added bus traffic and access time that would result from using the system copy. The creation of the *files* array is done by the system at the beginning of each process, and is transparent to the user. The advantage to this centralized scheme is that changes to the buffer assignments need be made only in *allfiles*, and user programs do not need to be recompiled after such changes.

4.1. Unix Framework

The basic subroutine calls (*open, close, read, write*) have been chosen to be identical to the corresponding *C/Unix* (System V) system calls. This is a convenient and widely known quasi-standard. Many programmers will be familiar with the usage, and will find the *GRAMPS* procedures easy to remember. Also, this choice allows programs to be compiled and tested under *Unix* without requiring changes when transporting them to a target board.

These basic calls, fully analogous to the corresponding *C* calls, are:

```
file_descriptor = open(filename, mode);
number_of_chars = read(file_descriptor, buffer, count);
number_of_chars = write(file_descriptor, buffer, count);
return_code = close(file_descriptor);
```

Note that the *buffer* argument above refers to the address, within the program, that one reads into or writes from. The common memory buffer is specified by the *filename* argument.

The *open*() and *close*() calls provide the mechanism for the system to enforce the rule that at most one program may have a given buffer open at any one time. Thus the *open*() function will check whether any other *CPU* has the buffer open. If so, it polls until the buffer is closed. Only then does it open the buffer and return a small positive number, which is used like a *Unix* file-descriptor, to the calling program.

Since *open*() waits until a buffer is free, an alternative call, *openn*() (open-or-return-after-n-tries), provides a way to try to open a buffer but to continue with other processing if it is busy. A return of -2 from *openn* means the buffer is busy; this is *not* an error condition.

Error conditions are flagged by -1, and should be checked for as in *Unix*. However, error messages (and optionally stack traces) are printed by the system, so the user need not add additional code to print messages. Successful *read*() and *write*() calls return the number of bytes transferred. *read*() and *write*() check the system's flag area to be sure the file has been opened, and to the correct user; this checking can be turned off, to achieve optimum speed, once a program has been sufficiently debugged. Similarly, an *open_fd*() call can replace the *open*(), to eliminate the search through the *files* array, once a buffer's location in that array has been fixed.

4.2. Protocol Implementation

The *open*() and *close*() functions are implemented by means of a one-byte flag associated with each buffer. This flag is zero when the buffer is closed, and *hex* 80 (bit 7 set) when the buffer is open. When a particular user attempts to open the buffer (put a *1* in bit 7 of the flag), he does so by means of a test-and-set (*TAS*) instruction *(Figure 1)*. This instruction, on the *680x0*, first fetches bit 7 of the specified address (the one-byte flag), and then sets bit 7 (stores a *1* in it), all the while locking other users out of the bus. This prevents anyone else from changing the flag in the time between the fetch and the store portions of the instruction.

Having stored a *1*-bit in bit 7 of the current flag, the user then determines what had been fetched from that flag by examining the negative-flag in the *680x0*'s status register. If it was a *1*, someone else had the buffer open; this user has not changed anything (he stored a *1*, but a *1* had already been there). He *must not change anything* in the buffer or its associated flags. He may wait and try again, in the case of *open*(), or continue processing, in the case of *openn*().

If he fetched a *0*, on the other hand, this means the buffer was closed. It is now open to this user, since he has changed bit 7 of the flag to *1*. He immediately puts his "user ID" in the adjacent byte, called the *previous_user* byte.

Saving the user ID of the last user to open the buffer is necessary for the initialization procedure (see next section). In addition, the *previous_user* byte has proven useful to applications programs in ascertaining whether data is "old" (last opened by the inquiring process), or "new" (last opened by the other user). They are also invaluable in debugging, for example to see who in fact last used, and perhaps is not relinquishing, a given buffer. To further aid debugging, the system maintains the time of last open and last close, for each buffer. Maintenance of these time entries can also be switched off once they are no longer needed.

Note that the flags need not be adjacent to the buffers themselves. In fact, in our vision application we have chosen to locate these flags all together, so that the status of all of the system's buffers may be seen at a single glance. This is no longer important, since the *tools* now include a *display system flags* command. [3]

4.3. Execution Time The basic protocol is very fast. Several subroutines were run on a *68020* microprocessor, running with a 25 MHz clock. The *open* and *close* primitives (including maintaining the "previous user" byte) run in 4.5 and 2.2 μsec, respectively. The complete *debug* versions, including run-time checks on the parameters and on the internal *files* array, run in 100 and 40 μsec, respectively. (These routines are written in *C* and their execution time is compiler-dependent.)

3 On certain hardware configurations the *TAS* instruction does not lock the on-board bus, but only the backplane bus linking the boards. In such cases, individual flag buffers must be located so that they are off-board to those processors needing to set the flags.

These timings may be compared with values quoted in a recent article [14] on a *multi-tasking* robot operating system. That system required 913 and 353 μsec, respectively, for *message send* and *message receive* system calls.

4.4. Buffer Initialization

Initializing the flags at startup requires care. The design requirements state that *any process may be started up independently of the others.* That is, any process may be reset and restarted while the others are running; and processes may be started in any order. Achieving this goal required that each buffer have one additional parameter, its *owner*. In cases where it is not otherwise clear which process is responsible for cleaning up a buffer's flags, the *owner* is designated to do the job.

Specifically, when a process is first started, it looks at each of its buffers (this occurs in the system prologue that is automatically linked to the user's program). If the *previous_user* byte of a particular buffer contains that user's own ID, it initializes the buffer. If the previous user was another user assigned to the buffer, the job is left to that other user. (That other user may have just written good data, for example, or he may have crashed and been restarted--this user doesn't know, only the other user knows.) Otherwise, the flags are junk (such as the *FFFF* that often appears at power up). If this process is the *owner*, it cleans up the flags; otherwise it does nothing. This algorithm identifies one and only one process to initialize any given buffer and its flags. No determination of "who gets there first" needs to be made.

During buffer initialization, there is one case in which a user must write to a flag buffer without having first formally opened the buffer. This occurs when the user is the *owner*, and finds the buffer's *OPEN/CLOSED* flag set to *OPEN*, but the *previous_user* byte is garbage (i.e. not a valid user of that buffer). He goes to open the buffer. The problem is, it may be that the *OPEN* flag is validly set in the course of another user's open of the buffer, but the other user has not yet put his own userid in the *previous_user* byte. The solution is not to allow any user to open a buffer with a garbage *previous_user* byte, or a garbage flag for that matter. If the initialization process *SYS* (see Section 3) is run, no buffers are left uninitialized.

An even more insidious case can occur if a process wakes up to find a flag set to *OPEN*, with the *previous_user* set to the process's own userid. Here again, it may be that another user is in the process of opening the buffer, and has set the flag to *OPEN* but has not yet put his userid in the *previous_user* byte (compare this with the much more likely situation, that this process had died while the file was open to him). The presence of this user's userid, although this user is just now initializing, can be explained as coming from a "previous incarnation", i.e. before he was stopped and restarted. Our solution to this ambiguity, however inelegant, is for the process just starting up to go to sleep, and then re-awaken and see if the flag has changed or not. If not, he then presumes it to be indeed open *to him*, and initializes and closes the buffer.

Buffers are zeroed at initialization by the *owner*, if he's also the one doing the initializing of the flags, or by the system process *SYS* at powerup. The various initializing functions are called by a subroutine *vmain*, which is automatically included by the linker and started by the downloader/PROM. This initialization is transparent to the user.

4.5. Multiple-Reader Buffers

The buffers described above may have any number of users permitted to access them. In the protocol just described, however, only one user is permitted to access a given file at any one time. A user, even if he is only reading, shuts out other users, even those who are also waiting only to read.

Multiple-reader buffers, on the other hand, allow several readers to have a file open, for reading only, at the same time. For writing into the file, however, a user must still be the only one to have the file open.

The fact that a buffer permits multiple readers is transparent to the user. He still uses *open* or *openn* to open a buffer, and *close* to close it. His internal *files* array is unchanged, except that now the entry *multiple* is set to *TRUE*.

The multiple-reader protocol is implemented by adding a flag called *rdusers*, which lists users to whom the file is currently open. Each bit in *rdusers* is assigned to a user, and is set if that user has the file open. To prevent contention when two users try to change the *rdusers* flag at the same time, the protocol described in the previous subsections is appplied to the *rdusers* flag (actually to the whole *flagbuf* structure containing it), instead of to the buffer. Thus *rdusers* can be changed by only one user at a time. Note that *previous_user* now refers to the last user to open the *flagbuf* structure, rather than the last user to open the message buffer.

To the system, the *open* flag (and also the *previous_user* byte) are taken to refer to just the *flagbuf* structure, rather than the whole buffer. That is, only one process may write into or read from the *flagbuf* structure at a time, namely that process that has the *flagbuf* structure open, and whose name therefore appears in the *previous_user* byte. The same *open/close* protocols are used, but they refer to the *flagbuf* structure as the entity being opened or closed.

Once the *flagbuf* structure is open to you, you may modify the *rdusers* vector (the *open* call does this automatically) to add or delete yourself as a current reader, or you may request write privileges. If you request write privileges, you will be kept waiting until all other current users have closed the buffer. No new users will be admitted until you close the buffer: *openn* will return *FILEBUSY* status (-2) to prospective new users. Again, this protocol is internal to the system functions *open*() and *close*(); the user need not even know whether a given buffer is a "multiple-reader buffer" or not.

The *open*() and *close*() of a multiple-reader buffer will clearly take longer than for a regular buffer, so the multiple-reader facility should not be used for all files. The advantage of this protocol accrues when several users all need to read the same file, and it is not updated often. An obvious candidate is *parmfile*, the system parameter file. This is written by *SYS* during initialization, and then read by all other processes as needed.

The "multiple-reader" capability is a switchable option.

4.6. Dynamic Common Memory Allocation

The use of dynamic memory allocation for interprocessor communication is quite elegant. Suppose process *A* has data that he wishes to send to processes *B*, *C*, and *D*. He requests a block of common memory from the system, in the amount he requires for his data, using the *alloccm*() call. In so doing, he also informs the system who will be using these data, namely *B*, *C*, and *D*. The system returns with a pointer to a block of memory of the requested size, and keeps a list of the prospective users.

At this point *A* is the only process that knows about (has the address of) this block of memory. He goes ahead and writes his data there; he does not have to *open* it like a buffer, since no other process even knows of its existence. When he is done writing, he passes a pointer to the block to the appropriate processes, *B*, *C*, and *D*. (He also notifies the system that he is done with the block.)

The data is now available to be read. Since no writers have access to the block, readers need not worry that the data will change. They too need not open or close the data buffer. Moreover, all the readers may consider the block to be *open* to them simultaneously. The bus hardware arbitrates each fetch request so that only one word of data is fetched at a time; no further software protocol is required.

Whenever each process is finished using the data, it notifies the system by freeing the block, with a call to *freecm*(). The system maintains the list of those processes using the block of memory (initially *A* through *D*, in the example). When the last process frees it, the system returns the block to the pool of free memory, from which it can be reallocated.

The pointers themselves are currently passed in buffers of the kind described above (requiring *opens* and *closes*--see section 4.2). Because of the specially simple nature of the messages (just a 32-bit address), the special calls *readheader* and *writeheader* are available to open, read (or write), and close with a single call. We anticipate modifying this protocol further, so that the buffers themselves will vanish, and the four-byte pointers will be placed directly in the *flagbuf* structure. Finally, with the advent of 32-bit wide bus hardware, the *opens* and *closes* can be eliminated, since the writing of the (32-bit) pointer will be done in a single, indivisible bus cycle; the reader will always read the latest pointer, and will never get garbage (i.e. 16 bits of new pointer, and 16 bits of old).

In *GRAMPS*, the dynamic common memory allocation is implemented as part of the kernel, which is resident on each process. There is no centralized system process to perform such allocation for the whole bucket. Instead, each process is given a region in common memory from which to carve out allocation blocks for himself, as needed. The alternative, of having the *SYS* process handle all allocation requests, has not been pursued. This choice is transparent to the user: he simply calls the system function *alloccm*() and receives a pointer to memory in return.

5. Additional Functions

The system also provides *I/O* services to and from the *CRT*, including *getchar*, *putchar*, *printf*, and *readnl*. In each case it first checks to see whether a *CRT* is in fact hooked up. This allows the user to include diagnostic *printf* statements in his program, while still being able to run in real-time (i.e. without the slow terminal *I/O*) simply by unplugging the terminal from the computer board.

A system clock is created by the *SYS* process, and continues to run on one board, interrupt driven and therefore transparent to the user. This clock may be read by any user by calling a function *systime()*; as currently implemented, it ticks once each millisecond. A *sleep(n)* function puts the process to sleep for approximately *n* milliseconds. *abort(message)* and *exit(n)* perform as expected.

A *PROC* (per process) array, maintained in common memory, provides a snapshot of the current state of each process. A *tools* command displays this information on request.

Every error detected by the system gives rise to a message to the *CRT* (if there is one) including the name of the subroutine and the error, and relevant parameters. If the *debug* flag is set, a stack trace is also automatically printed. Thus, while a user must check for a *-1* error return, he need not bother coding an error message. (The system error messages may be disabled, if desired, by clearing a flag.)

The same buffer protocol as described above has also been implemented for host-target data transfers, to allow programs to be debugged on the host, or to permit the host to influence execution on the target. Since the host cannot directly lock the bus, the *CPU* on the interface board (running code in its *PROM*) accepts a request from the host to do the test-and-set on the host's behalf, and to return the result to the host.

Additional buffer ("file") functions are provided, such as *readran* and *writeran*, which provide random access, and *autoread* and *autowrite*, which include the *open* and *close* functions automatically, and so result in fewer programmer errors. *Open_synch*, together with the global variables *synchr_base* and *synchr_incr*, allows the user to require that message passing be done at fixed times only (synchronously).

These and other functions are described, and directions for their use is given, in *The GRAMPS Operating System: User's Guide* [12]. Details of how to set up the *files* arrays, user IDs, etc, will be presented in *The GRAMPS Operating System: Administrator's Guide* [13].

6. Application

GRAMPS -- an earlier, less developed version -- was used as the operating system in the vision system component of the National Bureau of Standards Automated Manufacturing Research Facility *(AMRF)*. The *AMRF* is a fully-automated machine-tool shop designed as a demonstration and test-bed facility. The vision system was built and tested in 1982-1985, and continues in operation today. Hardware consists of five *CPU* boards running *Intel 8086* microprocessors in a *Multibus* backplane, and a separate host computer consisting of an *8086 CPU* board in an *S-100* backplane, communicating with the *Multibus* bucket in real-time as discussed in the previous section.

Program execution in the vision system was data-driven. Each processor performed a different function on the image. One did run-length encoding of the image (see Ref. [3]), the next performed segmentation, the third extracted features such as corners, etc. Images arrived from the camera 30 times each second. Each image was processed by the first processor. The output of this stage was then passed (via the *GRAMPS* operating system) to the next, etc. The receiving processor is always available (at its programmer's discretion) to deal with the incoming data. There is never any concern that it might be swapped out of memory, or when it will again gain control of the processor, or how to preserve the data until it is again running. Interrupts were not used; each process simply checked to see if new data was available, at appropriate points in its code. The operating system's responsibility is simply to provide

bookkeeping for the data passing, and to assure that there are no collisions. *GRAMPS* does this with low overhead, and what overhead there is occurs at points explicitly recognized by the user, i.e. during a call to *read* or *write*.

Since then *GRAMPS* has been converted to run on *680x0* processors, and its capabilities have been greatly enhanced. Three processes of the *AMRF* vision system were transported to three *68010*'s, where they ran under *GRAMPS* as before. Other programs were written to test some of the new capabilities of *GRAMPS*, and they demonstrated the usefulness of these new features. *GRAMPS* has thus proved itself to be dependable, as well as fast, on both *8086*'s and *680x0*'s.

7. Acknowledgments

The author wishes to acknowledge the following people for their valuable help and their contributions to the development of *GRAMPS*: Jonathan Brickman, Ernie Kent, Mark Rosol, Wally Rutkowski, Mike Shneier, and Tom Wheatley, and to thank those others who used *GRAMPS* before it was fully debugged, and by so doing, helped to debug it.

REFERENCES

[1] Kent, Ernest W., and Albus, James S., "Servoed World Models as Interfaces between Robot Control Systems and Sensory Data", *Robotica* 2, pp. 17-25 (1984).

[2] Shneier, Michael O., Lumia, Ronald, and Herman, Martin, "Prediction-Based Vision for Robot Control", *Computer Magazine*, August 1987, pp. 46-55.

[3] Albus, J., Kent, E., Mansbach, P., Nashman, M., Palombo, L, and Shneier, M., "Six-Dimensional Vision System", *SPIE* 336, pp. 142-153 (1982).

[4] Schwan, Karsten, Bihari, Tom, Weide, Bruce W., and Taulbee, Gregor, "GEM: Operating System Primitives for Robots and Real-Time Control Systems", *Proc. IEEE Intl. Conf. Robotics*, pp. 807-813 (1985).

[5] Gentleman, W.M., *Using the Harmony Operating System*, ERB-966 (NRCC No. 24685), National Research Council of Canada, Ottawa, Ont. (1985).

[6] Gaglianello, Robert D., and Katseff, Howard P., "A Distributed Computing Environment for Robotics", *Proc. IEEE Intl. Conf. Robotics and Automation*, pp. 1890-1896 (1986).

[7] *MTOS-UX User's Guide for the 680XX*, Industrial Programming Inc., Jericho, NY (1984, rev. 1986).

[8] Bracho, Rafael, Schlag, John F., and Sanderson, Arthur C., *POPEYE: A Gray-Level Vision System for Robotics Applications*, Technical Report CMU-RI-TR-83-6, Carnegie-Mellon Univ., Pittsburgh, PA (1983).

[9] *pSOS-68K User's Manual*, and *pRISM-68K User's Manual*, Software Components Group, Inc., Santa Clara, CA (1986).

[10] *VRTX/86 User's Guide* and *Multi-Processor Applications Using VRTX*, Hunter & Ready, Inc., Palo Alto, CA (1984).

[11] *VxWorks Overview* and *VxWorks: A Real-Time Partner for Unix*, Wind River Systems Inc., Emeryville, CA (1986).

[12] Mansbach, Peter, and Shneier, Michael, *The GRAMPS Operating System: User's Guide*, NBSIR 88-3776, National Bureau of Standards, Gaithersburg, MD (1988).

[13] Mansbach, Peter, and Shneier, Michael, *The GRAMPS Operating System: Administrator's Guide*, National Bureau of Standards, Gaithersburg, MD *(to be publ.)*.

[14] Lee, Insup, King, Robert B., and Paul, Richard P., "Kernel for Distributed Multisensor Systems", *Computer Magazine* pp. 78-83 (June 1989).

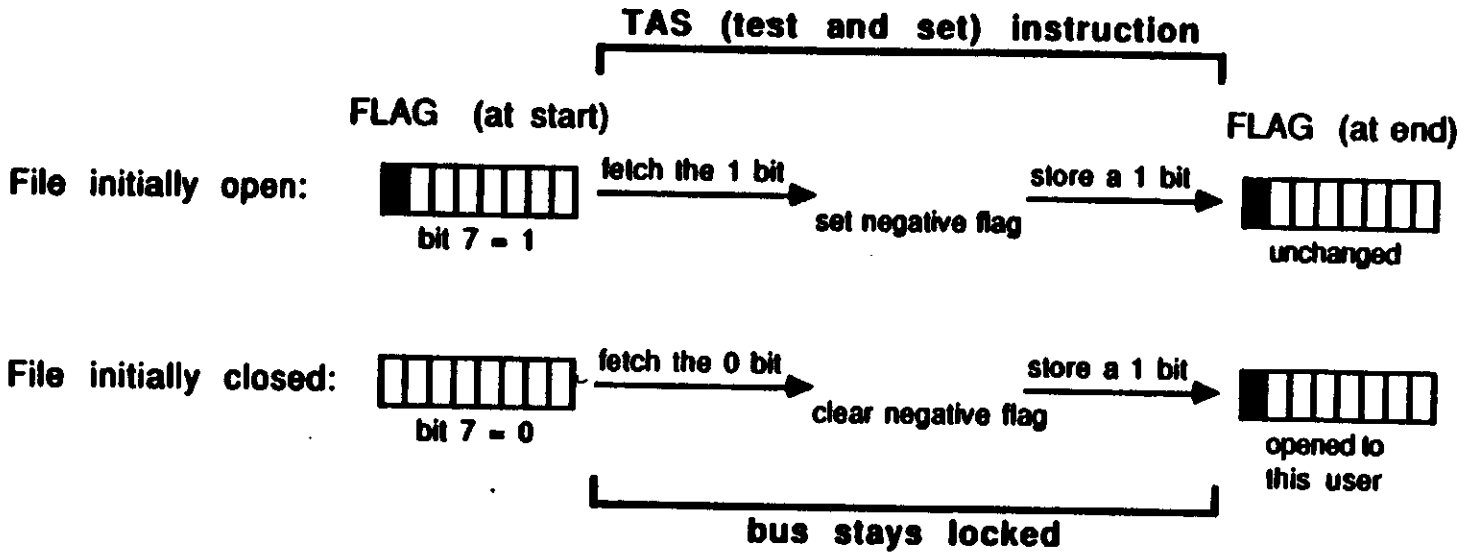

Fig. 1. Diagram of *TAS* (test and set) instruction.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NISTIR 89-4190	2. Performing Organ. Report No.	3. Publication Date January 1990

4. TITLE AND SUBTITLE

Overview of the "GRAMPS" Multiprocessor Operating System.

5. AUTHOR(S)

Peter Kansbach

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
U.S. DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

National Bureau of Standards
Robot System Division
Building #220, B-124
Gaithersburg, MD 20899

10. SUPPLEMENTARY NOTES

☑ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

GRAMPS is an operating system designed for use by a number of independent functionally divided processors which communicate via common (shared) memory. Three mechanisms of asynchronous data communication between processors are provided. These include a set of Unix-compatible calls, so a program can be debugged in a Unix environment and simply re-linked to run on a target board. GRAMPS allows individual processors to be stopped and restarted, without interrupting the other processors, and allows single processors to be run in isolation. The operating system includes a fast downloader, a monitor (in PROM), and an array of debugging tools. GRAMPS provides an extremely fast system for single-tasking multiple-processor applications and some multi-tasking capability as well. It is currently running on Motorola 680x0 microprocessor, and has also run on Intel 8086's.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

asynchronous communication; communications protocol; functionally-divided processes; GRAMPS; multi-processing; multi-processor; multiprocessing; multiprocessor; operating system; real-time; robot vision; vision

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES 14

15. Price A02

USCOMM-DC 6043-P80